Java Program Verification via a Hoare Logic with Abrupt Termination *

MARIEKE HUISMAN, BART JACOBS

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{marieke,bart}@cs.kun.nl

Abstract. This paper formalises a semantics for statements and expressions (in sequential imperative languages) which includes non-termination, normal termination and abrupt termination (*e.g.* because of an exception, break, return or continue). This extends the traditional semantics underlying *e.g.* Hoare logic, which only distinguishes termination and non-termination. An extension of Hoare logic is elaborated that includes means for reasoning about abrupt termination (and side-effects). It prominently involves rules for reasoning about while loops, which may contain exceptions, breaks, continues and returns. This extension applies in particular to Java. As an example, a standard pattern search algorithm in Java (involving a while loop with returns) is proven correct using the proof-tool PVS.

1 Introduction

Java is quickly becoming one of the most widely used programming languages. Being able to establish the correctness of Java programs is thus of evident importance. In [19] a tool is presented which translates Java classes into logical theories (of the proof tools Isabelle [26] or PVS [23, 24]). The translation involves a particular semantics for statements and expressions, which forms a basis for proving correctness formulas. But "[...] reasoning about correctness formulas in terms of semantics is not very convenient. A much more promising approach is to reason directly on the level of correctness formulas." (quote from [3, p. 57]). Hoare logic is a formalism for doing precisely this.

The first contribution of this paper is a precise description of the semantics of statements and expressions underlying [19]. It involves abrupt termination as a prominent feature. The present description is more detailed than the one in [19] and more abstract in two aspects. First, and most importantly, it is not formulated in the language of PVS or Isabelle, but in a general type theoretical language involving records and variants. This means that the reader need not be familiar with particulars of (the language of) PVS or Isabelle. Secondly, the semantics described here is not especially focused on Java, and may apply to other languages with similar forms of abrupt termination.

* From: T. Maibaum (ed), Fundamental Approaches to Software Engineering (FASE'00), Springer LNCS 1783, p.284-303, 2000.

The second contribution consists of a concrete and detailed elaboration and adaptation of existing approaches to programming logics with exceptions, notably from [9, 22, 21] (which are mostly in weakest precondition form). This elaboration and adaptation will be done for a real-world programming language like Java. Although the basic ideas used here are the same as in [9, 22, 21], the elaboration is different. For example, we have many forms of abrupt termination, and not just one sole exception, and we have a semantics of statements and expressions as particular functions (actually coalgebras, or maps in a Kleisli category, see [18]), and not a semantics of traces.

The logic presented here did not arise as a purely theoretical exercise, but was developed during actual verification of Java programs. For example, the ability to handle abnormalities was crucially needed for the vector class verification from [17], especially when dealing with `for` loops with `return` statements in their bodies.

Regarding the semantics that we shall be using, we recall that in classical program semantics and Hoare logic the assumption is that statements will either terminate normally, resulting in a successor state, or will not terminate at all, see *e.g.* [6, Chapter 3] or [28, Section 2.2]. In the latter case one also says that the statement hangs, typically because of a non-terminating loop. Hence, statements may be understood as partial functions from states to states. Writing Self for the state space, we can see statements as "state transformer" functions

$$\mathsf{Self} \longrightarrow \mathsf{lift}[\mathsf{Self}] \quad (= 1 + \mathsf{Self})$$

where 1 is a one-element set and + is disjoint union. This classical view of statements turns out to be inadequate for reasoning about Java programs. Java statements may hang, or terminate normally (like above), but they may additionally "terminate abruptly" (see *e.g.* [13, 4]). Abrupt termination may be caused by an exception (typically a division by 0), a return, a break or a continue (inside a loop). Abrupt (or abnormal) termination is fundamentally different from non-termination: abnormalities may be temporary because they may be caught at some later stage, whereas recovery from non-termination is impossible.

Abrupt termination requires a modification of the standard semantics of statements and expressions, resulting in a failure semantics, as for example in [28, Section 5.1]. Here, statements will be modeled as more general state transformer functions

$$\mathsf{Self} \xrightarrow{\mathsf{stat}} 1 + \mathsf{Self} + \mathsf{StatAbn}$$

where $\mathsf{StatAbn}$ forms a new option, which itself can be subdivided into four parts:

$$\mathsf{StatAbn} = \mathsf{Exception} + \mathsf{Return} + \mathsf{Break} + \mathsf{Continue}$$

These four constituents of $\mathsf{StatAbn}$ will typically consist of a state in Self together with some extra information (*e.g.* the kind of exception, or the label of a break). This structure of the codomain of our Java state transformer functions will be captured formally in a type $\mathsf{StatResult}$, see Section 4.

In classical Hoare logic, expressions are viewed as functions

$$\mathsf{Self} \longrightarrow \mathsf{Out}$$

where Out is the type of the result. Also this view is not quite adequate for our purposes, because it does not involve non-termination, abrupt termination or side-effects. In contrast, an expression in Java may hang, terminate normally or terminate abruptly. If it terminates normally it produces an output result (of the type of the expression) together with a state (since it may have a side-effect). If it terminates abruptly, this can only be because of an exception (and not because of a break, continue or return). Hence an expression of type Out will be (in our view) a function of the form:

$$\mathsf{Self} \xrightarrow{\mathsf{expr}} 1 + (\mathsf{Self} \times \mathsf{Out}) + \mathsf{ExprAbn}$$

The first option 1 captures the situation where an expression hangs. The second option $\mathsf{Self} \times \mathsf{Out}$ occurs when an expression terminates normally, resulting in a successor state together with an output result. The final option ExprAbn describes abrupt termination—because of an exception—for expressions. Again, this will be captured by a suitable type ExprResult in Section 4.

This abstract representation of statements and expressions as "one entry / multiexit" functions (terminology of [9]) forms the basis for the current work. It will be used to give meaning to basic programming constructs like composition, if-then-else, and while.

Hoare logic for a particular programming language consists of a series of deduction rules for special sentences, involving constructs from the programming language, like assignment, if-then-else and composition, see Section 3. In particular, while loops have received much attention in Hoare logic, because they involve a judicious and often non-trivial choice of a loop invariant. For more information, see *e.g.* [6, 14, 2, 11, 3]. There is what we would like to call a "classical" body of Hoare logic, which applies to standard constructs from an idealised imperative programming language. This forms a well-developed part of the theory of Hoare logic. It is couched in general terms, and not aimed at a particular programming language. This generality is an advantage, but also a disadvantage, in particular when one wishes to reason about a specific programming language. In this paper, an extension of standard Hoare logic is presented in which the different output options of statements and expressions will result in different kinds of sentences (for e.g. Break or Return), see Section 5 below.

We should emphasise that the extension of Hoare logic that is introduced here applies to only a small (sequential, non-object-oriented) part of Java. Hoare logics for reasoning about concurrent programs may be found in [3], and for reasoning about object-oriented programs in [8, 1]. There is also more remotely related work on "Hoare logic with jumps", see [10, 5] (or also Chapter 10 by De Bruin in [6]), but in those logics it is not always possible to reason about intermediate, "abnormal" states. And in [27] a programming logic for Java is described, which, in its current state, does not cover forms of abrupt termination—the focus point of this work.

This paper is organised as follows. Section 2 sketches the type theory and logic in which we shall be working. Section 3 briefly discusses the basics of Hoare logic. Section 4 discusses the formalisation of the semantics of Java statements and expressions in type theory. It also describes Hoare logic of normal termination. Section 5 discusses our extension of Hoare logic of abrupt termination. Proof rules for abruptly terminating while loops are discussed in Section 6. Section 7 gives an example of the use of Hoare logic of abrupt termination. Finally, we end with conclusions and future work in Section 8.

2 Basic type theory and logic

In this section we shall present the simple type theory and (classical) higher-order logic in which we will be working. It can be seen as a common abstraction from the type theories and logics of both PVS and Isabelle/HOL[1]. Using this general type theory and logic means that we can stay away from the particulars of the languages of PVS and Isabelle and make this work more accessible to readers unfamiliar with these languages. Due to space restrictions, the explanation will have to be rather sketchy.

Our type theory is a simple type theory with types built up from:[2] type variables $\alpha, \beta, \ldots$, type constants nat, bool, string (and some more), exponent types $\sigma \to \tau$, labeled product (or record) types $[\, \mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n \,]$ and labeled coproduct (or variant) types $\{\, \mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n \,\}$, for given types $\sigma, \tau, \sigma_1, \ldots, \sigma_n$. New types can be introduced via definitions, as in:

$$\mathsf{lift}[\alpha] : \mathsf{TYPE} \stackrel{\text{def}}{=} \{\, \mathsf{bot} \colon \mathsf{unit} \mid \mathsf{up} \colon \alpha \,\}$$

where unit is the empty product type $[\,]$. This lift type constructor adds a bottom element to an arbitrary type, given as type variable α. It is frequently used.

For exponent types we shall use the standard lambda abstraction $\lambda x \colon \sigma.\, M$ and application NL notation. For terms $M_i \colon \sigma_i$, we have a labeled tuple $(\, \mathsf{lab}_1 = M_1, \ldots, \mathsf{lab}_n = M_n \,)$ inhabiting the labeled product type $[\, \mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n \,]$. For a term $N \colon [\, \mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n \,]$ in this product, we write $N.\mathsf{lab}_i$ for the selection term of type σ_i. Similarly, for a term $M \colon \sigma_i$ there is a labeled or tagged term $\mathsf{lab}_i\, M$ in the labeled coproduct type $\{\, \mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n \,\}$. And for a term $N \colon \{\, \mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n \,\}$ in this coproduct type, together with n terms $L_i(x_i) \colon \tau$ containing a free variable $x_i \colon \sigma_i$ there is a case term $\mathsf{CASES}\ N\ \mathsf{OF}\ \{\, \mathsf{lab}_1\, x_1 \mapsto L_1(x_1) \mid \ldots \mid \mathsf{lab}_n\, x_n \mapsto L_n(x_n) \,\}$ of type τ. These introduction and elimination terms for labeled products and coproducts are required to satisfy standard (β)- and (η)-conversions.

[1] Certain aspects of PVS and Isabelle/HOL are incompatible, like the type parameters in PVS versus type polymorphism in Isabelle/HOL, so that the type theory and logic that we use is not really in the intersection. But with some good will it should be clear how to translate the constructions that we present into the particular languages of these proof tools. See [15] for a detailed comparison.

[2] In this paper we only use non-recursive types, but in the translation of Java constructs like `catch` and `switch` we also use the (recursive) list type constructor.

Formulas in higher-order logic are terms of type bool. We shall use the connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\neg$ (negation, used with rules of classical logic) and constants true and false, together with the (typed) quantifiers $\forall x\colon \sigma.\, \varphi$ and $\exists x\colon \sigma.\, \varphi$, for a formula φ. There is a conditional term $\mathsf{IF}\ \varphi\ \mathsf{THEN}\ M\ \mathsf{ELSE}\ N$, for terms M, N of the same type, and a choice operator $\varepsilon x\colon \sigma.\, \varphi(x)$, yielding a term of type σ. We shall use inductive definitions (over the type nat of natural numbers), and also reason with the standard induction principle. All this is present in both PVS and Isabelle/HOL.

3 Basics of Hoare logic

Traditionally, Hoare logic allows one to reason about simple imperative programs, containing assignments, conditional statements, while and for loops, and block statements with local variables. It provides proof rules to derive the correctness of a complete program from the correctness of parts of the program. Sentences (also called asserted programs) in this logic have the form $\{P\}\,S\,\{Q\}$, for partial correctness, or $[P]\,S\,[Q]$, for total correctness. They involve assertions P and Q in some logic (usually predicate logic), and statements S from the programming language that one wishes to reason about. The partial correctness sentence $\{P\}\,S\,\{Q\}$ expresses that if the assertion P holds in some state x and *if* the statement S, when evaluated in state x, terminates normally, resulting in a state x', then the assertion Q holds in x'. Total correctness $[P]\,S\,[Q]$ expresses something stronger, namely: if P holds in x, *then* S in x terminates normally, resulting in a state x' where Q holds. Some well-known proof rules are:

$$\frac{\{P\}\,S\,\{Q\} \qquad \{Q\}\,T\,\{R\}}{\{P\}\,S;T\,\{R\}}\,[\mathtt{comp}] \qquad \frac{\{P \wedge C\}\,S\,\{Q\} \qquad \{P \wedge \neg C\}\,T\,\{Q\}}{\{P\}\,\mathtt{if}\,C\,\mathtt{then}\,S\,\mathtt{else}\,T\,\{Q\}}\,[\mathtt{if}]$$

$$\frac{\{P \wedge C\}\,S\,\{P\}}{\{P\}\,\mathtt{while}\,C\,\mathtt{do}\,S\,\{P \wedge \neg C\}}\,[\mathtt{while}]$$

The predicate P in the `while` rule is often called the loop invariant.

Most classical partial correctness proof rules immediately carry over to total correctness. A well-known exception is the rule for the while statement, which needs an extra condition to prove termination. Consider for example the program (fragment) $\mathtt{while}\,\mathtt{true}\,\mathtt{do}\,\mathtt{skip}$. For every predicate P, it is easy to prove $[P]\,\mathtt{skip}\,[P]$. But the whole statement never terminates, so we should not be able to conclude $[P]\,\mathtt{while}\,\mathtt{true}\,\mathtt{do}\,\mathtt{skip}\,[P \wedge \mathtt{false}]$. An extra condition, which guarantees termination, should be added to the rule. The standard approach is to define a mapping from the underlying state space to some well-founded set and to require that every time the body is executed, the result of this mapping decreases. As this can happen only finitely often, the loop has to terminate. Often this mapping is called the variant (in contrast to the loop invariant). This gives the following proof rule for total correctness of while statements.

$$\frac{[P \wedge C \wedge \text{variant} = n]\,S\,[P \wedge \text{variant} < n]}{[P]\,\mathtt{while}\,C\,\mathtt{do}\,S\,[P \wedge \neg C]}$$

3.1 Some limitations of Hoare logic

Hoare logic has had much influence on the way of thinking about (imperative) programming, but unfortunately it also has some shortcomings. First of all, it is not really feasible to verify non-trivial programs by hand. Most computer science students—at some stage during their training—have to verify some well-known algorithm, such as quicksort. At that moment they often decide never to do this again. One would like to have a tool, which does most of the proving automatically, so that the user only has to interfere at crucial steps in the proof. Secondly, classical Hoare logic enables reasoning about an ideal programming language, without side-effects, exceptions, abrupt termination of statements, *etc.* However, most widely-used (imperative) programming languages do have side-effects, exceptions and the like.

In our project [16, 19] we aim at reasoning about real, widely-used programming languages. Thus far we concentrated on Java. The first step of our project is to provide a formal semantics to Java statements and expressions, in the higher-order logic and type theory from the previous section. Reasoning about a particular Java program can only be done after it is translated into type theory, for which we use our translation tool. In the logic, the user can write down required properties about the program—using partial and total correctness sentences—and try to prove these. Via appropriate rewrite rules—used as "interpreter"—many properties for non-looping, non-recursive programs can be proven without user interaction. This translation (to PVS) and reasoning are described in more detail elsewhere [19], while this paper presents more details of the formal semantics of Java. The underlying memory model is explained in [7].

As mentioned in the introduction, reasoning that is directly based on the semantics of the programming language is often not appropriate for looping or recursive programs. These kinds of programs require the use of a special purpose logic, such as Hoare logic. Gordon [12] describes how the rules of Hoare logic are mechanically derived from the semantics of a simple imperative language. This enables both semantic and axiomatic reasoning about programs in this language. What we describe next may be seen as a deeper elaboration of this approach, building on ideas from [9, 22, 21].

4 Semantics of Java statements and expressions

This section describes in more detail how the semantics of Java statements and expressions is formalised in type theory. Statements and expressions are regarded as state transformer functions, possibly producing a new state with a tag telling whether the state is normal or abnormal. This will be explained first. The last part of this section (Subsection 4.4) describes our first extension of classical Hoare logic, namely Hoare logic of normal termination for Java-like languages, which incorporates side-effects. Section 5 describes the more substantial extension of Hoare logic with abrupt termination.

4.1 Statements and expressions as state transformers

In this section we will describe state transformers for statements and expressions in type theory. As explained in the introduction, an extra possibility has to be added (besides non-termination and normal termination) to capture abrupt termination of statements and expressions: statements and expressions are modeled as functions with types $\mathsf{Self} \to 1 + \mathsf{Self} + \mathsf{StatAbn}$ and $\mathsf{Self} \to 1 + (\mathsf{Self} \times \mathsf{Out}) + \mathsf{ExprAbn}$, respectively. For convenience, the output types are represented in two steps, via the variant types $\mathsf{PreStatResult}$ and $\mathsf{PreExprResult}$.

$$\begin{array}{ll} \mathsf{PreStatResult}[\mathsf{Self}, A] \,:\, \mathsf{TYPE} \stackrel{\text{def}}{=} & \mathsf{PreExprResult}[\mathsf{Self}, \mathsf{Out}, A] \,:\, \mathsf{TYPE} \stackrel{\text{def}}{=} \\ \quad \{\, \mathsf{hang} \colon \mathsf{unit}, & \quad \{\, \mathsf{hang} \colon \mathsf{unit}, \\ \quad \mid \mathsf{norm} \colon \mathsf{Self}, & \quad \mid \mathsf{norm} \colon [\, \mathsf{ns} \colon \mathsf{Self}, \mathsf{res} \colon \mathsf{Out}\,], \\ \quad \mid \mathsf{abnorm} \colon A \,\} & \quad \mid \mathsf{abnorm} \colon A \,\} \end{array}$$

These definitions involve type variables $\mathsf{Self}, \mathsf{Out}$ and A. Further, we have two different types for abnormalities. Expressions only terminate abruptly, because of an exception, while statements can also terminate abruptly because of a `break`, `continue` or `return` [13]. Below, in Section 4.2, the meaning of these statements will be described in more detail.

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

An (expression or statement) exception abnormality consists of a state together with a reference to an exception object. The reference is represented as an element of a special type $\mathsf{RefType}$ (see [7]), which does not play a rôle in the sequel. A return abnormality only consists of a (tagged) state, and break and continue abnormalities consist of a state, possibly with a label (given as string).

Finally, we define abbreviations $\mathsf{StatResult}$ and $\mathsf{ExprResult}$ by substitution as:

$$\begin{aligned} \mathsf{StatResult}[\mathsf{Self}] &\stackrel{\text{def}}{=} \mathsf{PreStatResult}[\mathsf{Self}, \mathsf{StatAbn}[\mathsf{Self}]] \\ \mathsf{ExprResult}[\mathsf{Self}, \mathsf{Out}] &\stackrel{\text{def}}{=} \mathsf{PreExprResult}[\mathsf{Self}, \mathsf{Out}, \mathsf{ExprAbn}[\mathsf{Self}]] \end{aligned}$$

To summarise, in our formalisation, statements are modeled as functions from Self to $\mathsf{StatResult}[\mathsf{Self}]$, and expressions as functions from Self to $\mathsf{ExprResult}[\mathsf{Self}, \mathsf{Out}]$, for the appropriate result type Out.

There is one technicality that deserves some attention. Sometimes an expression has to be transformed into a statement, which is only a matter of forgetting the result of the expression. However, in our formalisation we have to do this transformation explicitly, using a function $\mathsf{E2S}$.

$$
\begin{array}{l}
e\colon \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{Out}] \;\vdash \\
\quad \mathsf{E2S}(e) \;:\; \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \;\stackrel{\mathrm{def}}{=} \\
\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{CASES}\ e\,x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}(y.\mathsf{ns}) \\
\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}(\mathsf{excp}(\mathsf{es} = a.\mathsf{es}, \mathsf{ex} = a.\mathsf{ex}))\ \}
\end{array}
$$

In the last line an expression abnormality (an exception) is transformed into a statement abnormality.

4.2 Throwing and catching abnormalities

Based on the types representing statements and expressions various program constructs can be formalised. This will be done here, and in the next subsection. We start with statements dealing with abrupt termination. We shall use the notation $[\![S]\!]$ to denote the interpretation (translation) of the Java statement or expression S in type theory.

Abnormalities can both be thrown and be caught, basically via re-arranging coproduct options. We shall describe constructs for both throwing and catching in type theory. Abrupt termination affects the flow of control: once it arises, all subsequent statements are ignored, until the abnormality is caught, see the definition of composition ";" in the next subsection. From that moment on, the program executes normally again. We shall discuss breaks and returns in some detail, and only sketch continues and exceptions.

Break A `break` statement can be used to exit from any block. If a `break` statement is labeled, it exits the block with the same label. Typically, a `break` statement with label `lab` must occur inside a (nested) block with the same label `lab`, so that it can not be used as an arbitrary goto. Unlabeled `break` statements exit the innermost `switch`, `for`, `while` or `do` statement. A Java `break` statement is translated as

$$
\begin{aligned}
[\![\texttt{break}]\!] &\stackrel{\mathrm{def}}{=} \mathsf{BREAK} \\
[\![\texttt{break label}]\!] &\stackrel{\mathrm{def}}{=} \mathsf{BREAK\text{-}LABEL}(\text{“}\texttt{label}\text{”})
\end{aligned}
$$

where BREAK and $\mathsf{BREAK\text{-}LABEL}(s)$, for $s\colon \mathsf{string}$, are defined as functions $\mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}]$:

$$
\begin{aligned}
\mathsf{BREAK} &\stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\, \mathsf{abnorm}(\mathsf{break}(\mathsf{bs} = x, \mathsf{blab} = \mathsf{bot})) \\
\mathsf{BREAK\text{-}LABEL}(s) &\stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\, \mathsf{abnorm}(\mathsf{break}(\mathsf{bs} = x, \mathsf{blab} = \mathsf{up}\, s))
\end{aligned}
$$

There is an associated function $\mathsf{CATCH\text{-}BREAK}$ which turns abnormal states, because of breaks with the appropriate label, back into normal states.

$$
\begin{array}{l}
ll\colon \mathsf{lift}[\mathsf{string}], s\colon \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \ \vdash \\
\quad \mathsf{CATCH\text{-}BREAK}(ll)(s) \ : \ \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \ \stackrel{\mathrm{def}}{=} \\
\quad\quad \lambda x\colon \mathsf{Self}.\ \mathsf{CASES}\ s\,x\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\quad\quad\quad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}\, y \\
\quad\quad\quad\quad \mid \mathsf{abnorm}\, a \mapsto \ \mathsf{CASES}\ a\ \mathsf{OF}\ \{ \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{excp}\, e \mapsto \mathsf{abnorm}(\mathsf{excp}\, e) \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{rtrn}\, z \mapsto \mathsf{abnorm}(\mathsf{rtrn}\, z) \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{break}\, b \mapsto \ \mathsf{IF}\ b.\mathsf{blab} = ll \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{THEN}\ \mathsf{norm}(b.\mathsf{bs}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{ELSE}\ \mathsf{abnorm}(\mathsf{break}\, b) \\
\quad\quad\quad\quad\quad\quad \mid \mathsf{cont}\, c \mapsto \mathsf{abnorm}(\mathsf{cont}\, c)\ \}\ \}
\end{array}
$$

In the Java translation [19] every labeled block is enclosed with $\mathsf{CATCH\text{-}BREAK}$ applied to the appropriate label:

$$
[\![\texttt{label:body}]\!] \stackrel{\mathrm{def}}{=} \mathsf{CATCH\text{-}BREAK}(\mathsf{up}(\text{“}\texttt{label}\text{”}))([\![\texttt{body}]\!])
$$

Similarly, every `switch`, `while`, `for` and `do` statement is enclosed with $\mathsf{CATCH\text{-}BREAK}$ applied to bot.

Return When a `return` statement is executed, the program immediately exits from the current method. A `return` statement may have an expression argument; if so, this expression is evaluated and returned as the result of the method. The translation of the Java `return` statement (without argument) is $[\![\texttt{return}]\!] = \mathsf{RETURN}$ where RETURN is defined in type theory as:

$$
\mathsf{RETURN} \ : \ \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}] \ \stackrel{\mathrm{def}}{=} \ \lambda x\colon \mathsf{Self}.\, \mathsf{abnorm}(\mathsf{rtrn}\, x)
$$

This statement produces an abnormal state. Such a return abnormality can be undone, via appropriate catch-return functions. In our translation of Java programs, such a function $\mathsf{CATCH\text{-}STAT\text{-}RETURN}$ is wrapped around every method body that returns `void`. First the method body is executed. This may result in an abnormal state, because of a return. In that case the function $\mathsf{CATCH\text{-}STAT\text{-}RETURN}$ turns the state back to normal again. Otherwise, it leaves the state unchanged.

$$
\begin{array}{l}
s\colon \mathsf{Self} \to \mathsf{StatResult[Self]} \;\vdash \\
\quad \mathsf{CATCH\text{-}STAT\text{-}RETURN}(s) \;:\; \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\mathrm{def}}{=} \\
\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{CASES}\ s\,x\ \mathsf{OF}\ \{ \\
\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{norm}\, y \\
\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto\ \mathsf{CASES}\ a\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad \mid \mathsf{excp}\, e \mapsto \mathsf{abnorm}(\mathsf{excp}\, e) \\
\qquad\qquad\qquad \mid \mathsf{rtrn}\, z \mapsto \mathsf{norm}\, z \\
\qquad\qquad\qquad \mid \mathsf{break}\, b \mapsto \mathsf{abnorm}(\mathsf{break}\, b) \\
\qquad\qquad\qquad \mid \mathsf{cont}\, c \mapsto \mathsf{abnorm}(\mathsf{cont}\, c)\ \}\ \}
\end{array}
$$

The translation of a `return` statement with argument is similar, but more subtle. First the value of the expression is stored in a special local variable, and then the state becomes abnormal, via the above RETURN. Instead of CATCH-STAT-RETURN a function CATCH-EXPR-RETURN is used, which eventually turns the state back to normal and, in that case, returns the output that is held by the special variable.

Continue Within loop statements (`while`, `do` and `for`) a `continue` statement can occur. The effect is that control skips the rest of the loop's body and starts re-evaluating the (update statement, in a `for` loop, and) Boolean expression which controls the loop. A `continue` statement can be labeled, so that the `continue` is applied to the correspondingly labeled loop, and not to the innermost one.

Within the translation of loop statements, the function CATCH-CONTINUE is used, which catches abnormal states, because of continues with the appropriate label. The definitions of CONTINUE and CATCH-CONTINUE are similar to those of BREAK and CATCH-BREAK, respectively.

Exceptions An exception can occur for two reasons: it can either be thrown explicitly, or implicitly by a run-time error. Java provides a statement `try` ... `catch` ... `finally` to catch exceptions. Our formalisation contains statements THROW, TRY-CATCH and TRY-CATCH-FINALLY which realise throwing and catching of exceptions. They do not play a rôle in the rest of this paper.

4.3 The formalisation of composite statements and expressions

The semantics of program constructs is described compositionally. For example, $[\![S;T]\!]$ is defined as $[\![S]\!]\,;[\![T]\!]$, where ";" is the translation of the statement composition operator ;. It is defined on $s, t\colon \mathsf{Self} \to \mathsf{StatResult[Self]}$ as:

$$
\begin{array}{l}
s\,;t \;:\; \mathsf{Self} \to \mathsf{StatResult[Self]} \stackrel{\mathrm{def}}{=} \lambda x\colon \mathsf{Self}.\ \mathsf{CASES}\ s\,x\ \mathsf{OF}\ \{ \\
\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{hang} \\
\qquad\qquad \mid \mathsf{norm}\, y \mapsto t\, y \\
\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\ \}
\end{array}
$$

$$
\begin{array}{l}
c\colon \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{bool}],\, s\colon \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}],\, x\colon \mathsf{Self} \vdash \\
\quad \mathsf{NoStops}(c, s, x) \;:\; \mathsf{nat} \to [\mathsf{result}\colon \mathsf{bool}, \mathsf{state}\colon \mathsf{Self}] \;\stackrel{\mathrm{def}}{=} \\
\qquad \lambda n\colon \mathsf{nat}.\ \mathsf{IF}\ \forall m\colon \mathsf{nat}.\, m < n \supset \\
\qquad\qquad\qquad \mathsf{CASES}\ \mathsf{iterate}(\mathsf{E2S}(c)\,;s, m)\, x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\qquad\qquad\qquad\quad \mid \mathsf{norm}\, y \mapsto\ \mathsf{CASES}\ c\, y\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{norm}\, z \mapsto z.\mathsf{res} \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{abnorm}\, b \mapsto \mathsf{false}\,\} \\
\qquad\qquad\qquad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{false}\,\} \\
\qquad\qquad \mathsf{THEN}\ \ \mathsf{CASES}\ \mathsf{iterate}(\mathsf{E2S}(c)\,;s, n)\, x\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad \mid \mathsf{hang} \mapsto (\mathsf{result} = \mathsf{false}, \mathsf{state} = x) \\
\qquad\qquad\qquad \mid \mathsf{norm}\, y \mapsto (\mathsf{result} = \mathsf{true}, \mathsf{state} = y) \\
\qquad\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto (\mathsf{result} = \mathsf{false}, \mathsf{state} = x)\,\} \\
\qquad\qquad \mathsf{ELSE}\ (\mathsf{result} = \mathsf{false}, \mathsf{state} = x)
\end{array}
$$

$$
\begin{array}{l}
c\colon \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{bool}],\, s\colon \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}],\, x\colon \mathsf{Self} \vdash \\
\quad \mathsf{NormalStopNumber?}(c, s, x) \;:\; \mathsf{nat} \to \mathsf{bool} \;\stackrel{\mathrm{def}}{=} \\
\qquad \lambda n\colon \mathsf{nat}.\ (\mathsf{NoStops}(c, s, x)\, n).\mathsf{result} \wedge \\
\qquad\qquad\qquad \mathsf{CASES}\ c\ \big((\mathsf{NoStops}(c, s, x)\, n).\mathsf{state}\big)\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\qquad\qquad\qquad\quad \mid \mathsf{norm}\, y \mapsto \neg(y.\mathsf{res}) \\
\qquad\qquad\qquad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{false}\}
\end{array}
$$

$$
\begin{array}{l}
c\colon \mathsf{Self} \to \mathsf{ExprResult}[\mathsf{Self}, \mathsf{bool}],\, s\colon \mathsf{Self} \to \mathsf{StatResult}[\mathsf{Self}],\, x\colon \mathsf{Self} \vdash \\
\quad \mathsf{AbnormalStopNumber?}(c, s, x) \;:\; \mathsf{nat} \to \mathsf{bool} \;\stackrel{\mathrm{def}}{=} \\
\qquad \lambda n\colon \mathsf{nat}.\ (\mathsf{NoStops}(c, s, x)\, n).\mathsf{result} \wedge \\
\qquad\qquad\qquad \mathsf{CASES}\ (\mathsf{E2S}(c); s)\ \big((\mathsf{NoStops}(c, s, x)\, n).\mathsf{state}\big)\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad\quad \mid \mathsf{hang} \mapsto \mathsf{false} \\
\qquad\qquad\qquad\quad \mid \mathsf{norm}\, y \mapsto \mathsf{false} \\
\qquad\qquad\qquad\quad \mid \mathsf{abnorm}\, a \mapsto \mathsf{true}\}
\end{array}
$$

Fig. 1. Auxiliary functions NoStops, NormalStopNumber? and AbnormalStopNumber? for the definition of WHILE in type theory

$$
\begin{array}{l}
ll\colon \mathsf{lift[string]}, c\colon \mathsf{Self} \to \mathsf{ExprResult[Self, bool]}, s\colon \mathsf{Self} \to \mathsf{StatResult[Self]} \;\vdash \\
\quad \mathsf{WHILE}(ll)(c)(s) \;:\; \mathsf{Self} \to \mathsf{StatResult[Self]} \;\stackrel{\mathrm{def}}{=} \\
\qquad \lambda x\colon \mathsf{Self}.\ \mathsf{LET}\ \mathit{iter\_body} = \mathsf{E2S}(c)\,;\mathsf{CATCH\text{-}CONTINUE}(ll)(s), \\
\qquad\qquad\qquad\quad \mathit{NormalStopSet} = \\
\qquad\qquad\qquad\qquad \mathsf{NormalStopNumber?}(c, \mathsf{CATCH\text{-}CONTINUE}(ll)(s), x) \\
\qquad\qquad\qquad\quad \mathit{AbnormalStopSet} = \\
\qquad\qquad\qquad\qquad \mathsf{AbnormalStopNumber?}(c, \mathsf{CATCH\text{-}CONTINUE}(ll)(s), x)\ \mathsf{IN} \\
\qquad\qquad\quad \mathsf{IF}\ \exists n\colon \mathsf{nat}.\ \mathit{NormalStopSet}\, n \\
\qquad\qquad\quad \mathsf{THEN}\ \big(\mathsf{iterate}(\mathit{iter\_body}, \varepsilon n\colon \mathsf{nat}.\ \mathit{NormalStopSet}\, n)\,; \mathsf{E2S}(c)\big)\, x \\
\qquad\qquad\quad \mathsf{ELSIF}\ \exists n\colon \mathsf{nat}.\ \mathit{AbnormalStopSet}\, n \\
\qquad\qquad\quad \mathsf{THEN}\ \big(\mathsf{iterate}(\mathit{iter\_body}, \varepsilon n\colon \mathsf{nat}.\ \mathit{AbnormalStopSet}\, n)\,; \mathit{iter\_body}\big)\, x \\
\qquad\qquad\quad \mathsf{ELSE\ hang}
\end{array}
$$

Fig. 2. WHILE in type theory, using definitions from Figure 1

Thus if statement s terminates normally in state x, resulting in a next state y, then $(s\,;t)\,x$ is $t\,y$. And if s hangs or terminates abruptly in state x, then $(s\,;t)\,x$ is $s\,x$ and t is not executed. This binary operation ; forms a monoid with the following skip statement as unit.

$$\mathsf{skip} \;:\; \mathsf{Self} \to \mathsf{StatResult[Self]} \;\stackrel{\mathrm{def}}{=}\; \lambda x\colon \mathsf{Self}.\ \mathsf{norm}\, x$$

Skip and composition are used in the following iterate function.

$$
\begin{array}{ll}
\mathsf{iterate}(s,n) \;:\; \mathsf{Self} \to \mathsf{StatResult[Self]} \;\stackrel{\mathrm{def}}{=}\; \lambda x\colon \mathsf{Self}. & \mathsf{IF}\ n = 0 \\
& \mathsf{THEN\ skip} \\
& \mathsf{ELSE\ iterate}(s, n-1)\,;s
\end{array}
$$

It will be used in the definition of the WHILE function, see Figures 1 and 2. All Java language constructs are formalised in a similar way, following closely the Java language specification [13].

The translation of the Java `while` statement depends on the occurrence of a label (immediately before the while):

$$
\begin{array}{rl}
[\![\mathtt{while(cond)}\{\mathtt{body}\}]\!] \stackrel{\mathrm{def}}{=} & \mathsf{CATCH\text{-}BREAK}(\mathsf{bot})(\\
& \quad \mathsf{WHILE}(\mathsf{bot})([\![\mathtt{cond}]\!])([\![\mathtt{body}]\!])) \\
[\![\mathtt{lab\!:\!while(cond)}\{\mathtt{body}\}]\!] \stackrel{\mathrm{def}}{=} & \mathsf{CATCH\text{-}BREAK}(\mathsf{bot})(\\
& \quad \mathsf{WHILE}(\mathsf{up}(\text{“}\mathtt{lab}\text{”}))([\![\mathtt{cond}]\!])([\![\mathtt{body}]\!]))
\end{array}
$$

The outer CATCH-BREAK(bot) makes sure that the while loop terminates normally if an unlabeled break occurs in its body. Figure 2 shows the definition of WHILE in type theory, making use of the auxiliary functions from Figure 1. The earlier given function iterate is applied to the composite statement

$$\mathsf{E2S(cond)}\,; \mathsf{CATCH\text{-}CONTINUE(lift\_label)(body)}$$

where lift_label is either bot or up("lab"). Below, this statement will be referred to as the iteration body. It first evaluates the condition (for its side-effect, discarding its result), and then evaluates the statement, making sure that occurrences of a continue (with appropriate label) in this statement are caught. The sets NormalStopNumber? and AbnormalStopNumber? in Figure 1 characterise the point where the loop will terminate in the next iteration, either because the condition becomes false, resulting in normal termination of the loop, or because an abnormality occurs in the iteration body, resulting in abnormal termination of the loop. From the definitions it follows that if NormalStopNumber? or AbnormalStopNumber? is non-empty, then it is a singleton. And if both are non-empty, then the number in NormalStopNumber? is smaller or equal than the number in AbnormalStopNumber?. Therefore, the WHILE function first checks if NormalStopNumber? is non-empty, and subsequently if AbnormalStopNumber? is non-empty. In both cases, the iteration body is executed the appropriate number of times, so that the loop will terminate in the next iteration. In the case of normal termination this is followed by an additional execution of the condition (for its side-effect), and in the case of abnormal termination this is followed by an execution of iteration body, resulting in abrupt termination. If both sets NormalStopNumber? and AbnormalStopNumber? are empty, the loop will never terminate (normally or abruptly), thus hang is returned. Basically, this definition makes WHILE a least fixed point, see [18] for details.

4.4 Hoare logic with normal termination for Java-like languages

Having described some ingredients of the semantics of statements and expressions of Java-like languages, we can formalise the notions of partial and total correctness in this context. For the moment we only consider normal termination. The predicates PartialNormal? and TotalNormal? formalise the notions of partial and total correctness, using variables pre, post: Self → bool and stat: Self → StatResult[Self].

$$\begin{aligned}\mathsf{PartialNormal?}(\mathsf{pre},\mathsf{stat},\mathsf{post}) \;:\; \mathsf{bool} \;\stackrel{\mathrm{def}}{=}\; \forall x\colon \mathsf{Self}.\, \mathsf{pre}\,x \supset \mathsf{CASES}\ \mathsf{stat}\,x\ \mathsf{OF}\ \{&\\ &\mid \mathsf{hang} \mapsto \mathsf{true}\\ &\mid \mathsf{norm}\,y \mapsto \mathsf{post}\,y\\ &\mid \mathsf{abnorm}\,a \mapsto \mathsf{true}\,\}\end{aligned}$$

$$\begin{aligned}\mathsf{TotalNormal?}(\mathsf{pre},\mathsf{stat},\mathsf{post}) \;:\; \mathsf{bool} \;\stackrel{\mathrm{def}}{=}\; \forall x\colon \mathsf{Self}.\, \mathsf{pre}\,x \supset \mathsf{CASES}\ \mathsf{stat}\,x\ \mathsf{OF}\ \{&\\ &\mid \mathsf{hang} \mapsto \mathsf{false}\\ &\mid \mathsf{norm}\,y \mapsto \mathsf{post}\,y\\ &\mid \mathsf{abnorm}\,a \mapsto \mathsf{false}\,\}\end{aligned}$$

It is easy to prove the validity of all the well-known Hoare logic proof rules, using definitions like $\{P\}\,[\![S]\!]\,\{Q\} = \mathsf{PartialNormal?}(P, [\![S]\!], Q)$. Even more, it is also easy to incorporate side-effects into these rules.

We also can formulate (and prove) extra proof rules, capturing the correctness of abruptly terminating statements. For instance, the next rule states that if we have a labeled block, containing some statement S, followed by an appropriately labeled `break` statement, then it suffices to look at the correctness of S.

$$\frac{[P]\,S\,[Q]}{[P]\ \mathsf{CATCH\text{-}BREAK}(l)(S\,;\mathsf{BREAK\text{-}LABEL}(l))\ [Q]}$$

It is immediately clear how to formulate similar rules for other abnormalities.

5 Hoare logic with abrupt termination

Unfortunately, the proof rules for normal termination do not give enough power to reason about arbitrary Java-like programs. To achieve this, it is necessary to have a "correctness notion" for being in an abnormal state, *e.g.* if execution of S starts in a state satisfying P, then execution of S terminates abruptly, because of a `return`, in a state satisfying Q. To this end, we introduce the notions of abnormal correctness. They will appear in four forms, corresponding to the four possible kinds of abnormalities. Rules will be formulated to derive the (abnormal) correctness of a program compositionally. These rules will allow the user to move back and forth between the various correctness notions.

The first notion we introduce is partial break correctness (with notation: $\{P\}\,S\,\{\mathsf{break}(Q,l)\}$), meaning that if execution of S starts in some state satisfying P, and execution of S terminates in an abnormal state, because of a `break`, then the resulting abnormal state satisfies Q. If the `break` is labeled with `lab`, then $l = \mathsf{up}(\text{“}\mathtt{lab}\text{”})$, otherwise $l = \mathsf{bot}$.

Naturally, we also have total break correctness ($[P]\,S\,[\mathsf{break}(Q,l)]$), meaning that if execution of S starts in some state satisfying P, then execution of S will terminate in an abnormal state, satisfying Q, because of a `break`. If this `break` is labeled with a label `lab`, then $l = \mathsf{up}(\text{“}\mathtt{lab}\text{”})$, otherwise $l = \mathsf{bot}$. Continuing in this manner leads to the following eight notions of abnormal correctness.

partial break correctness	$\{P\}\,S\,\{\mathsf{break}(Q,l)\}$
partial continue correctness	$\{P\}\,S\,\{\mathsf{continue}(Q,l)\}$
partial return correctness	$\{P\}\,S\,\{\mathsf{return}(Q)\}$
partial exception correctness	$\{P\}\,S\,\{\mathsf{exception}(Q,e)\}$
total break correctness	$[P]\,S\,[\mathsf{break}(Q,l)]$
total continue correctness	$[P]\,S\,[\mathsf{continue}(Q,l)]$
total return correctness	$[P]\,S\,[\mathsf{return}(Q)]$
total exception correctness	$[P]\,S\,[\mathsf{exception}(Q,e)]$

It is tempting to change the standard notation $\{P\}\,S\,\{Q\}$ and $[P]\ S\ [Q]$ into $\{P\}\,S\,\{\mathsf{norm}(Q)\}$ and $[P]\,S\,[\mathsf{norm}(Q)]$ to bring it in line with the new notation, but we will stick to the standard notation for normal termination.

The formalisation of these correctness notions in type theory is straightforward. As an example, we consider the predicate $\mathsf{PartialReturn?}$ of partial return correctness. It is used to give meaning to the notation $\{P\}\,[\![S]\!]\,\{\mathsf{return}(Q)\} = \mathsf{PartialReturn?}(P, [\![S]\!], Q)$.

$$
\begin{array}{l}
\mathsf{pre}, \mathsf{post}\colon \mathsf{Self} \to \mathsf{bool}, \mathsf{stat}\colon \mathsf{Self} \to \mathsf{StatResult[Self]} \ \vdash \\
\quad \mathsf{PartialReturn?}(\mathsf{pre}, \mathsf{stat}, \mathsf{post}) \ : \ \mathsf{bool} \ \stackrel{\mathrm{def}}{=} \\
\qquad \forall x\colon \mathsf{Self}.\, \mathsf{pre}\, x \supset \mathsf{CASES}\ \mathsf{stat}\, x\ \mathsf{OF}\ \{ \\
\qquad\qquad \mid \mathsf{hang} \mapsto \mathsf{true} \\
\qquad\qquad \mid \mathsf{norm}\, y \mapsto \mathsf{true} \\
\qquad\qquad \mid \mathsf{abnorm}\, a \mapsto\ \mathsf{CASES}\ a\ \mathsf{OF}\ \{ \\
\qquad\qquad\qquad \mid \mathsf{excp}\, e \mapsto \mathsf{true} \\
\qquad\qquad\qquad \mid \mathsf{rtrn}\, z \mapsto \mathsf{post}\, z \\
\qquad\qquad\qquad \mid \mathsf{break}\, b \mapsto \mathsf{true} \\
\qquad\qquad\qquad \mid \mathsf{cont}\, c \mapsto \mathsf{true}\,\}\,\}
\end{array}
$$

Many straightforward proof rules can be formulated and proven, for these correctness notions, like

$$
\{P\}\ \mathsf{RETURN}\ \{\mathsf{return}(P)\} \qquad\qquad \frac{[P]\,S\,[\mathsf{return}(Q)]}{[P]\,S\,;T\,[\mathsf{return}(Q)]}
$$

And finally there are rules to move between two correctness notions, from normal to abnormal and vice versa. Here are some examples for the return statement.

$$
\frac{\{P\}\,S\,\{\mathsf{return}(Q)\} \qquad \{P\}\,S\,\{Q\}}{\{P\}\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ \{Q\}} \quad \frac{[P]\,S\,[\mathsf{return}(Q)]}{[P]\ \mathsf{CATCH\text{-}STAT\text{-}RETURN}(S)\ [Q]}
$$

Most of these proof rules are easy and straightforward to formulate, and they provide a good framework to reason about programs in Java-like languages. But while loops are more interesting.

6 Hoare logic of while loops with abnormalities

In classical Hoare logic, reasoning about while loops involves the following ingredients. (1) An invariant, *i.e.* a predicate over the state space which remains true as long as the while loop is executed; (2) a condition, which is false after normal termination of the while loop; (3) a body, whose execution is iterated a number of times; (4) (when dealing with total correctness) a variant, *i.e.* a mapping from the state space to some well-founded set, which strictly decreases every time the body is executed. To extend this to abnormal correctness, we first look at a silly example of an abruptly terminating while loop.

```
while (true) { if (i < 10) { i++; } else { break; } }
```

This loop will always terminate, and we can find some variant to show this, but after termination we can not conclude that the condition has become false. We need special proof rules, from which, in this case, we can conclude that after termination of this while loop $i < 10$ does not hold (anymore). This desire leads us to the development of special rules for partial and total abnormal correctness of while loops. Below, we will describe the partial and total break correctness rules in full detail, the rules for the other abnormalities are basically the same.

6.1 Partial break while rule

Suppose that we have a while loop $\mathsf{WHILE}(l_1)(C)(S)$, which is executed in a state satisfying P. We wish to prove that if the while loop terminates abruptly, because of a break, then the result state satisfies Q—where P is the loop invariant and Q is the predicate that holds upon abrupt termination (in the example above: $i \geq 10$). A natural condition for the proof rule is thus that if the body terminates abruptly, because of a break, then Q should hold. Furthermore, we have to show that P is an invariant if the body terminates normally.

$$\frac{\begin{array}{c}\{P\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\,\{P\}\\ \{P\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\,\{\mathsf{break}(Q,l_2)\}\end{array}}{\{P\}\,\mathsf{WHILE}(l_1)(C)(S)\,\{\mathsf{break}(Q,l_2)\}}\;\text{[partial-break]}$$

Thus, assume: (1) if the iteration body $\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)$ is executed in a state satisfying P and terminates normally, then P still holds, and (2) if the iteration body is executed in a state satisfying P and ends in an abnormal state, because of a break, then this state satisfies some property Q. Then, if the while statement is executed in a state satisfying P and it terminates abruptly, because of a break, then its final state satisfies Q.

Soundness of this rule is easy to see (and to prove): suppose we have a state satisfying P, in which $\mathsf{WHILE}(l_1)(C)(S)$ terminates abruptly, because of a break. This means that the iterated statement $\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)$ terminates normally a number of times. All these times, P remains true. However, at some stage the iterated statement must terminate abruptly, because of a break, labeled l_2, and then the resulting state satisfies Q. As this is also the final state of the whole loop, we get $\{P\}\,\mathsf{WHILE}(l_1)(C)(S)\,\{\mathsf{break}(Q,l_2)\}$.

6.2 Total break while rule

Next we formulate a proof rule for the total break correctness of the while statement. Suppose that we have a state satisfying $P \wedge C$ and we wish to prove that execution of $\mathsf{WHILE}(l_1)(C)(S)$ in this state terminates abruptly, because of a break, resulting in a state satisfying Q. We have to show that (1) the iteration body terminates normally only a finite number of times (using a variant), and (2) if the iteration body does not terminate normally, it must be because of a break, resulting in an abnormal state, satisfying Q. This gives:

$$\frac{\begin{array}{c}[P \wedge C]\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}BREAK}(l_2)(\mathsf{CATCH\text{-}CONTINUE}(l_1)(S))\,[\texttt{true}] \\ \{P \wedge C \wedge \text{variant} = n\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\,\{P \wedge C \wedge \text{variant} < n\} \\ \{P \wedge C\}\,\mathsf{E2S}(C)\,;\mathsf{CATCH\text{-}CONTINUE}(l_1)(S)\,\{\mathsf{break}(Q, l_2)\}\end{array}}{[P \wedge C]\,\mathsf{WHILE}(l_1)(C)(S)\,[\mathsf{break}(Q, l_2)]}\quad[\text{total-break}]$$

The first condition states that execution of the iteration body followed by a CATCH-BREAK, in a state satisfying $P \wedge C$, always terminates normally, thus the iteration body itself must terminate either normally, or abruptly because of a break. The second condition expresses that if the iteration body terminates normally, the invariant and condition remain true and some variant decreases. Thus, the iteration body can only terminate normally a finite number of times. Finally, the last condition of this rule requires that when the iteration body terminates abruptly (because of a break), the resulting state satisfies Q. Soundness of this rule is easy to prove.

In [9] a comparable rule "(R9)" is presented, which is slightly more restrictive: it requires that the abnormality occurs when the variant becomes 0. In our case we require that it should occur at some unspecified stage.

7 An example verification of a Java program in PVS

To demonstrate the use of Hoare logic with abrupt termination, we consider the following pattern match algorithm in Java.

```
class Pattern {
  int [] base;
  int [] pattern;
  int find_pos () {
    int p = 0, s = 0;
    while (true)
      if (p == pattern.length) return s;
      else if (s + p == base.length) return -1;
           else if (base[s + p] == pattern[p]) p++;
                else { s++; p = 0; } } }
```

This algorithm is based on a pattern match algorithm described in [25]. The `it-ti` construction proposed there is programmed in Java as a `while` loop, with a condition which always evaluates to true. The loop is exited using one of two `return` statements. Explicit continues, as used in [25], are not necessary, because the loop body only consists of one `if` statement. In [21, Chapter 5] a comparable algorithm is presented which searches the position of an element in a 2-dimensional array via two (nested) while loops. If the element is found, an exception is thrown, which is caught later. This has the same effect as a return. The algorithm is derived from a specification, via rules for exceptions.

This `find_pos` algorithm in itself is not particularly spectacular, but it is a typical example of a program with a while loop, in which a key property holds upon abrupt termination (caused by a `return`). The task of the algorithm is, given two arrays `base` and `pattern`, to determine whether `pattern` occurs in `base`, and if so, to return the starting position of the first occurrence of `pattern`. The algorithm checks—in a single while loop—for each position in the array `base` whether it is the starting point of the pattern—until the pattern is found. If the pattern is found, the while loop terminates abruptly, because of a return.

In the verification of this algorithm, we assume that both `pattern` and `base` are non-null references. In the proof our Hoare logic rules are applied, until substatements do not contain loops anymore. Then in principle everything can be rewritten automatically, and no user-interaction is required. We shall briefly discuss the invariant, variant and exit condition.

Some basic ingredients of the invariant for this while loop are:

- the value of the local variable `p` ranges between 0 and `pattern.length`;
- the value of $\mathtt{s} + \mathtt{p}$ ranges between 0 and `base.length`, so that the local variable `s` is always between 0 and $\mathtt{base.length} - \mathtt{p}$;
- for every value of `p`, the sub-pattern `pattern[0]`, ... ,`pattern[p-1]` is a sub-array of `base`;
- for all i smaller than `s`, i is not a starting point for an occurence of `pattern` (*i.e.* `pattern` has not been found yet).

To prove termination of the while loop, a variant with codomain $\mathsf{nat} \times \mathsf{nat}$ is used, namely $(\mathtt{base.length} - \mathtt{s}, \mathtt{pattern.length} - \mathtt{p})$. If the loop body terminates normally, the value of this expression strictly decreases, with respect to the lexical order on $\mathsf{nat} \times \mathsf{nat}$. Either `s` is increased by one, so that the value of $\mathtt{base.length} - \mathtt{s}$ decreases by one, or `s` remains unchanged and `p` is increased by one, in which case the value of the first component remains unchanged and the value of the second component decreases.

The exit condition states that if the pattern occurs, then $\mathtt{p} = \mathtt{pattern.length}$ and the value `s`, which is the starting point of the first occurence of `pattern`, will be returned, else, if the pattern does not occur, $\mathtt{s} = \mathtt{base.length}$ and -1 will be returned. Being able to handle such exit conditions is a crucial feature of the Hoare logic described in this paper.

The correctness of this algorithm is shown in PVS 2.2 in two lemmas. The first lemma states that if the `pattern` occurs in `base`, its starting position will be returned, the other lemma states that if `pattern` does not occur, -1 will be returned. Each proof consists of approximately 250 proof commands. The crucial step in the proof is the application of the total return while rule with appropriate invariant. Rerunning the proofs takes approximately 5000 seconds on a Pentium II, 300 MHz.

8 Conclusions and future work

We have presented the essentials of a semantics of (Java) statements and expressions with abrupt termination and of an associated Hoare logic. This forms part of a wider project for reasoning about Java programs. The logic presented here is heavily used in a verification case study [17] focussing on a class from Java's standard library. Future work includes extending the semantics and proof rules to the "Annotated Java" language JML [20], consisting of standard Java with correctness assertions added as comments. Ultimately, our tool [19] should translate these assertions into appropriate verification conditions.

Acknowledgements Thanks are due to Rustan Leino for his sharp comments on an earlier version of this paper and to Joachim van den Berg for helpful discussions. Hans Meijer suggested to use the algorithm used in Section 7 as example verification.

References

1. M. Abadi and K.R.M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 682–696. Springer-Verlag, 1997.
2. K.R. Apt. Ten years of Hoare's logic: A survey—part I. *ACM Trans. on Progr. Lang. and Systems*, 3(4):431–483, 1981.
3. K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 2^{nd} rev. edition, 1997.
4. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 2^{nd} edition, 1997.
5. E.A. Ashcroft, M. Clint, and C.A.R. Hoare. Remarks on "Program proving: jumps and functions by M. Clint and C.A.R. Hoare". *Acta Informatica*, 6:317–318, 1976.
6. J.W. de Bakker. *Mathematical Theory of Program Correctness.* Prentice Hall, 1980.
7. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. Techn. Rep. CSI-R9924, Comput. Sci. Inst., Univ. of Nijmegen, 1999.
8. F.S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computation Structures*, number 1578 in LNCS, pages 135–149. Springer, Berlin, 1999.
9. F. Christian. Correct and robust programs. *IEEE Trans. on Software Eng.*, 10(2):163–174, 1984.
10. M. Clint and C.A.R. Hoare. Program proving: jumps and functions. *Acta Informatica*, 1:214–224, 1972.
11. M.J.C. Gordon. *Programming Language Theory and its Implementation.* Prentice Hall, 1988.
12. M.J.C. Gordon. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer-Verlag, 1989.
13. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
14. D. Gries. *The Science of Programming.* Springer, 1981.
15. W.O.D. Griffioen and M. Huisman. A comparison of PVS and Isabelle/HOL. In J. Grundy and M. Newey, editors, *Proceedings of the 12 International Workshop on Theorem Proving in Higher Order Logics (TPHOLs '98)*, volume 1479 of *LNCS*, September 1998.

16. U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *Proceedings of European Symposium on Programming (ESOP)*, volume 1381 of *LNCS*, pages 105–121. Springer-Verlag, March 1998.
17. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class (abstract). In B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, volume 251 - 5/1999 of *Informatik berichte FernUniversität Hagen*, 1999.
18. B. Jacobs and E. Poll. A monad for basic Java semantics. Techn. Rep. CSI-R9926, Comput. Sci. Inst., Univ. of Nijmegen, 1999.
19. 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.
20. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06c, Iowa State University, Department of Computer Science, January 1999.
21. K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Inst. of Techn., 1995.
22. R. Leino and J. van de Snepscheut. Semantics of exceptions. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 447–466. North-Holland, 1994.
23. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification (CAV '96)*, volume 1102 of *LNCS*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
24. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
25. D. Parnas. A generalized control structure and its formal definition. *Communications of the ACM*, 26(8):572–581, 1983.
26. L.C. Paulson. *Isabelle - a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. With contributions by Tobias Nipkow.
27. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, LNCS, pages 162–176. Springer, Berlin, 1999.
28. J.C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.