A Formalisation of Java's Exception Mechanism

Bart Jacobs

Dep. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
bart@cs.kun.nl   http://www.cs.kun.nl/~bart

Abstract. This paper examines Java's exception mechanism, and formalises its main operations (`throw`, `try-catch` and `try-catch-finally`) in a type-theoretic setting. This formalisation uses so-called coalgebras for modeling Java statements and expressions, thus providing a convenient setting for handling the various termination options that may arise in exception handling (closely following the Java Language Specification). This semantics of exceptions is used within the LOOP project on Java program verification. It is illustrated in two example verifications in PVS.

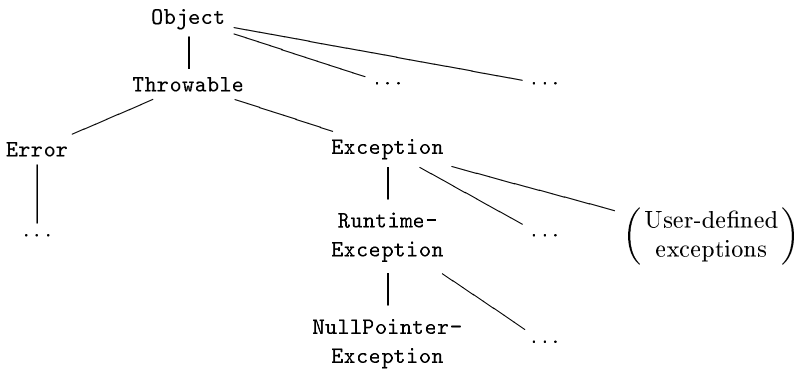
1 Introduction

The LOOP project [27] at the University of Nijmegen aims at Java program verification using proof tools (such as PVS [23] and Isabelle [24]) and a special purpose front-end compiler (the so-called LOOP tool, see [3]) for translating Java classes into the logic of the back-end proof tools. Incorporated in this LOOP tool is a semantics of (sequential) Java in the higher order logic of PVS and Isabelle. A distinguishing feature of this semantics is its mathematical basis given by so-called coalgebras. Several aspects of this semantics have already been described elsewhere (see [15, 2, 10, 9, 8]), but the semantics of exceptions has not been published yet. It will be the topic of the present paper. The aim of the formalisation is to (1) clarify the existing informal specification, and (2) provide a semantical basis for (tool-assisted) verification of Java programs. Currently, the main application area is JavaCard [25, 26].

As in earlier publications we shall not describe Java semantics in the language of PVS or of Isabelle/HOL, but in a type-theoretic common abstraction of these, which incorporates the essentials of higher order logic. It is described briefly in Section 2 below. The main type constructors are labeled product and coproduct, function space and list. For more information, see *e.g.* [8]. Higher order logic is too much for what we need in this paper, but since it exists both in PVS and Isabelle/HOL, we take it for granted.

Exceptions form an integrated aspect of the Java programming language, which can contribute to the reliability and robustness of programs written in Java—if the semantics of the exception mechanism is clear. Exceptions occur in programs when certain constraints are violated, *e.g.* a division by zero, an array access out of the arrays bounds, an object creation when there is no unused memory left, or a situation which is seen as unexpected or inappropriate by the programmer. The occurrence of an exception in a program leads to what is called abrupt termination[1]. It means that all subsequent statements are skipped (and locks are released), until (possibly) an exception handler is reached. One says that an exception "is thrown" at the point where it occurs, and "is caught" at the point where it is handled. As we shall see, exception handling is based on the exceptions type. It will restore normal operation[2], when the exception is handled properly. The Java exception mechanism is integrated with the synchronisation model, but that will not be relevant here: we only consider what it means when exceptions are thrown or caught, and not how this affects the flow of control in a multi-threaded scenario.

We describe a part of Java's pre-defined exception hierarchy, with superclasses sitting above subclasses.



The class `Throwable` is a direct subclass of the root class `Object`. It has two subclasses, `Error` and `Exception`. Errors (instances of `Error`) are exceptions from which programs are not ordinarily expected to recover [7, §§11.5]. Instances of `Error` and `RuntimeException` are special because they are the only so-called unchecked exceptions. For all other, checked, exceptions the Java compiler makes sure that each method either handles this exception (via a `catch` statement) or declares it in its method header, as in: `void m() throws IOException` {...}. This `throws` clause may be understood as a contract between the implementor and the user (in the style of Design-by-Contract [20]), see [7, §§11.2]. An overriding method in a subclass must respect the `throws` clause of the method that is being overridden in the superclass, *i.e.* cannot throw more exceptions. Whether or not an exception is checked does not play a rôle for the Java semantics within the LOOP project.

[1] A `return`, `break` or `continue` statement in Java also leads to abrupt termination.

[2] Normal termination is not restored at the point where the exception arises: Java has a so-called termination model for exceptions, and not a resumption model, see [4, §16.4].

The semantics of programming languages with exceptions forms a good illustration of the appropriateness of using coalgebras to organise the relevant structure (via different termination modes, distinguished via coproduct types). In general, a coalgebra is a "transition" function of the form $S \longrightarrow \boxed{\cdots S \cdots}$ with a structured result type that captures a certain kind of computation, where S is a set of states. See [14] for an introduction. Such a semantics can also be described in terms of monads [13]. The monadic view emphasises the input-output relation, whereas the coalgebraic view emphasises the state-based aspect of the computations—and thus leads to notions like invariant and bisimilarity (which will not be used here), but also to a logic with appropriate modalities, which we shall briefly describe here as a Hoare logic (like in [10, 12]). The advantage of this coalgebraic approach—and the reason why we emphasise it—is that the type system forces one to explicitly handle all possible termination options (in the box above). See for instance the many cases in the definitions of TRY-CATCH and TRY-CATCH-FINALLY in Section 5 below, closely corresponding to the cases that are distinguished in the Java Language Specification [7]. A very different alternative is to incorporate exceptions into one's state space, like in the continuation-based approach of [1] or the operational and axiomatic approaches of [22, 21]. This simplifies the type of state transformers, at the expense of complicating the state space (certainly when the other forms of abrupt termination are taken into account), and makes the handling of the various cases less transparent. The axiomatic semantics of exceptions is studied in for example [5, 18, 17] (mostly via a weakest precondition calculus), involving a single possible exception, and not many forms of abrupt termination (like in Java).

This paper starts with two introductory sections. First there is a brief account of the simple type theory that will be used, concentrating on labeled (co)products. Next, the (coalgebraic) representation of Java statements and expressions is explained, together with an associated Hoare logic dealing with the different termination modes. This forms the basis for the formalisations of exception throwing in Section 4 and exception handling in Section 5. The latter section has two parts, one for `try-catch` and one for `try-catch-finally`. Each part contains an extensive quote from the Java Language Specification [7], containing the informal explanations of exception handling. Subsequently, Section 6 describes two example programs involving some tricky aspects of exception handling. Appropriate specifications are provided in the language JML [16], and proved (after translation by the LOOP tool) in PVS.

2 A brief look at the type theory

The type theory that we use is the same as in [2, 10, 9, 8]. It has some basic types like bool, string and unit (for a singleton type), plus function types, labeled products and coproducts, list *etc.* as type constructors. We assume that these are more or less familiar, and only wish to mention the notation we use for labeled (co)product and function types.

Given types $\sigma_1, \ldots, \sigma_n$, we can form a product (or record) type $[\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$ and a labeled coproduct (or variant) type $\{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$, where all labels lab_i are assumed to be different. An example is the well-known lift type constructor $\mathsf{lift}[\alpha] = \{\,\mathsf{bot}\colon \mathsf{unit} \mid \mathsf{up}\colon \alpha\,\}$ which adds a bottom element to an arbitrary type α. For terms $M_i : \sigma_i$, there is a labeled tuple $(\,\mathsf{lab}_1 = M_1, \ldots, \mathsf{lab}_n = M_n\,)$ inhabiting the corresponding product type $[\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$. For a term $N : [\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$ in this product type, we write $N.\mathsf{lab}_i$ for the selection term of type σ_i. Similarly, for a term $M : \sigma_i$ there is a labeled or tagged term $\mathsf{lab}_i\, M$ in the coproduct type $\{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$. And for a term $N : \{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$ in this coproduct type, together with n terms $L_i : \tau$ containing a free variable $x_i : \sigma_i$ there is a case term $\mathsf{CASES}\ N\ \mathsf{OF}\ \{\,\mathsf{lab}_1\, x_1 \mapsto L_1 \mid \ldots \mid \mathsf{lab}_n\, x_n \mapsto L_n\,\}$ of type τ which binds the x_i. For function types we shall use the standard notation $\lambda x\colon \sigma.\, M$ for lambda abstraction and $N \cdot L$ for application.

3 Basics of Java semantics

As described earlier, the LOOP tool provides a semantics for (sequential) Java by translating Java classes into the higher order logic of PVS or Isabelle. This section will introduce the basic aspects of the semantics and provide the setting for the description of exception handling in the remainder of the paper. It will concentrate on some special types, on the (coalgebraic) representation of statements and expressions, and on some basic language constructs.

A memory model is constructed as a specific type OM, for object memory. It consists of a heap, a stack, and static memory, each consisting of an infinite series of memory cells. These memory cells can store the contents of objects and arrays. The type OM comes with various put and get operations for reading and writing in the object memory. Its precise structure is not so relevant for what follows, and the interested reader is referred to [2] for more information. Elements of OM will often be called states.

References will be values of the following type.

– TYPE THEORY

$$\mathsf{RefType} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \{\,\mathsf{null}\colon \mathsf{unit} \mid \mathsf{ref}\colon \mathsf{MemLoc}\,\}$$

Thus a reference is either a null-reference, or a non-null-reference consisting of a memory location (inhabiting an appropriate type MemLoc) pointing to a memory cell on the heap. In [2] we have included type information in references, but here we shall assume it to be part of memory cells. Therefore, there is a function

– TYPE THEORY

$$\mathsf{gettype}\colon \mathsf{MemLoc} \to \mathsf{OM} \to \mathsf{ClassName} \qquad \text{where} \qquad \mathsf{ClassName} \stackrel{\mathrm{def}}{=} \mathsf{string}$$

which gives for a specific memory location p the type of the object stored at p on the heap. This type is represented as a string. There is also a special predicate

– TYPE THEORY

$$\mathsf{SubClass?} \colon \mathsf{ClassName} \to \mathsf{ClassName} \to \mathsf{bool} \tag{1}$$

incorporating the subtype relationship between classes, given as strings.

Statements and expressions in Java may have different termination modes: they can hang (*e.g.* because of an infinite loop), terminate normally, or terminate abruptly (typically because of an exception, but (statements) also because of a `return`, `break` or `continue`). All these options are captured in appropriate datatypes. First, abnormal termination leads to the following two types, one for statements and one for expressions.

– TYPE THEORY

$$\begin{array}{l} \mathsf{StatAbn} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad \{\, \mathsf{excp} \colon [\, \mathsf{es} \colon \mathsf{OM}, \mathsf{ex} \colon \mathsf{RefType} \,] \\ \quad \mid \mathsf{rtrn} \colon \mathsf{OM} \\ \quad \mid \mathsf{break} \colon [\, \mathsf{bs} \colon \mathsf{OM}, \mathsf{blab} \colon \mathsf{lift}[\mathsf{string}] \,] \\ \quad \mid \mathsf{cont} \colon [\, \mathsf{cs} \colon \mathsf{OM}, \mathsf{clab} \colon \mathsf{lift}[\mathsf{string}] \,] \,\} \end{array} \qquad \begin{array}{l} \mathsf{ExprAbn} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad [\, \mathsf{es} \colon \mathsf{OM}, \mathsf{ex} \colon \mathsf{RefType} \,] \end{array}$$

These types are used to define the result types of statements and expressions:

– TYPE THEORY

$$\begin{array}{l} \mathsf{StatResult} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad \{\, \mathsf{hang} \colon \mathsf{unit} \\ \quad \mid \mathsf{norm} \colon \mathsf{OM} \\ \quad \mid \mathsf{abnorm} \colon \mathsf{StatAbn} \,\} \end{array} \qquad \begin{array}{l} \mathsf{ExprResult}[\alpha] : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad \{\, \mathsf{hang} \colon \mathsf{unit} \\ \quad \mid \mathsf{norm} \colon [\, \mathsf{ns} \colon \mathsf{OM}, \mathsf{res} \colon \alpha \,] \\ \quad \mid \mathsf{abnorm} \colon \mathsf{ExprAbn} \,\} \end{array}$$

A Java statement is then translated as a state transformer function $\mathsf{OM} \to \mathsf{StatResult}$, and a Java expression of type `Out` as a function $\mathsf{OM} \to \mathsf{ExprResult}[\mathsf{Out}]$. Thus both statements and expressions are coalgebras. The result of such functions applied to a state $x \colon \mathsf{OM}$ yields either hang, norm, or abnorm (with appropriate parameters), indicating the sort of outcome.

On the basis of this representation of statements and expressions all language constructs from (sequential) Java are translated. For instance, the composition of two statements is defined as:

– TYPE THEORY

$$
\begin{array}{l}
s, t\colon \mathsf{OM} \to \mathsf{StatResult} \vdash \\
\quad (s\,;t)\ :\ \mathsf{OM} \to \mathsf{StatResult} \stackrel{\mathrm{def}}{=} \\
\qquad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto t \cdot y \\
\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\, \}
\end{array}
$$

What is important to note is that if s hangs or terminates abruptly, then so does the composition $s\,;t$. In particular, if an exception is thrown, subsequent statements are not executed.

Recall that `Throwable` is the root class of all exceptions. Its constructors call a native method for creating an exception object. In the LOOP semantics there is a corresponding function, called MAKE-EXCEPTION. It takes a string as argument, for the exceptions message, and performs some basic memory operations: allocating an appropriate new memory cell on the heap, and storing the message[3]. We skip the details of MAKE-EXCEPTION and only mention its type:

– TYPE THEORY

$$\mathsf{MAKE\text{-}EXCEPTION}\colon \mathsf{string} \to \mathsf{OM} \to [\,\mathsf{es}\colon \mathsf{OM}, \mathsf{ex}\colon \mathsf{RefType}\,]$$

It takes a string and a state, and produces an appropriately adapted return state together with a (non-null) reference to the exception object that it created in the return state.

Exception classes in the Java API typically call the constructors from `Throwable` to create new instances. Therefore we can also use MAKE-EXCEPTION for these classes directly.

3.1 Specifications with exceptions

The coalgebraic representation of statements and expressions formalises the different termination modes that can occur. It naturally gives rise to a Hoare logic with different, corresponding modes for reasoning about "normal" and "abnormal" states, see [10]. For example, there is a partial Hoare triple:

$$\{\mathsf{pre}\}\ \mathsf{stat}\ \{\mathsf{exception}(E, \mathsf{post})\}$$

Informally, it says that if the precondition pre holds and the statement stat terminates abruptly by throwing a non-null exception (see (2) below), this exception belongs to class E and the postcondition post holds. More formally,

[3] Our semantics does not take the `backtrace` field in `Throwable` into account.

— TYPE THEORY

$$
\begin{array}{l}
\mathsf{pre}\colon \mathsf{OM} \to \mathsf{bool},\ \mathsf{post}\colon \mathsf{OM} \to \mathsf{RefType} \to \mathsf{bool}, \\
\qquad \mathsf{stat}\colon \mathsf{OM} \to \mathsf{StatResult},\ E\colon \mathsf{ClassName}\ \vdash \\[1ex]
\quad \{\mathsf{pre}\}\ \mathsf{stat}\ \{\mathsf{exception}(E, \mathsf{post})\}\ :\ \mathsf{bool}\ \stackrel{\text{def}}{=} \\
\quad\quad \forall x\colon \mathsf{OM}.\ \mathsf{pre}\cdot x\ \Rightarrow\ \mathsf{CASES}\ \mathsf{stat}\cdot x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \\
\qquad\qquad\qquad\qquad\quad \mathsf{CASES}\ a\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{excp}\, e \mapsto \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{CASES}\ e.\mathsf{ex}\ \mathsf{OF}\{ \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{null} \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{ref}\, p \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{SubClass?}\cdot(\mathsf{gettype}\cdot p\cdot(e.\mathsf{es}))\cdot E \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathsf{post}\cdot(e.\mathsf{es})\cdot(e.\mathsf{ex})\ \} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{rtrn}\, z \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{break}\, b \mapsto \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{cont}\, c \mapsto \mathsf{true}\ \}\ \}
\end{array}
$$

Notice that the postcondition has type $\mathsf{OM} \to \mathsf{RefType} \to \mathsf{bool}$ and can thus also say something about the exception object (like in the example in Subsection 6.2). Similar such Hoare triples can be defined for the other termination modes. They are essential for reasoning about Java programs, for example for proving a suitable postcondition for a program which involves an exception inside a `while` loop, see *e.g.* [11].

These different termination modes also occur in the behavioural interface specification language JML [16] that will be used in Section 6. JML has pre- and post-conditions which can be used to describe "normal" and "exceptional" behaviour. The LOOP tool translates these JML specifications into suitable Hoare formulas, combining several termination options, see [12] for details.

4 Throwing exceptions

A programmer in Java can explicitly throw an exception via the command `throw` *Expression*, where *Expression* should belong to `Throwable`, or one of its subclasses. This statement will immediately lead to abrupt termination. The Java Language Specification [7, §§14.17] says:

> A `throw` statement first evaluates the *Expression*. If the evaluation of the *Expression* completes abruptly for some reason, then the `throw` completes abruptly for that reason. If evaluation of *Expression* completes normally, producing a non-`null` value *V*, then the `throw` statement completes abruptly, the reason being a `throw` with value *V*. If evaluation of the *Expression* completes

> normally, producing a `null` value, then an instance *V'* of class `NullPointer-Exception` is created and thrown instead of `null`. The `throw` statement then completes abruptly, the reason being a `throw` with value *V'*.

The LOOP tool uses the following translation of throw statements.

$$[\![\, \texttt{throw}\ \mathit{Expression}]\!] \stackrel{\text{def}}{=} \mathsf{THROW} \cdot [\![\, \mathit{Expression}]\!]$$

The function THROW captures the above explanation in ordinary language in a type-theoretic formulation.

– TYPE THEORY

$$
\begin{array}{l}
e\colon \mathsf{OM} \to \mathsf{ExprResult}[\mathsf{RefType}] \ \vdash \\
\quad \mathsf{THROW} \cdot e \ : \ \mathsf{OM} \to \mathsf{StatResult} \ \stackrel{\text{def}}{=} \\
\quad\quad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ e \cdot x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{CASES}\ y.\mathsf{res}\ \mathsf{OF}\{ \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{null} \mapsto \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{LET}\ d = \mathsf{MAKE\text{-}EXCEPTION} \cdot \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\text{“NullPointerException”}) \cdot (y.\mathsf{ns}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{IN}\ \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = d.\mathsf{es}, \mathsf{ex} = d.\mathsf{ex})) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{ref}\, p \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = y.\mathsf{ns}, \mathsf{ex} = \mathsf{ref}\, p))\ \} \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\ \}
\end{array}
$$

Interestingly, the formalisations within the LOOP project and the Bali project (see [22, p. 123]) revealed an omission in the first edition of the Java Language Specification [6, §§14.16]: the case where *Expression* evaluates to a null-reference was not covered. Following a subsequent suggestion for improvement, this was repaired in the second edition [7] (as described in the quote above).

There is an important implicit assumption about Java related to this, namely:

$$\boxed{\text{A thrown exception is never a null-reference.}} \qquad (2)$$

This “invariant” holds clearly for exceptions thrown by users (as can be seen from the definition of THROW, or the explanation of `throw`), but also holds for exceptions that are thrown by the Java Virtual Machine (both for synchronous and asynchronous exceptions), see [19]. It seems that this assumption has not been made explicit before (but it is hard-wired into the Bali semantics [22, 21]: there it automatically holds because of a syntactic distinction between valid locations and Null; exceptions can only return valid locations). It will play a rôle in the way we formalise the catching mechanism.

5 Catching exceptions

For neatly handling possible exceptional cases in a statement `S`, Java uses `try S` followed by a series of `catch` blocks for different exceptions, possibly followed by a `finally` block. When `S` terminates normally, no `catch` block is executed, but the `finally` block is (if any). If `S` results in an exception, say belonging to class `E`, the first `catch` block in the series that handles `E`-exceptions is executed, followed by the `finally` block (if any).

The list of catches in a `try` statement will be translated into a list (in type theory) consisting of pairs of strings (with label exc) and functions (with label handler) from RefType to statements for the corresponding handler code. The possible input of these functions is a reference to the exception thrown by the `try` statement. The parameter exceptions are treated as local variables. These are initialised to the RefType input of the handler function. The interpretations used by the LOOP tool look as follows.

$$
\begin{array}{l}
[\![\, \texttt{try\{tb\}catch(E1 e1)\{h1\}...catch(En en)\{hn\}} \,]\!] \\
\stackrel{\text{def}}{=} \mathsf{TRY\text{-}CATCH} \cdot [\![\, \texttt{tb} \,]\!] \cdot \\
\qquad [\, (\, \mathsf{exc} = \text{“}\texttt{E1}\text{”}, \\
\qquad\quad \mathsf{handler} = \lambda v_1 : \mathsf{RefType}.\, [\![\, \texttt{E1 e1 = } v_1\texttt{; h1} \,]\!] \,), \\
\qquad\qquad \vdots \\
\qquad\ \ (\, \mathsf{exc} = \text{“}\texttt{En}\text{”}, \\
\qquad\quad \mathsf{handler} = \lambda v_n : \mathsf{RefType}.\, [\![\, \texttt{En en = } v_n\texttt{; hn} \,]\!] \,)\,]
\end{array}
$$

$$
\begin{array}{l}
[\![\, \texttt{try\{tb\}catch(E1 e1)\{h1\}...catch(En en)\{hn\}finally\{fb\}} \,]\!] \\
\stackrel{\text{def}}{=} \mathsf{TRY\text{-}CATCH\text{-}FINALLY} \cdot [\![\, \texttt{tb} \,]\!] \cdot \\
\qquad [\, (\, \mathsf{exc} = \text{“}\texttt{E1}\text{”}, \\
\qquad\quad \mathsf{handler} = \lambda v_1 : \mathsf{RefType}.\, [\![\, \texttt{E1 e1 = } v_1\texttt{; h1} \,]\!] \,), \\
\qquad\qquad \vdots \\
\qquad\ \ (\, \mathsf{exc} = \text{“}\texttt{En}\text{”}, \\
\qquad\quad \mathsf{handler} = \lambda v_n : \mathsf{RefType}.\, [\![\, \texttt{En en = } v_n\texttt{; hn} \,]\!] \,)\,] \cdot \\
\qquad [\![\, \texttt{fb} \,]\!]
\end{array}
$$

The two type-theoretic functions TRY-CATCH and TRY-CATCH-FINALLY used for these interpretations will be described separately. They involve many subtle case distinctions, which are not easy to understand without direct access to the relevant descriptions of the Java Language Specification. Therefore, these are included.

5.1 Try-catch

The Java Language Specification [7, §§14.19.1] says:

> A `try` statement without a `finally` block is executed by first executing the `try` block. Then there is a choice:

- If execution of the `try` block completes normally, then no further action is taken and the `try` statement completes normally.
- If execution of the `try` block completes abruptly because of a throw of a value V, then there is a choice:
  - If the run-time type of V is assignable (§5.2) to the *Parameter* of any `catch` clause of the `try` statement, then the first (leftmost) such `catch` clause is selected. The value V is assigned to the parameter of the selected `catch` clause, and the Block of that `catch` clause is executed. If that block completes normally, then the `try` statement completes normally; if that block completes abruptly for any reason, then the `try` statement completes abruptly for the same reason.
  - If the run-time type of V is not assignable to the parameter of any `catch` clause of the `try` statement, then the `try` statement completes abruptly because of a `throw` of the value V.
- If execution of the `try` block completes abruptly for any other reason, then the `try` statement completes abruptly for the same reason.

This behaviour will be realised by the TRY-CATCH function below. It first executes its first argument s (the meaning of the try block), and then, when an exception occurs, it calls a recursive function TRY-LOOP; otherwise it does nothing else. By the earlier mentioned invariant (2), this exception can be assumed to be a non-null reference. Therefore we can choose an arbitrary outcome (hang) when the null reference case is distinguished.

– TYPE THEORY

$$
\begin{array}{l}
s\colon \mathsf{OM} \to \mathsf{StatResult},\\
\ell\colon \mathsf{list}[[\,\mathsf{exc}\colon \mathsf{ClassName}, \mathsf{handler}\colon \mathsf{RefType} \to \mathsf{OM} \to \mathsf{StatResult}\,]] \;\vdash\\
\quad \mathsf{TRY\text{-}CATCH} \cdot s \cdot \ell :\; \mathsf{OM} \to \mathsf{StatResult} \stackrel{\mathrm{def}}{=}\\
\qquad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{\\
\qquad\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang}\\
\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}\, y\\
\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto\\
\qquad\qquad\qquad\quad \mathsf{CASES}\ a\ \mathsf{OF}\ \{\\
\qquad\qquad\qquad\qquad \mid \mathsf{excp}\, e \mapsto\\
\qquad\qquad\qquad\qquad\quad \mathsf{CASES}\ e.\mathsf{ex}\ \mathsf{OF}\ \{\\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{null} \mapsto \mathsf{hang} \quad \text{// don't care, see (2)}\\
\qquad\qquad\qquad\qquad\qquad \mid \mathsf{ref}\, r \mapsto \mathsf{TRY\text{-}LOOP} \cdot r \cdot \ell \cdot (e.\mathsf{es})\,\}\\
\qquad\qquad\qquad\qquad \mid \mathsf{rtrn}\, z \mapsto \mathsf{rtrn}\, z\\
\qquad\qquad\qquad\qquad \mid \mathsf{break}\, b \mapsto \mathsf{break}\, b\\
\qquad\qquad\qquad\qquad \mid \mathsf{cont}\, c \mapsto \mathsf{cont}\, c\,\}\,\}
\end{array}
$$

The TRY-LOOP function recursively goes through the list of exception class names and corresponding handler functions, checking whether an exception is assignable to a parameter. It uses the SubClass? predicate from (1). If the end of the list is reached and the exception is still not handled, it is returned.

— TYPE THEORY

$$
\begin{array}{l}
p\colon \mathsf{MemLoc}, \ell\colon \mathsf{list}[[\,\mathsf{exc}\colon \mathsf{ClassName}, \mathsf{handler}\colon \mathsf{RefType} \to \mathsf{OM} \to \mathsf{StatResult}\,]] \;\vdash \\
\quad \mathsf{TRY\text{-}LOOP} \cdot p \cdot \ell \;:\; \mathsf{OM} \to \mathsf{StatResult} \;\stackrel{\mathrm{def}}{=} \\
\qquad \lambda x\colon \mathsf{OM}.\ \mathsf{CASES}\ \ell\ \mathsf{OF}\ \{ \\
\qquad\qquad \mid \mathsf{nil} \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = x, \mathsf{ex} = \mathsf{ref}\, p)) \\
\qquad\qquad \mid \mathsf{cons}(h, t) \mapsto \mathsf{IF}\ \mathsf{SubClass?} \cdot (\mathsf{gettype} \cdot p \cdot x) \cdot (h.\mathsf{exc}) \\
\qquad\qquad\qquad\qquad \mathsf{THEN}\ (h.\mathsf{handler}) \cdot (\mathsf{ref}\, p) \cdot x \\
\qquad\qquad\qquad\qquad \mathsf{ELSE}\ \mathsf{TRY\text{-}LOOP} \cdot p \cdot t \cdot x \\
\qquad\qquad\qquad\qquad \mathsf{ENDIF}\,\}
\end{array}
$$

5.2 Try-catch-finally

Again, our starting point is the Java Language Specification [7, §§14.19.2]. Now there are many more cases to be distinguished.

> A `try` statement with a `finally` block is executed by first executing the `try` block. Then there is a choice:
>
> - If execution of the `try` block completes normally, then the `finally` block is executed, and then there is a choice:
>   - ♦ If the `finally` block completes normally, then the `try` statement completes normally.
>   - ♦ If the `finally` block completes abruptly for reason S, then the `try` statement completes abruptly for reason S.
> - If execution of the `try` block completes abruptly because of a `throw` of a value V, then there is a choice:
>   - ♦ If the run-time type of V is assignable to the parameter of any `catch` clause of the `try` statement, then the first (leftmost) such `catch` clause is selected. The value V is assigned to the parameter of the selected `catch` clause, and the Block of that `catch` clause is executed. Then there is a choice:
>   - ∗ If the `catch` block completes normally, then the `finally` block is executed. Then there is a choice:
>   - · If the `finally` block completes normally, then the `try` statement completes normally.
>   - · If the `finally` block completes abruptly for any reason, then the `try` statement completes abruptly for the same reason.
>   - ∗ If the `catch` block completes abruptly for reason R, then the finally block is executed. Then there is a choice:
>   - · If the `finally` block completes normally, then the `try` statement completes abruptly for reason R.
>   - · If the `finally` block completes abruptly for reason S, then the `try` statement completes abruptly for reason S (and reason R is discarded).
>   - ♦ If the run-time type of V is not assignable to the parameter of any `catch` clause of the `try` statement, then the `finally` block is executed. Then there is a choice:

  - ∗ If the `finally` block completes normally, then the `try` statement completes abruptly because of a `throw` of the value V.
  - ∗ If the `finally` block completes abruptly for reason S, then the `try` statement completes abruptly for reason S (and the `throw` of value V is discarded and forgotten).
- If execution of the `try` block completes abruptly for any other reason R, then the `finally` block is executed. Then there is a choice:
  - ♦ If the `finally` block completes normally, then the `try` statement completes abruptly for reason R.
  - ♦ If the `finally` block completes abruptly for reason S, then the `try` statement completes abruptly for reason S (and reason R is discarded).

– TYPE THEORY

$$
\begin{array}{l}
s, f \colon \mathsf{OM} \to \mathsf{StatResult},\\
\ell \colon \mathsf{list}[[\,\mathsf{exc} \colon \mathsf{ClassName}, \mathsf{handler} \colon \mathsf{RefType} \to \mathsf{OM} \to \mathsf{StatResult}\,]] \;\vdash\\
\quad \mathsf{TRY\text{-}CATCH\text{-}FINALLY} \cdot s \cdot \ell \cdot f \;:\; \mathsf{OM} \to \mathsf{StatResult} \;\stackrel{\mathrm{def}}{=}\\
\quad\quad \lambda x \colon \mathsf{OM}.\ \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{\\
\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang}\\
\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto f \cdot y\\
\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto\\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{CASES}\ a\ \mathsf{OF}\ \{\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{excp}\, e \mapsto\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{CASES}\ e.\mathsf{ex}\ \mathsf{OF}\ \{\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{null} \mapsto \mathsf{hang} \quad \text{// don't care, see (2)}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{ref}\, r \mapsto \mathsf{TRY\text{-}LOOP\text{-}FINALLY} \cdot r \cdot \ell \cdot f \cdot (e.\mathsf{es})\,\}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{rtrn}\, z \mapsto\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{CASES}\ f \cdot z\ \mathsf{OF}\ \{\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y' \mapsto \mathsf{abnorm}(\mathsf{rtrn}\, y')\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a' \mapsto \mathsf{abnorm}\, a'\,\}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{break}\, b \mapsto\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{CASES}\ f \cdot (b.\mathsf{bs})\ \mathsf{OF}\ \{\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y' \mapsto \mathsf{abnorm}(\mathsf{break}(\mathsf{bs} = y', \mathsf{blab} = b.\mathsf{blab}))\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a' \mapsto \mathsf{abnorm}\, a'\,\}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{cont}\, c \mapsto\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{CASES}\ f \cdot (c.\mathsf{cs})\ \mathsf{OF}\ \{\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang}\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{norm}\, y' \mapsto \mathsf{abnorm}(\mathsf{cont}(\mathsf{cs} = y', \mathsf{clab} = c.\mathsf{clab}))\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid \mathsf{abnorm}\, a' \mapsto \mathsf{abnorm}\, a'\,\}\,\}\,\}
\end{array}
$$

Fig. 1. Formalisation of Java's `try-catch-finally`

As before, this is formalised in two steps, see Figures 1 and 2. The main difference with the TRY-CATCH function is in the occurrence of the additional "finally" statement f, which is executed after each possible outcome of the "try" statement s, and the catch statements. The most subtle point is that in case the statement s terminates abruptly because of a `return`, `break` or `continue`, and the finally clause f terminates normally, the side-effect of f is passed on in the eventual result (via the state y'). This is not so explicitly stated in (the above quote from) [7, §§14.19.2], but made explicit in our type-theoretic formalisation. It will be illustrated in an example in the next section.

The function TRY-LOOP-FINALLY in Figure 2 handles the actual catching much like before, except that the "finally" statement needs to be executed after every possibility. This involves appropriate handling of side-effects, like for TRY-CATCH-FINALLY above. The following results are then as expected.

Lemma 1. *Let* $\mathsf{skip}\colon \mathsf{OM} \to \mathsf{StatResult}$ *be the function* $\lambda x\colon \mathsf{OM}.\,\mathsf{norm}\,x$ *which directly terminates normally. For all locations* $p\colon \mathsf{MemLoc}$*, statements* $s\colon \mathsf{OM} \to \mathsf{StatResult}$ *and lists* $\ell\colon \mathsf{list}[[\,\mathsf{exc}\colon \mathsf{ClassName}, \mathsf{handler}\colon \mathsf{RefType} \to \mathsf{OM} \to \mathsf{StatResult}\,]]$*,*

1. $\mathsf{TRY\text{-}LOOP\text{-}FINALLY} \cdot p \cdot \ell \cdot \mathsf{skip} = \mathsf{TRY\text{-}LOOP} \cdot p \cdot \ell$
2. $\mathsf{TRY\text{-}CATCH\text{-}FINALLY} \cdot p \cdot \ell \cdot \mathsf{skip} = \mathsf{TRY\text{-}CATCH} \cdot p \cdot \ell.$

Proof. The first statement follows by induction on ℓ. The second one by unpacking the definitions, distinguishing many cases, and using 1. □

6 Examples

In order to illustrate the rôle of our formalisation of Java's exception mechanism we shall discuss two examples. These are two artificial Java programs, concentrating on exception handling. The relevant properties of these programs are stated as annotations, written in the behavioural specification language JML [16]. We shall not describe this language in detail, and hope that the annotations are largely self-explanatory. The two examples have been translated into PVS [23], using the LOOP tool. The JML annotations become predicates, on class implementations. The actual Java code is translated into a specific implementation. Thus it becomes possible to prove in PVS that the given implementation satisfies the JML specification. This has been done for both the examples. The proofs proceed almost entirely by automatic rewriting—unfolding in particular the type-theoretic functions for exception handling from the previous section—and do not require real user interaction. Hence there is not much to say about these proofs. But we hope that the reader appreciates the organisational and semantical complications that are involved.

6.1 Side effects in `finally` clauses

In the previous section the formalisations TRY-CATCH-FINALLY and TRY-LOOP-FINALLY showed the handling of side effects of the finally clause f (via the state y'). Here we shall see that these effects indeed take place in Java. For this purpose we use the following Java program.

— JAVA

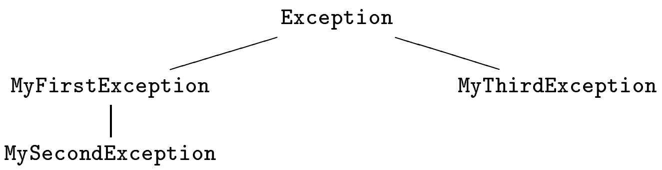
```
class SideEffectFinally {
    int i, j;
    int aux_test() { try { return i; }
                     finally { i += 10; j += 100; } }
    /*@ normal_behavior
      @   requires: true;
      @ modifiable: i, j;
      @    ensures: \result == \old(i) + \old(j) + 100
      @             && i == \old(i) + 10 && j == \old(j) + 100;
      @*/
    int test() { return aux_test() + j; }
}

class SideEffectFinallyPrint {
    public static void main (String[] args) {
        SideEffectFinally a = new SideEffectFinally();
        System.out.println(a.test());  }
}
```

This example contains two classes, namely `SideEffectFinally` and `SideEffect-FinallyPrint`. The latter is only used for printing one specific result, namely the outcome of the `test` method after both `i` and `j` from `SideEffectFinally` have been initialised to the default value `0`. The `main` method will then print `100`. There are actually two subtle points here. First, of course that the `finally` clause does have an effect after the `return` statement (which leads to abrupt termination). Secondly, the result of the `aux_test` method only shows the effect on `j` because the value of `i` has already been bound to the result of the method before the `finally` clause, so that the increment statement `i += 10` does not have an influence on the outcome.

The JML specification for the `test` method involves a higher degree of generality, because it is not restricted to the case where both `i` and `j` are `0`. It states that the `test` method always terminates normally and that its result equals the sum of the values of `i` and `j` *before* the method call, plus `100`. It also states that this method may modify both `i` and `j`—which it actually does, but the modification of `i` is not shown via the result of the method. As said, this specification holds for the method implementation. The proof in PVS relies on the `try-catch-finally` formalisation from Subsection 5.2.

6.2 Exception selection

The second example concentrates on the selection of the appropriate `catch` clause, for a thrown exception. It requires several auxiliary exception classes, with suitable inheritance relations between them, namely:



JAVA

```
class MyFirstException extends Exception {
    public MyFirstException(String s) { super(s); }
}
class MySecondException extends MyFirstException {
    public MySecondException(String s) { super(s); }
}
class MyThirdException extends Exception {
    public MyThirdException(String s) { super(s); }
}
class MyExceptions {
    int i;
    void throwSecond() throws Exception {
        throw new MySecondException("oops"); }
    /*@ exceptional_behavior
      @   requires: true;
      @ modifiable: i;
      @    signals: (MyFirstException e) i == \old(i) + 1010 &&
      @                           e.getMessage().equals("oops");
      @*/
    void test() throws Exception {
        String s = "";
        try { throwSecond(); }
        catch (MyThirdException e) { i += 1; }
        catch (MyFirstException e) {
            i += 10;
            s = e.getMessage();
            throw new MyThirdException("bla"); }
        catch (Exception e) { i += 100; }
        finally { i += 1000; throw new MyFirstException(s); } }
}
```

The exception that is thrown by the method `throwSecond` is handled by the second `catch`, because `MySecondException` is a subclass of `MyFirstException`. Subsequently, the third `catch` clause is not executed, but, of course, the `finally` clause is. Thus `i` is incremented by `10 + 1000 = 1010`. The exception thrown in the `finally` clause is the one that eventually appears.

The JML specification of the method `test` tells that this method will terminate abruptly because of a `MyFirstException`. Further, that in the resulting "abnormal" state the value of `i` is `1010` more than in the original state (before the method call), and the message of the exception is "oops". The verification in PVS proceeds entirely automatic, and involves almost 5000 small rewrite steps.

7 Conclusion

Java's exception handling mechanism can be a powerful technique for increasing the reliability and robustness of programs written in Java. Proper use of it requires proper understanding of its behaviour. The type-theoretic semantics presented in this paper helps to clarify the different termination possibilities that may occur, by describing them via coalgebras in a precise formal language. It also allows us to precisely formalise the throw and catch behaviour, following the informal language specification. This semantics forms the basis for Java program verification with an appropriate Hoare logic, using proof tools. This has been illustrated with two Java programs involving non-trivial exception handling, whose specifications in JML were verified in PVS.

Acknowledgements

Thanks to Gilad Bracha, Tobias Nipkow and David von Oheimb for discussing (and confirming) the exception invariant (2). Joachim van den Berg, Marieke Huisman, Hans Meijer and Erik Poll provided useful feedback on a first draft.

References

1. J. Alves-Foss and F. S. Lam. Dynamic denotational semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lect. Notes Comp. Sci., pages 201–240. Springer, Berlin, 1998.
2. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Lect. Notes Comp. Sci., pages 1–21. Springer, Berlin, 2000.
3. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. Techn. Rep. CSI-R0019, Comput. Sci. Inst., Univ. of Nijmegen. To appear at TACAS'01., 2000.
4. T. Budd. *Understanding Object-Oriented Programming with Java.* Addison-Wesley, 2000. Updated Edition.
5. F. Christian. Correct and robust programs. *IEEE Trans. on Software Eng.*, 10(2):163–174, 1984.
6. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, 2000.
8. M. Huisman. *Reasoning about JAVA Programs in higher order logic with PVS and Isabelle.* PhD thesis, Univ. Nijmegen, 2001.

9. M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lect. Notes Comp. Sci., pages 301–319. Springer, Berlin, 2000.
10. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 284–303. Springer, Berlin, 2000.
11. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. To appear in *Software Tools for Technology Transfer*, 2001.
12. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. Techn. Rep. CSI-R0018, Comput. Sci. Inst., Univ. of Nijmegen. To appear at FASE'01., 2000.
13. B. Jacobs and E. Poll. A monad for basic Java semantics. In T. Rus, editor, *Algebraic Methodology and Software Technology*, number 1816 in Lect. Notes Comp. Sci., pages 150–164. Springer, Berlin, 2000.
14. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
15. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.
16. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.
17. K.R.M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Inst. of Techn., 1995.
18. K.R.M. Leino and J.L.A. van de Snepscheut. Semantics of exceptions. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 447–466. North-Holland, 1994.
19. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification Second Edition*. The Java Series. Addison-Wesley, 1999.
20. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2^{nd} rev. edition, 1997.
21. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Techn. Univ. München, 2000.
22. D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lect. Notes Comp. Sci., pages 119–156. Springer, Berlin, 1998.
23. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
24. L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and computer science*, pages 361–386. Academic Press, London, 1990. The APIC series, vol. 31.
25. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application*, pages 135–154. Kluwer Acad. Publ., 2000.
26. E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Comp. Networks Mag.*, 2001. To appear.
27. Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.

— TYPE THEORY

```
p: MemLoc, f: OM → StatResult
ℓ: list[[ exc: ClassName, handler: RefType → OM → StatResult ]]  ⊢

  TRY-LOOP-FINALLY · p · ℓ · f  :  OM → StatResult  ≝
    λx: OM. CASES ℓ OF {
              | nil ↦
                  CASES f · x OF {
                    | hang ↦ hang
                    | norm y ↦ abnorm(excp(es = y, ex = ref p))
                    | abnorm a ↦ abnorm a }
              | cons(h, t) ↦
                  IF SubClass? · (gettype · p · x) · (h.exc)
                  THEN
                    CASES (h.handler) · (ref p) · x OF {
                      | hang ↦ hang
                      | norm y ↦ f · y
                      | abnorm a ↦
                          CASES a OF {
                            | excp e ↦
                                CASES f · (e.es) OF {
                                  | hang ↦ hang
                                  | norm y′ ↦ abnorm(excp( es = y′,
                                                           ex = e.ex))
                                  | abnorm a′ ↦ abnorm a′ }
                            | rtrn z ↦
                                CASES f · z OF {
                                  | hang ↦ hang
                                  | norm y′ ↦ abnorm(rtrn y′)
                                  | abnorm a′ ↦ abnorm a′ }
                            | break b ↦
                                CASES f · (b.bs) OF {
                                  | hang ↦ hang
                                  | norm y′ ↦ abnorm(break( bs = y′,
                                                            blab = b.blab))
                                  | abnorm a′ ↦ abnorm a′ }
                            | cont c ↦
                                CASES f · (c.cs) OF {
                                  | hang ↦ hang
                                  | norm y′ ↦ abnorm(cont( cs = y′,
                                                           clab = c.clab))
                                  | abnorm a′ ↦ abnorm a′ } } }
                  ELSE TRY-LOOP-FINALLY · p · t · f · x
                  ENDIF }
```

Fig. 2. Formalisation of the auxiliary function TRY-LOOP-FINALLY used in Figure 1