Verification Rules for Exception Handling in Eiffel

Emil Sekerinski and Tian Zhang

McMaster University, Hamilton, ON, Canada
{emil,zhangt26}@mcmaster.ca

Abstract. The Eiffel exception mechanism supports two methodological aspects. First, a method specification by a pre- and postcondition also determines when the method exits exceptionally, namely when the stated postcondition cannot be satisfied. Secondly, the rescue and retry statements combine catching an exception with a loop structure, thus requiring a dedicated form of correctness reasoning. We present verification rules for total correctness that take these two aspects into account. The rules handle normal loops and retry loop structures in an analogous manner. They also allow the Eiffel's mechanism to be slightly generalized. The verification rules are derived from a definition of statements by higher-order predicate transformers and have been checked with a theorem prover.

1 Introduction

Programming languages offer exception handling for responding to detected failures, for dealing with rare or undesired circumstances, and for allowing for imperfections in the design (like an incomplete implementation). Compared to treating these situations by an explicit case analysis—with testing for permissibility of an operation *a priori* or testing for success of an operation *a posteriori*—exception handling allows the original, idealized design to remain largely unchanged and *separates the concern* of exceptional situations.

The exception mechanism of Eiffel is particularly methodological in that it is combined with the specification of methods by pre- and postconditions that are evaluated at run-time [16]. When a precondition does not hold, it is the caller's fault and an exception is signalled in the caller. When a postcondition does not hold, it is the callee's fault and an exception is signalled in the callee. If the callee cannot establish the desired postcondition by alternative means, the callee propagates the exception to the caller. Thus, a single postcondition determines whether a method exits normally or exits exceptionally, i.e. *fails*. This is in contrast to the view that exceptions provide an alternative exit from methods (like "item not found"), and as such have to be mentioned in method interfaces, together with the condition when they are raised and the postcondition in that case [12,14,15]. The second methodological aspect in Eiffel is that an exception handler may *retry* a method, in which case execution continues at the beginning of a method. The exception handler has to ensure that the precondition of the method holds, independently of where the exception in the body occurred, as in following fragment:

```
meth
  require
    pre
  do
    body
  ensure
    post
  rescue
    handler
    retry
  end
```

Here, *handler* is invoked if *post* does not hold at the end of *body.* The retry statement will restart the method, hence *handler* has to establish *pre.* Unlike in the *termination model* (as in Java) and the *resumption model* (as in Mesa), the *retrying model* of exception handling leads to a loop structure [5,24]. In this paper we are concerned with the correctness theory of exception handling in the retrying model of Eiffel.

The main contribution of this paper is a mechanically formalized verification theory for total correctness based on weakest precondition predicate transformers. Predicate transformers, as introduced by Dijkstra, define the input-output behaviour of statements and at the same time allow the extraction of verification conditions. The treatment of exception handling with predicate transformers goes back to Cristian [7]: statements have one entry and multiple exits (one of those being the normal one) and are defined by a set of predicate transformers, one for each exit. As King and Morgan point out, this disallows nondeterminism [11], which is useful for the specification and development of sequential programs and necessary for defining concurrent programs. The solution is to use a single *weakest exceptional precondition* predicate transformer with one postcondition for each exit instead. Leino and Snepscheut derive weakest exceptional preconditions of statements from a trace semantics [13]. Here we start immediately with weakest exceptional preconditions. Jacobs gives a mechanical formalization of try-catch-finally statements [10]. However, that formalization includes all the other “abrupt termination” modes of Java, which we do not need for Eiffel, and uses state transformers rather than predicate transformers, which again precludes nondeterminism.

Verification rules for *partial correctness* of Eiffel statements have been proposed by Nordio et al. [17]. The present work extends these rules by considering *total correctness*, which necessitates loop variants for normal loops and *retry variants* for methods with a retry statement. Loop variants were originally considered in Eiffel, but not retry variants [16]. Nordio et al. justify the rules with respect to an operational semantics; here we derive the rules from a (denotational) predicate transformer semantics. Another difference is the linguistic form for retrying. Eiffel originally has a *retry statement* which can appear only in the exception handler.

Nordio et al. propose instead to have a *retry variable*, a boolean variable which determines if at the end of an exception handler the body is attempted again. Tschannen et al. use this form of retrying for translating Eiffel into the Boogie intermediate verification language [21]. Here we consider retry statements, as they are of interest on their own and as the current versions of EiffelStudio (version 7) and SmartEiffel (version 2.3) only support retry statements. As a consequence, all statements have three exits, the normal, exceptional, and retry exit.

All theorems (formulae (1) to (23)) have been checked with the Isabelle/HOL theorem prover; for this reason, we allow ourselves to omit proofs [1]. The formalization is a *shallow embedding* in which each statement is directly defined as a term in the logic. This style goes back to Gordon [8] and has been explored for program verification and refinement, e.g. [4,23,18]. As noted by Harrison [9], this is a more natural formalization compared to a *deep embedding*, in which the syntax of statements and their meaning are inductively defined. A shallow embedding has also the advantage that all data types and operators of the underlying logic are immediately available in the programming language for specification and reasoning. The advantage of a deep embedding, to allow proofs over the structure of statements, is not needed here. The second contribution of this paper is to work out a shallow embedding of Eiffel statements with three exits. By comparison, the formalization of Jacobs uses a deep embedding [10]. Programming languages have partially defined expressions (pointer dereferencing, array indexing, arithmetic operations) and conditional boolean operators (**and then** and **or else**), which cannot be expressed directly in HOL, a logic of *total functions*. In order to avoid a dedicated logic with partial functions, the approach is to introduce partial functions within HOL only for program expression and to continue using total functions for reasoning about statements.

An elegant way to define loops is in terms of *strong iteration* S^ω, which stands for S being repeated zero or more times, i.e., **skip**, S, S ; S, ..., but possibly infinitely often. (Weak iteration S^* repeats S only finitely often.) Such a definition allows the algebraic properties of loops to be derived, which are useful for transformations like splitting/merging loops and atomicity refinement, e.g. as in [3,6,22]. The third contribution of this paper is to explore an algebraic style of defining retry loops. Here we have statements with three exits, i.e. with three kinds of "sequential composition", one for each exit. Thus three kinds of strong iteration are defined, one for each exit.

This work originated in an effort to identify and formalize design patterns for exception handling; one of those patterns is a simpler form of retrying [19]. The formalization here covers specifically the Eiffel mechanism of retrying. The authors' work on a new notion of *partial correctness* was inspired by the methodological aspects of exception handling in Eiffel [20].

Outline. As a prelude, the meaning of program expressions with undefinedness and conditional operators is given in Section 1. The definition of Eiffel statements is split in two parts. First a core language is defined by weakest preconditions

[1] The Isabelle/HOL formalization is available at
`http://www.cas.mcmaster.ca/~zhangt26/SBMF/`

in Section 2. The remaining statements of Eiffel are defined in terms of the core statements in Section 3. Correctness assertions are derived from the weakest preconditions for all statements in Section 4. This allows the conditions for method correctness to be derived in Section 5. The example of computing the square root by binary search is used to illustrate the application of the rules in Section 6. We conclude with a summary and discussion.

Notation. Following higher order logic, every term has a type and *predicates* are boolean terms. We write $=$ for the equality of terms and $\equiv$ for the equality of predicates. Arithmetic operators bind stronger than $=$, which itself binds stronger than boolean operators, which themselves bind stronger than $\equiv$.

2 Program Expressions

Before embarking on defining statements, we need to determine on how to treat possible *undefinedness* in expressions. We distinguish *terms* in the underlying logic, here higher order logic, from *program expressions*, here those of Eiffel. A boolean term, even one like $x/y > 0$ and $a[i] < k$ is always true or false. However, the program expressions $x/y > 0$ and $a[i] < k$ may not always yield a result. For program expression E its *definedness* ΔE and *value* 'E' are in part determined by the underlying machine; the result of ΔE and 'E' are terms. Formally, a program expression of type T is a total function whose range is either some element of T or *None*.

We consider a subset of Eiffel operators on booleans and integers: assuming that c is a constant, x a variable, and $\approx$ is $=, <$ or another relational operator, $\circ$ is $+, -$, or $*$, and $\mid$ is // or \\ (integer division and modulo), we have

$$\begin{array}{llll}
\Delta c & \mathrel{\widehat{=}} \mathit{True} & `c\text{'} & \mathrel{\widehat{=}} c \\
\Delta x & \mathrel{\widehat{=}} \mathit{True} & `x\text{'} & \mathrel{\widehat{=}} x \\
\Delta(E \mathbf{\ and\ } F) & \mathrel{\widehat{=}} \Delta E \wedge \Delta F & `E \mathbf{\ and\ } F\text{'} & \mathrel{\widehat{=}} `E\text{'} \wedge `F\text{'} \\
\Delta(E \mathbf{\ or\ } F) & \mathrel{\widehat{=}} \Delta E \wedge \Delta F & `E \mathbf{\ or\ } F\text{'} & \mathrel{\widehat{=}} `E\text{'} \vee `F\text{'} \\
\Delta(E \mathbf{\ and\ then\ } F) & \mathrel{\widehat{=}} \Delta E \wedge (`E\text{'} \Rightarrow \Delta F) & `E \mathbf{\ and\ then\ } F\text{'} & \mathrel{\widehat{=}} `E\text{'} \wedge `F\text{'} \\
\Delta(E \mathbf{\ or\ else\ } F) & \mathrel{\widehat{=}} \Delta E \wedge (\neg `E\text{'} \Rightarrow \Delta F) & `E \mathbf{\ or\ else\ } F\text{'} & \mathrel{\widehat{=}} `E\text{'} \vee `F\text{'} \\
\Delta(E \approx F) & \mathrel{\widehat{=}} \Delta E \wedge \Delta F & `E \approx F\text{'} & \mathrel{\widehat{=}} `E\text{'} \approx `F\text{'} \\
\Delta(E \mid F) & \mathrel{\widehat{=}} \Delta E \wedge \Delta F \wedge `F\text{'} \neq 0 & `E \mid F\text{'} & \mathrel{\widehat{=}} `E\text{'} \mid `F\text{'} \\
\Delta(E \circ F) & \mathrel{\widehat{=}} \Delta E \wedge \Delta F \wedge & `E \circ F\text{'} & \mathrel{\widehat{=}} `E\text{'} \circ `F\text{'} \\
& \quad \mathit{min\_int} \leq `E \circ F\text{'} & & \\
& \qquad \leq \mathit{max\_int} & &
\end{array}$$

where *min_int* and *max_int* are the smallest and largest machine-representable integers, operators **and** and **or** evaluate both operands, and operators **and then** and **or else** evaluate conditionally. For example, assuming that $\mathit{min\_int} \leq 0 \leq l \leq u \leq \mathit{max\_int}$, we can show for program expression $(l + u)$ // 2 that

$$\Delta((l+u) \mathbin{//} 2) \equiv l+u \leq \mathit{max\_int} \tag{1}$$
$$`(l+u) \mathbin{//} 2\text{'} \quad = (l+u) \mathbin{//} 2 \tag{2}$$

and:

$$\Delta(l+(u-l) \mathbin{//} 2) \equiv \mathit{True} \tag{3}$$
$$`l+(u-l) \mathbin{//} 2\text{'} \quad = l+(u-l) \mathbin{//} 2 = (l+u) \mathbin{//} 2 \tag{4}$$

That is, program expressions $(l + u) \mathbin{//} 2$ and $l + (u - l) \mathbin{//} 2$ have the same value, namely the term $(l + u) \mathbin{//} 2$, but the later is always defined under above assumption, whereas the former is not. We give the proof of (4):

$$\begin{array}{lll} & `(l+(u-l) \mathbin{//} 2)\text{'} = (l+u) \mathbin{//} 2 & \\ \equiv & l+(u-l) \mathbin{//} 2 = (l+u) \mathbin{//} 2 & \text{definition of } \mathit{val} \\ \equiv & l*2+(u-l) \mathbin{//} 2*2 = (l+u) \mathbin{//} 2*2 & \text{congruence, distribution} \\ \equiv & l*2+(u-l) = l+u & \text{as } x \mathbin{//} y*y = x \text{ if } y \neq 0 \\ \equiv & l+u = l+u & \text{arithmetic} \end{array}$$

The distinction between terms in the logic and program expressions keeps the logic simple, e.g. all familiar laws of the boolean algebra like the law of the excluded middle still hold, while allowing to capture all restrictions of an underlying machine.

3 Core Statements

We consider a core language of statements with three exits, namely *normal*, *exceptional*, and *retry* exit. The statement **abort** is completely uncontrollable and the statement **stop** blocks execution. The statements **skip**, **raise**, **retry** do not modify any variables, but jump to each of the three exits directly instead. **skip** terminates normally, **raise** terminates exceptionally, and **retry** terminates retrying.

Let a, b, c be predicates. In a language with single exit, the statement **assume** a or $[a]$ terminates if a is true and blocks if a is false. With three exits, the statement $[a, b, c]$ terminates normally if a is true, terminates exceptionally if b is true, terminates retrying if c is true, and blocks if all are false. If several conditions are true, the choice among the corresponding exits is nondeterministic. The *assignment* $x := e$, where x is a variable and e is a term, always terminates normally. The *nondeterministic choice* $S \sqcap T$ executes either S if S does not block and executes T if T does not block. If both do not block, the choice is nondeterministic. The *normal (sequential) composition* $S \,;\, T$ starts with statement S and continues with statement T on normal termination of S, the *exceptional (sequential) composition* $S ;_{\mathsf{E}} T$ continues with T on exceptional termination of S, and the *retrying (sequential) composition* $S ;_{\mathsf{R}} T$ continues with T on retrying termination of S.

This is formalized by a generalization of Dijkstra's weakest precondition predicate transformers. For predicates q, r, s,

$$wp\,`S\text{'}\,(q, r, s)$$

is the weakest precondition such that S terminates, on normal termination q holds finally, on exceptional termination r holds finally, and on retrying termination s holds finally [2]:

$$\begin{array}{ll}
wp\,`\mathbf{abort}\text{'}\,(q, r, s) & \mathrel{\widehat{=}} \mathit{False} \\
wp\,`\mathbf{stop}\text{'}\,(q, r, s) & \mathrel{\widehat{=}} \mathit{True} \\
wp\,`\mathbf{skip}\text{'}\,(q, r, s) & \mathrel{\widehat{=}} q \\
wp\,`\mathbf{raise}\text{'}\,(q, r, s) & \mathrel{\widehat{=}} r \\
wp\,`\mathbf{retry}\text{'}\,(q, r, s) & \mathrel{\widehat{=}} s \\
wp\,`[a, b, c]\text{'}\,(q, r, s) & \mathrel{\widehat{=}} (a \Rightarrow q) \wedge (b \Rightarrow r) \wedge (c \Rightarrow s) \\
wp\,`x := e\text{'}\,(q, r, s) & \mathrel{\widehat{=}} q[x \backslash e] \\
wp\,`S \sqcap T\text{'}\,(q, r, s) & \mathrel{\widehat{=}} wp\,`S\text{'}\,(q, r, s) \wedge wp\,`T\text{'}\,(q, r, s) \\
wp\,`S\,;\,T\text{'}\,(q, r, s) & \mathrel{\widehat{=}} wp\,`S\text{'}\,(wp\,`T\text{'}\,(q, r, s), r, s) \\
wp\,`S ;_{\mathsf{E}}\, T\text{'}\,(q, r, s) & \mathrel{\widehat{=}} wp\,`S\text{'}\,(q, wp\,`T\text{'}\,(q, r, s), s) \\
wp\,`S ;_{\mathsf{R}}\, T\text{'}\,(q, r, s) & \mathrel{\widehat{=}} wp\,`S\text{'}\,(q, r, wp\,`T\text{'}\,(q, r, s))
\end{array}$$

As a direct consequence, we have that **stop** $= [\mathit{False}, \mathit{False}, \mathit{False}]$, **skip** $= [\mathit{True}, \mathit{False}, \mathit{False}]$, **raise** $= [\mathit{False}, \mathit{True}, \mathit{False}]$, and **retry** $= [\mathit{False}, \mathit{False}, \mathit{True}]$. For local variable declarations, let X_0 be the initial value of variables of type X and let q, r, s be predicates that do not mention variable x:

$$wp\,`\,\mathbf{local}\,x : X\;\; S\text{'}\,(q, r, s) \mathrel{\widehat{=}} (wp\,`S\text{'}\,(q, r, s))[x \backslash X_0]$$

One more construct is needed for defining loops. In a language with single-exit statements, the *iteration* S^ω repeats S an arbitrary number of times, i.e. intuitively is $\mathbf{skip} \sqcap S \sqcap (S\,;\,S) \sqcap (S\,;\,S\,;\,S)\ldots$, until S blocks. While-loops can be defined in terms of iteration by $\mathbf{while}\, g\, \mathbf{do}\, S\, \mathbf{end} = ([g]\,;\,S)^\omega\,;\,[\neg g]$. Here, statements have three exits, so three variants of iteration exist: S^ω repeats S on normal termination; if S terminates exceptionally or retrying, S^ω terminates immediately. The iteration S^{ω_E} repeats S on exceptional termination; if S terminates normally or retrying, S^{ω_E} terminates immediately. Finally, the iteration S^{ω_R} repeats S on retrying termination; if S terminates normally or exceptionally, S^{ω_R} terminates immediately.

Iterations are defined in terms of fixed points. We skip the definition here and instead give the main rule for reasoning about iterations [3]. The formulation

[2] In the formalization with Isabelle/HOL, a statement is *identified* with its predicate transformer, thus we would write $S(q, r, s)$ instead of $wp\,`S\text{'}\,(q, r, s)$. We use the latter notation here for familiarity.

[3] The Isabelle/HOL formalization contains the details.

follows the treatment of statements with single exits by Back and von Wright [2]. Let $W \neq \emptyset$ be a well-founded set, i.e. a set in which there are no infinitely decreasing chains, and let p_w for $w \in W$ be an indexed collection of predicates called *ranked predicates* of the form $p_w \equiv p \wedge v = w$. Here p is the *invariant* and v the *variant*. We define $p_{<w} \equiv (\exists\, w' \in W \cdot w' < w \wedge p_{w'})$ to be true if a predicate with lower rank than p_w is true:

$$(\forall w \in W \cdot q_w \Rightarrow wp\,`S\text{'}\,(q_{<w}, r, s)) \Rightarrow (q \Rightarrow wp\,`S^{\omega}\text{'}\,(q, r, s)) \tag{5}$$

$$(\forall w \in W \cdot r_w \Rightarrow wp\,`S\text{'}\,(q, r_{<w}, s)) \Rightarrow (r \Rightarrow wp\,`S^{\omega_{\mathsf{E}}}\text{'}\,(q, r, s)) \tag{6}$$

$$(\forall w \in W \cdot s_w \Rightarrow wp\,`S\text{'}\,(q, r, s_{<w})) \Rightarrow (s \Rightarrow wp\,`S^{\omega_{\mathsf{R}}}\text{'}\,(q, r, s)) \tag{7}$$

The first of these rules states that if under q_w statement S terminates normally while decreasing the rank of q_w, then under q statement S terminates eventually with q; if S terminates exceptionally with r or retrying with s, then S^ω terminates likewise. Similarly, the last of these rules states that if under s_w statement S terminates retrying while decreasing the rank of s_w, then under r statement S terminates eventually with s; if S terminates normally with q or exceptionally with r, then S^ω terminates likewise.

A fundamental property of weakest preconditions is *conjunctivity*; it allows the weakest precondition of a conjunction of postconditions to be determined in terms of the precondition of each of the postconditions. Let $\mathcal{Q}$ be a non-empty set of triples of predicates. Extending $\wedge$ element-wise to triples, we say that statement S is conjunctive if:

$$wp\,`S\text{'}\,(\wedge\, Q \in \mathcal{Q} \cdot Q) \equiv (\wedge\, Q \in \mathcal{Q} \cdot wp\,`S\text{'}\,Q)$$

All statements above are conjunctive or preserve conjunctivity. A consequence of conjunctivity is *monotonicity*, which states that for predicate triples Q, R:

$$(Q \Rightarrow R) \Rightarrow (wp\,`S\text{'}\,Q \Rightarrow wp\,`S\text{'}\,R)$$

where $\Rightarrow$ is extended element-wise to triples. Hence all statements above are monotonic.

Weakest preconditions allow to define various useful *domains*. The *termination domain* $tr\,`S\text{'}$ characterizes those states in which $S$ will terminate at any exit. The *normal termination domain* $nr\,`S\text{'}$, the *exceptional termination domain* $ex\,`S\text{'}$, and the *retrying termination domain* $rt\,`S\text{'}$ characterize those states in which S is guaranteed to terminate normally, exceptionally, or retrying. The *enabledness domain* $en\,`S\text{'}$ characterizes those states in which S does not block:

$$\begin{aligned}
tr\,`S\text{'} &\equiv wp\,`S\text{'}\,(True, True, True)\\
nr\,`S\text{'} &\equiv wp\,`S\text{'}\,(True, False, False)\\
ex\,`S\text{'} &\equiv wp\,`S\text{'}\,(False, True, False)\\
rt\,`S\text{'} &\equiv wp\,`S\text{'}\,(False, False, True)\\
en\,`S\text{'} &\equiv \neg wp\,`S\text{'}\,(False, False, False)
\end{aligned}$$

For example **retry** always terminates, never terminates normally or exceptionally, always terminates retrying, and never blocks. We do not go further into the properties of domain.

4 Derived Statements

The assignment $x := E$, where E is now a program expression, terminates normally if E is defined, in which case the value of E is assigned to x, and terminates exceptionally if E is undefined, without changing any variables. The statement **check** B **end** only evaluates B without changing any variables and terminates exceptionally if B is not defined or its value is false. The statements **if** B **then** S **end** and **if** B **then** S **else** T **end** also terminate exceptionally if B is not defined.

$$\begin{array}{ll}
x := E & \mathrel{\widehat{=}} [\Delta E, \neg \Delta E, \mathit{False}] \mathbin{;} x := \text{‘}E\text{’} \\
\textbf{check}\, B\, \textbf{end} & \mathrel{\widehat{=}} [\Delta B \wedge \text{‘}B\text{’}, \neg \Delta B \vee \neg\text{‘}B\text{’}, \mathit{False}] \\
\textbf{if}\, B\, \textbf{then}\, S\, \textbf{end} & \mathrel{\widehat{=}} ([\Delta B \wedge \text{‘}B\text{’}, \neg \Delta B, \mathit{False}] \mathbin{;} S) \sqcap \\
& \quad [\Delta B \wedge \neg\text{‘}B\text{’}, \neg \Delta B, \mathit{False}] \\
\textbf{if}\, B\, \textbf{then}\, S\, \textbf{else}\, T\, \textbf{end} & \mathrel{\widehat{=}} ([\Delta B \wedge \text{‘}B\text{’}, \neg \Delta B, \mathit{False}] \mathbin{;} S) \sqcap \\
& \quad ([\Delta B \wedge \neg\text{‘}B\text{’}, \neg \Delta B, \mathit{False}] \mathbin{;} T)
\end{array}$$

Immediately we have that **check** B **end** $=$ **if** B **then skip else raise end** and **if** B **then** S **end** $=$ **if** B **then** S **else skip end** as consequences.

The loop **from** S **until** B **loop** T **end** first executes S and then, as long as B is false, executes T, and repeats that provided T terminates normally. If S or T terminate exceptionally, the whole loop terminates immediately exceptionally. If S or T terminate retrying, the whole loop terminates immediately retrying.

$$\begin{array}{ll}
\textbf{from}\, S\, \textbf{until}\, B\, \textbf{loop}\, T\, \textbf{end} \mathrel{\widehat{=}} & S \mathbin{;} \\
& ([\Delta B \wedge \neg\text{‘}B\text{’}, \neg \Delta B, \mathit{False}] \mathbin{;} T)^{\omega} \mathbin{;} \\
& [\Delta B \wedge \text{‘}B\text{’}, \neg \Delta B, \mathit{False}]
\end{array}$$

The rescue statement **do** S **rescue** T **end** starts with S and if S terminates normally, the whole statement terminates normally. If S terminates exceptionally, T is executed. If T terminates normally or exceptionally, the whole statement terminates exceptionally. This is captured by $U = S ;_{\mathsf{E}} (T \mathbin{;} \textbf{raise})$. If T terminates retrying, S the whole rescue statement is attempted again. Intuitively $U^{\omega_{\mathsf{R}}} = \textbf{skip} \sqcap U \sqcap (U ;_{\mathsf{R}} U) \sqcap (U ;_{\mathsf{R}} U ;_{\mathsf{R}} U) \ldots$ repeats zero or more times. However, **do** S **rescue** T **end** repeats indefinitely when T terminates retrying and may only terminate normally or retrying. This is captured by $U^{\omega_{\mathsf{R}}} ;_{\mathsf{R}} \textbf{stop}$, hence:

$$\textbf{do}\, S\, \textbf{rescue}\, T\, \textbf{end} \mathrel{\widehat{=}} (S ;_{\mathsf{E}} (T \mathbin{;} \textbf{raise}))^{\omega_{\mathsf{R}}} ;_{\mathsf{R}} \textbf{stop}$$

This kind of exception handling differs from **try** S **catch** T **end** $= S ;_{\mathsf{E}} T$ in two respects: there is no loop structure in a try-catch statement and normal termination of handler T leads to normal termination of the whole statement but to exceptional termination in **do** S **rescue** T **end**. This means that in Eiffel the handler cannot contain an alternative computation to establish the desired postcondition, but must instead direct the body S to attempt that, typically by setting a corresponding variable and retrying.

Eiffel does not allow retry statements in the body S of **do** S **rescue** T **end**. Above definition permits those, with the meaning that the whole statement is attempted again immediately.

5 Correctness Assertions

The *total correctness assertion* $\{p\}\,S\,\{q, r, s\}$ states that under p, statement S terminates normally with q, exceptionally with r, and retrying with s:

$$\{p\}\,S\,\{q, r, s\} \mathrel{\widehat{=}} p \Rightarrow wp\,\text{‘}S\text{’}\,(q, r, s)$$

We start with two universal rules, generalizing analogous ones for single-exit statements. In a correctness assertion, the precondition can be strengthened and any of the three postconditions weakened. Also, correctness assertions of a statement can be conjoined, thus allowing proofs to be split. By convention, predicates listed on separated lines are to be conjoined:

$$\begin{array}{ll} p' \Rightarrow p & \\ \{p\}\,S\,\{q, r, s\} & \Rightarrow \ \{p'\}\,S\,\{q', r', s'\} \\ (q \Rightarrow q') \wedge (r \Rightarrow r') \wedge (s \Rightarrow s') & \end{array} \tag{8}$$

$$\begin{array}{ll} \{p\}\,S\,\{q, r, s\} & \\ \{p'\}\,S\,\{q', r', s'\} & \Rightarrow \ \{p \wedge p'\}\,S\,\{q \wedge q', r \wedge r', s \wedge s'\} \end{array} \tag{9}$$

The first of these follows from the monotonicity of $wp\,\text{‘}S\text{’}$ and the second from the conjunctivity of $wp\,\text{‘}S\text{’}$. The correctness rules for Eiffel statements are:

$$p \Rightarrow s \quad \equiv \quad \{p\}\ \mathbf{retry}\ \{q, r, s\} \tag{10}$$

$$\begin{array}{ll} p \wedge \Delta E \Rightarrow q[x \backslash \text{‘}E\text{’}] & \\ p \wedge \neg \Delta E \Rightarrow r & \equiv \ \{p\}\ x := E\ \{q, r, s\} \end{array} \tag{11}$$

$$\begin{array}{ll} p \wedge \Delta B \wedge \text{‘}B\text{’} \Rightarrow q & \\ p \wedge \neg \Delta B \Rightarrow r & \equiv \ \{p\}\ \mathbf{check}\ B\ \mathbf{end}\ \{q, r, s\} \\ p \wedge \neg\text{‘}B\text{’} \Rightarrow r & \end{array} \tag{12}$$

$$\begin{array}{ll} \{p\}\,S\,\{t, r, s\} & \\ \{t\}\ T\,\{q, r, s\} & \Rightarrow \ \{p\}\,S\,;\,T\,\{q, r, s\} \end{array} \tag{13}$$

$$\begin{array}{ll} \{p \wedge \Delta B \wedge \text{‘}B\text{’}\}\,S\,\{q, r, s\} & \\ p \wedge \Delta B \wedge \neg\text{‘}B\text{’} \Rightarrow r & \Rightarrow \ \{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{end}\ \{q, r, s\} \\ p \wedge \neg \Delta B \Rightarrow s & \end{array} \tag{14}$$

$$\begin{array}{ll} \{p \wedge \Delta B \wedge \text{‘}B\text{’}\}\,S\,\{q, r, s\} & \\ \{p \wedge \Delta B \wedge \neg\text{‘}B\text{’}\}\ T\,\{q, r, s\} & \Rightarrow \ \{p\}\ \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{end}\ \{q, r, s\} \\ p \wedge \neg \Delta B \Rightarrow s & \end{array} \tag{15}$$

For the loop **from** S **until** B **loop** T **end**, we assume that the postconditions are of a particular form: at normal termination, the loop invariant holds, B is defined and true. At exceptional termination, either the exceptional postcondition of S or T holds (in case S or T failed), or the invariant holds and B is not defined (in case the evaluation of B failed). On retrying termination, the retrying postcondition of S or T holds (in case S or T executed **retry**). The role of S is to establish the loop invariant, here q:

$$\begin{array}{ll} & \{p\}\, S\, \{q, r, s\} \\ & \{q_w \wedge \Delta B \wedge \neg\text{`}B\text{'}\}\ T\ \{q_{<w}, r, s\} \\ \Rightarrow & \\ & \{p\}\ \textbf{from}\, S\, \textbf{until}\, B\, \textbf{loop}\ T\, \textbf{end}\, \{q \wedge \Delta B \wedge \text{`}B\text{'}, r \vee (q \wedge \neg\Delta B), s\} \end{array} \tag{16}$$

Recall that $q_w = q \wedge v = w$ where q is the invariant, v the variant, and $w \in W$. In Eiffel, variants are integer expressions and the well-founded set W of their values are non-negative integers. For integer variants, we have the following rule, where $w > 0$:

$$\begin{array}{ll} & \{p\}\, S\, \{q, r, s\} \\ & \{q \wedge v = w \wedge \Delta B \wedge \neg\text{`}B\text{'}\}\ T\ \{q \wedge v < w, r, s\} \\ \Rightarrow & \\ & \{p\}\ \textbf{from}\, S\, \textbf{until}\, B\, \textbf{loop}\ T\, \textbf{end}\, \{q \wedge \Delta B \wedge \text{`}B\text{'}, r \vee (q \wedge \neg\Delta B), s\} \end{array} \tag{17}$$

The rule for **do** S **rescue** T **end** requires that progress towards termination is made whenever S or T exits retrying; termination here means normal termination if S terminates normally or exceptional termination if T terminates normally or exceptionally:

$$\begin{array}{ll} & \{p_w\}\, S\, \{q, t_w, p_{<w}\} \\ & \{t_w\}\ T\ \{r, r, p_{<w}\} \\ \Rightarrow & \\ & \{p\}\ \textbf{do}\, S\, \textbf{rescue}\ T\, \textbf{end}\, \{q, r, s\} \end{array} \tag{18}$$

For integer variants, we have following rule, where $w > 0$:

$$\begin{array}{ll} & \{p \wedge v = w\}\, S\, \{q, t \wedge v = w, p \wedge v < w\} \\ & \{t \wedge v = w\}\ T\ \{r, r, p \wedge v < w\} \\ \Rightarrow & \\ & \{p\}\ \textbf{do}\, S\, \textbf{rescue}\ T\, \textbf{end}\, \{q, r, s\} \end{array} \tag{19}$$

Here p is the *retry invariant* and v is the *retry variant.*

6 Method Correctness

In Eiffel, each method is specified by a single precondition and single postcondition only. The normal exit is taken if the desired postcondition is established and the exceptional exit is taken if the desired postcondition cannot be established. Thus the situations under which an exceptional exit is taken is implicit in the method specification and a “defined” outcome is always possible, even in the presence of unanticipated failures. Since methods never terminate retrying, and some statements only terminate normally, we introduce two abbreviations:

$$\begin{aligned} \{p\}\, S\, \{q, r\} &\mathrel{\widehat{=}} \{p\}\, S\, \{q, r, \mathit{False}\} \\ \{p\}\, S\, \{q\} &\mathrel{\widehat{=}} \{p\}\, S\, \{q, \mathit{False}\} \end{aligned}$$

We propose to restrict the exceptional postcondition in case the specified postcondition cannot be established [20]. Since classes typically have a class invariant, the class invariant should hold even at exceptional termination, as otherwise the program is left in an inconsistent state and a subsequent call to the same object may fail. (As a consequence, if re-establishing the class invariant cannot be guaranteed, the class invariant needs to be weakened appropriately.) More generally, let p be the condition that holds before a call to method m with body **local** $x : X$ **do** S **rescue** T **end**, where p captures the computation that has been made by the whole program up to this point. We then require a call to m either to terminate normally with the desired postcondition q or terminate exceptionally with p:

$$\{p\}\ \textbf{local}\, x : X\ \textbf{do}\, S\ \textbf{rescue}\, T\ \textbf{end}\, \{q, p\}$$

That is, in case of failure, the method may leave the state changed, but has to undo sufficiently such that p holds again. This regime allows then failures to be propagated back over arbitrarily many method calls. From the correctness theorems for statements, we get immediately following rule, where p, q are predicates that may not mention x and p'_w is a collection of ranked predicates.

$$\begin{array}{lcl} p \wedge x = X_0 \Rightarrow p' & & \\ \{p'_w\}\, S\, \{q', t_w, p'_{<w}\} & & \{p\} \\ \{t_w\}\, T\, \{p', p', p'_{<w}\} & \Rightarrow & \textbf{local}\, x : X\ \textbf{do}\, S\ \textbf{rescue}\, T\ \textbf{end} \\ p' \Rightarrow p & & \{q, p\} \\ q' \Rightarrow q & & \end{array} \tag{20}$$

For integer variants, we have following rule, where $w > 0$:

$$\begin{array}{lcl} p \wedge x = X_0 \Rightarrow p' & & \\ \{p' \wedge v = w\}\, S\, \{q', t \wedge v = w, p' \wedge v < w\} & & \{p\} \\ \{t \wedge v = w\}\, T\, \{p', p', p' \wedge v < w\} & \Rightarrow & \textbf{local}\, x : X\ \textbf{do}\, S\ \textbf{rescue}\, T\ \textbf{end} \\ p' \Rightarrow p & & \{q, p\} \\ q' \Rightarrow q & & \end{array} \tag{21}$$

7 Example: Binary Search of Square Root

Suppose the task is to compute the approximate non-negative integer square root of n, which is a non-negative integer itself, such that $\textbf{Result}^2 \leq n <$

(**Result** $+1)^2$ using bounded arithmetic [4]. Assume that the result must be between l and u. The loop

```
**from until** $u - l = 1$ **loop**
  $m := l + (u - l) \mathbin{//} 2$
  **if** $n < m * m$ **then** $u := m$ **else** $l := m$ **end**
**end**
```

maintains the invariant $p \equiv 0 \leq l < u \wedge l^2 \leq n < u^2$. The statement $m := l + (u - l) \mathbin{//} 2$ will establish $m = (l + u) \mathbin{//} 2$ according to (4) and never fail according to (3). However, the **if** statement will fail if $m * m > max\_int$. Since necessarily $n \leq max\_int$, we know that in case of failure $n < m * m$, thus after assigning $u := m$ the loop can continue. We use the abbreviation {retry: q} for $\{False, False, q\}$. The full implementation with annotation is as follows:

```
$sqrt(n, l, u : INTEGER) : INTEGER$
  $\{p\}$
  **local**
    $m : INTEGER$
  {retry invariant: $p$}
  {retry variant: $u - l$}
  **do**
    {loop invariant: $p$}
    {loop variant: $u - l$}
    **from until** $u - l = 1$ **loop**
      $m := l + (u - l) \mathbin{//} 2$
      $\{p \wedge m = (l + u) \mathbin{//} 2\}$
      **if** $n < m * m$ **then** $u := m$ **else** $l := m$ **end**
      $\{p, p \wedge m = (l + u) \mathbin{//} 2 \wedge n < m^2\}$
    **end**
    $\{p \wedge u - l = 1\}$
    **Result** $:= l$
  **rescue**
    $\{p \wedge m = (l + u) \mathbin{//} 2 \wedge n < m^2\}$
    $u := m$
    $\{p\}$
    **retry**
    {retry: $p$}
  **end**
  $\{\mathbf{Result}^2 \leq n < (\mathbf{Result} + 1)^2\}$
```

Note that the retry loop only needs to decrease the variant on the retry exit.

[4] The Eiffel Standard [1] and Meyer [16] suggest that an arithmetic overflow leads to an exception. SmartEiffel (version 2.3) does raise an exception, but EiffelStudio (version 7) does not. However, the example can be expressed in EiffelStudio by first formulating a class for safe arithmetic, see `http://www.cas.mcmaster.ca/~zhangt26/SBMF/`

8 Discussion

In this paper we have derived verification rules for the retrying mechanism of Eiffel exceptions. Beside the contribution of total correctness rules, the novel aspects of the derivation are that we started with a weakest exceptional precondition semantics and defined both normal loops and retry loops through strong iteration. All theorems have been checked with Isabelle/HOL.

The statements considered include the **check** statement, but we have not discussed **ensure** and **require** method specifications. Since these are evaluated at run-time in Eiffel, they are restricted to be program expressions (extended with the **old** notation). However, since these are evaluated program expression they have be treated like the **check** statement. It should be straightforward to extend the approach for method correctness (Sec. 6) accordingly.

We have neither considered dynamic objects, therefore no method calls, nor other features of Eiffel like inheritance. While we believe that exception handling is largely independent of other features and the treatment here would carry over to a more general setting, this remains to be shown.

Strong and weak iteration are appealing because of their rich algebraic structure. However, we have not explored the resulting algebraic properties of rescue and retry statements. For example, following theorems can be shown to hold:

$$\mathbf{do\,skip\,rescue}\,S\,\mathbf{end} = \mathbf{skip} \tag{22}$$

$$\mathbf{do\,raise\,rescue\,retry\,end} = \mathbf{abort} \tag{23}$$

An interesting consequence of our definition of statements is that retry statements can also appear in the main body of a method, not only the exception handler. The proof rule (18) supports this use. With this, the binary search of the square root example can be rewritten without the **from** / **until** loop, using only the retry loop:

```
$sqrt2(n, l, u : INTEGER) : INTEGER$
  $\{p\}$
  local
    $m : INTEGER$
  $\{\text{retry invariant: } p\}$
  $\{\text{retry variant: } u - l\}$
  do
    $m := l + (u - l) \,//\, 2$
    $\{p \wedge m = (l + u) \,//\, 2\}$
    if $n < m * m$ then $u := m$ else $l := m$ end
    $\{p, p \wedge m = (l + u) \,//\, 2 \wedge n < m^2\}$
    if $u - l > 1$ then retry end
    $\{p \wedge u - l = 1, \text{retry: } p \wedge u - l > 1\}$
    Result $:= l$
  rescue
```

```
    {$p \wedge m = (l + u) \mathbin{//} 2 \wedge n < m^2$}
    $u := m$
    {$p$}
    **retry**
    {retry: $p$}
  **end**
  {$\mathbf{Result}^2 \leq n < (\mathbf{Result} + 1)^2$}
```

Nordio et al. propose to replace the retry statement with a retry variable in order to avoid the third exit [17]. Below is their example of safe division, with annotation to show termination of the retry loop; the example shows that the third exit does not cause further complications:

```
safe_division ($x, y$ : INTEGER) : INTEGER
  local
    $z$ : INTEGER
  {retry invariant: $(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0))$}
  {retry variant: $1 - z$}
  do
    $\mathbf{Result} := x \mathbin{//} (y + z)$
    {$(y = 0 \Rightarrow \mathbf{Result} = x) \wedge (y \neq 0 \Rightarrow \mathbf{Result} = x \mathbin{//} y), y = 0 \wedge z = 0$}
  rescue
    {$y = 0 \wedge z = 0$}
    $z := 1$
    {$y = 0 \wedge z = 1$}
    retry
    {retry: $y = 0 \wedge z = 1$}
  end
  {$(y = 0 \Rightarrow \mathbf{Result} = x) \wedge (y \neq 0 \Rightarrow \mathbf{Result} = x \mathbin{//} y)$}
```

Acknowledgement. We are grateful for the helpful comments of the reviewers. In particular, one reviewer suggested theorems (22) and (23).

References

1. Eiffel: Analysis, Design and Programming Language, 2nd edn. Standard ECMA-367. Ecma International (June 2006)
2. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
3. Back, R.J.R., von Wright, J.: Reasoning algebraically about loops. Acta Informatica 36(4), 295–334 (1999)
4. Bowen, J., Gordon, M.: A shallow embedding of Z in HOL. Information and Software Technology 37(5-6), 269–276 (1995)
5. Buhr, P.A., Russell Mok, W.Y.: Advanced exception handling mechanisms. IEEE Transactions on Software Engineering 26(9), 820–836 (2000)

6. Cohen, E.: Separation and Reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)
7. Cristian, F.: Correct and robust programs. IEEE Transactions on Software Engineering 10(2), 163–174 (1984)
8. Gordon, M.J.C.: Mechanizing programming logics in higher order logic. In: Birtwistle, G., Subrahmanyam, P.A. (eds.) Current Trends in Hardware Verification and Automated Theorem Proving, pp. 387–439. Springer, New York (1989)
9. Harrison, J.: HOL Light tutorial (for version 2.20). Technical report, Intel JF1-13 (January 2011)
10. Jacobs, B.: A Formalisation of Java's Exception Mechanism. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 284–301. Springer, Heidelberg (2001)
11. King, S., Morgan, C.: Exits in the refinement calculus. Formal Aspects of Computing 7(1), 54–76 (1995)
12. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Software Engineering Notes 31, 1–38 (2006)
13. Leino, K.R.M., van de Snepscheut, J.L.A.: Semantics of exceptions. In: Olderog, E.-R. (ed.) PROCOMET 1994: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi. IFIP Transactions A-56, pp. 447–466. North-Holland Publishing Co., Amsterdam (1994)
14. Rustan, M., Leino, K., Schulte, W.: Exception safety for C#. In: Software Engineering and Formal Methods, SEFM 2004, pp. 218–227. IEEE Computer Society (2004)
15. Liskov, B., Guttag, J.: Program Development in Java: Abstraction, Specification, and Object-Oriented Design. Addison-Wesley Longman Publishing Co., Boston (2000)
16. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (1997)
17. Nordio, M., Calcagno, C., Müller, P., Meyer, B.: A Sound and Complete Program Logic for Eiffel. In: Oriol, M., Meyer, B. (eds.) TOOLS EUROPE 2009. LNBIP, vol. 33, pp. 195–214. Springer, Heidelberg (2009)
18. von Oheimb, D.: Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic. PhD thesis, Technische Universität München (2001)
19. Sekerinski, E.: Exceptions for dependability. In: Petre, L., Sere, K., Troubitsyna, E. (eds.) Dependability and Computer Engineering: Concepts for Software-Intensive Systems—a Handbook on Dependability Research, pp. 11–35. IGI Global (2011)
20. Sekerinski, E., Zhang, T.: Partial correctness for exception handling. In: Bonakdarpour, B., Maibaum, T. (eds.) Proceedings of the 2nd International Workshop on Logical Aspects of Fault-Tolerance, pp. 116–132 (June 2011)
21. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Verifying Eiffel programs with Boogie. In: Rustan, K., Leino, M., Moskal, M. (eds.) First International Workshop on Intermediate Verification Languages: BOOGIE 2011. CADE 23 Workshop, pp. 14–26 (2011)
22. von Wright, J.: Towards a refinement algebra. Science of Computer Programming 51(1-2), 23–45 (2004); Mathematics of Program Construction (MPC 2002)
23. Wildmoser, M., Nipkow, T.: Certifying Machine Code Safety: Shallow Versus Deep Embedding. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 305–320. Springer, Heidelberg (2004)
24. Yemini, S., Berry, D.M.: A modular verifiable exception handling mechanism. ACM Trans. Program. Lang. Syst. 7(2), 214–243 (1985)