A Sound and Complete Program Logic for Eiffel

Martin Nordio[1], Cristiano Calcagno[2], Peter Müller[1], and Bertrand Meyer[1]

[1] ETH Zurich, Switzerland
{Martin.Nordio,Peter.Mueller,Bertrand.Meyer}@inf.ethz.ch
[2] Imperial College, London, UK
ccris@doc.ic.ac.uk

Abstract. Object-oriented languages provide advantages such as reuse and modularity, but they also raise new challenges for program verification. Program logics have been developed for languages such as C# and Java. However, these logics do not cover the specifics of the Eiffel language. This paper presents a program logic for Eiffel that handles exceptions, once routines, and multiple inheritance. The logic is proven sound and complete w.r.t. an operational semantics. Lessons on language design learned from the experience are discussed.

Key words: Software verification, program proofs, operational semantics, Eiffel

1 Introduction

Program verification relies on a formal semantics of the programming language, typically a program logic such as Hoare logic. Program logics have been developed for mainstream object-oriented languages such as Java and C#. For instance, Poetzsch-Heffter and Müller presented a Hoare-style logic for a subset of Java [18]. This logic includes the most important features of object-oriented languages such as abstract types, dynamic binding, subtyping, and inheritance. However, exception handling is not treated in their work. Huisman and Jacobs [4] developed a Hoare-stype logic which handles abrupt termination. It includes not only exception handling but also break, continue, and return statements. Their logic was developed to verify Java-like programs.

Eiffel has several distinctive features not present in mainstream languages, for instance, a different exception handling mechanism, once routines, and multiple inheritance. Eiffel's once routines (methods) are used to implement global behavior, similarly to static fields and methods in Java. Only the first invocation of a once routine triggers an execution of the routine body; subsequent invocations return the result of the first execution. The development of formal techniques for these concepts does not only allow formally verifying Eiffel programs, but also allows comparing the different concepts, and analyzing which concepts are more suitable to be applied for formal verification.

The main contributions of this paper are an operational and an axiomatic semantics for Eiffel, and soundness and completeness results. The semantics includes: (1) basic instructions such as loops, compounds and assignments; (2) routine invocations; (3) exceptions; (4) once routines, and (5) multiple inheritance. During this work, we have found that Eiffel's exception mechanism was not ideal for formal verification. The use of retry instructions in a rescue clause complicates its verification. For this reason, a change in the Eiffel exception handling mechanism has been proposed, and will be adopted by a future revision of the language standard.

Outline. Section 2 describes the subset of Eiffel and its operational semantics. Section 3 presents the Eiffel program logic. An example that illustrates the use of the logic is described in Section 4. The soundness and completeness theorems are presented in Section 5. Section 6 discusses related work. Finally, in Section 7 we compare the Eiffel features such as exception handling and once routines with other object-oriented languages features such as `try-catch` instructions and static methods. Proof of soundness and completeness for our logic are presented in a technical report [11].

2 A Semantics for Eiffel

2.1 The Source Language

The source language is a subset of Eiffel, which includes the most important features of Eiffel, although agents are omitted being beyond the scope of this work. The most interesting concepts supported by this subset are: (1) multiple inheritance, (2) exception handling, and (3) once routines.

An Eiffel program is a sequence of class declarations. A class declaration consist of an optional inheritance clause, and a class body. The inheritance clause supports multiple inheritance, and allows undefining, redefining, and renaming routines. If the routine is redefined, preconditions of subclasses can be weaker, and postconditions can be stronger. A class body is a sequence of attribute declarations or routine declarations. For simplicity, routines are functions that take always one argument and return a result. However, we do not assume that functions are side-effect free, that is, our logic fully handles heap modifications. The source language also supports once routines.

The syntax of the subset of Eiffel is presented in Figure 1. Class names, routine names, variables and attributes are denoted by *ClassId*, *RoutineId*, *VarId*, and *AttributeId*, respectively. The set of variables is denoted by *Var*; *VarId* is an element of *Var*. The arithmetic functions $*$ and $+$ are defined as usual, and *list_of* denotes a comma-separated list. For space restrictions, we omit attribute reading, attribute writing, and object creation here, but these features are presented in our technical report [11].

Boolean expressions and expressions (*BoolExp* and *Exp*) are side-effect-free, and do not trigger exceptions[1]. In addition, we write *ExpE* for the expressions which might raise exceptions. For simplicity, expressions *ExpE* are only allowed

[1] the necessary checks are delegated to the compiler.

in assignments. This assumption simplifies the presentation of the logic, especially the rules for routine invocation, `if then else`, and `loop` instructions. However, the logic could be easily extended.

One of the design goals of our logic is that programs behave in the same way when contracts are checked at runtime and when they are not. For this reason, we demand that expressions occurring in contracts to be side-effect-free, and to not trigger exceptions.

```
Program    ::= ClassDecl*
ClassDecl  ::= class ClassId [inherit Parent+] MemberDecl * end
Type       ::= boolT  |  intT  | ClassId  | voidT
Parent     ::= Type [Undefine] [Redefine] [Rename]
Undefine   ::= undefine list_of RoutineId
Redefine   ::= redefine list_of RoutineId
Rename     ::= rename list_of (RoutineId as RoutineId)
MemberDecl ::= AttributeId Type  |  Routine
Routine    ::= RoutineId (VarId : Type) : Type
                 require BoolExp
                 [ local list_of (VarId : Type) ]
                 (do | once)
                   Instr
                 [ rescue Instr ]
                 ensure BoolExp
                 end
Instr      ::= VarId := ExpE  |  Instr; Instr  |  check BoolExp end
              | from invariant BoolExp until BoolExp loop Instr end
              | if BoolExp then Instr else Instr end
              | VarId := VarId.Type : RoutineId (Exp )
Exp, ExpE  ::= Literal | VarId | Exp Op Exp | BoolExp
BoolExp    ::= Literal | VarId | BoolExp Bop BoolExp | Exp CompOp Exp
Op         ::= + | − | ∗ | //
Bop        ::= and | or | xor | and then | or else | implies
CompOp     ::= < | > | <= | >= | = | /=
```

Fig. 1. Syntax of the subset of Eiffel.

2.2 The Memory Model

The state of an Eiffel program describes the current values of local variables, arguments, the current object, and the current object store \$. A value is either a boolean, or an integer, or the void value, or an object reference. An object is characterized by its class and an identifier of infinite sort *ObjId*. The data type *Value* models values; its definition is the following:

data type *Value* = **boolV** *Bool* | **intV** *Int* | **objV** *ClassId ObjId* | **voidV**

The function $\tau : \mathit{Value} \rightarrow \mathit{Type}$ returns the dynamic type of a value, where $\tau(\mathbf{boolV}\ b) = \mathit{boolT}$; $\tau(\mathbf{intV}\ n) = \mathit{intT}$; $\tau(\mathbf{objV}\ \mathit{cId}\ \mathit{oId}) = \mathit{cId}$; and $\tau(\mathbf{voidV}) = \mathit{voidT}$.

The state of an object is defined by the values of its attributes. The sort *Attribute* defines the attribute declaration $T@a$ where a is an attribute declaration in the class T. We use a sort *Location*, and a function *instvar* where $\mathit{instvar}(V, T@a)$ yields the instance of the attribute $T@a$ if V is an object reference, and the object has an attribute $T@a$; otherwise it yields *undef*. The datatype definitions and the signature of *instvar* are the following:

$$\begin{array}{l}
\textbf{data type}\ \mathit{Attribute} = \mathit{Type}\ \mathit{AttributeId} \\
\textbf{data type}\ \mathit{Location}\ = \mathit{ObjId}\ \mathit{AttributeId} \\
\mathit{instvar} : \mathit{Value}\ \times\ \mathit{Attribute} \rightarrow \mathit{Location}\ \cup \{\mathit{undef}\}
\end{array}$$

The object store models the heap describing the states of all objects in a program at a certain point of execution. An object store is modeled by an abstract data type *ObjectStore*. We use the object store presented by Poetzsch-Heffter and Müller [17, 18]. The following operations apply to the object store: $os(l)$ denotes reading the location l in the object store os; $\mathit{alive}(o, os)$ yields true if and only if object o is allocated in the object store os; $\mathit{new}(os, C)$ yields a reference of type C to a new object in the store os; $os < l := v >$ updates the object store os at the location l with the value v; $os < C >$ denotes the store after allocating a new object of type C. An axiomatization of these functions is presented in [17].

$$\begin{array}{rl}
\_(\_) : & \mathit{ObjectStore}\ \times\ \mathit{Location} \rightarrow\ \mathit{Value} \\
\mathit{alive} : & \mathit{Value} \times \mathit{ObjectStore} \rightarrow \mathit{Bool} \\
\mathit{new} : & \mathit{ObjectStore}\ \times\ \mathit{ClassId} \rightarrow\ \mathit{Value} \\
\_ < \_ := \_ > : & \mathit{ObjectStore}\ \times\ \mathit{Location}\ \times\ \mathit{Value} \rightarrow \mathit{ObjectStore} \\
\_ < \_ > : & \mathit{ObjectStore}\ \times\ \mathit{ClassId} \rightarrow\ \mathit{ObjectStore}
\end{array}$$

2.3 Operational Semantics

Program states are a mapping from local variables and arguments to values, and from the current object store \$ to *ObjectStore*. The program state is defined as follows:

$$\begin{array}{l}
\mathit{State} \equiv \mathit{Local}\ \times\ \mathit{Heap} \\
\mathit{Local} \equiv \mathit{VarId} \cup \{\mathit{Current}, p, \mathit{Result}, \mathit{Retry}\} \rightarrow \mathit{Value} \cup \{\mathit{undef}\} \\
\mathit{Heap} \equiv \{\$\} \rightarrow \mathit{ObjectStore}
\end{array}$$

Local maps local variables, the receiver object *Current* (*this* in Java), arguments, *Result*, and *Retry* to values. Arguments are denoted by p. The variables *Result* and *Retry* are special variables used to store the result value, and the retry value but they are not part of *VarId*. For this reason, these variables are included explicitly.

For $\sigma \in \mathit{State}$, $\sigma(e)$ denotes the evaluation of the expression e in the state σ. Its signature is the following:

$$\sigma : \mathit{ExpE} \rightarrow \mathit{Value} \cup \{\mathit{exc}\}$$

The evaluation of an expression e can yield *exc* meaning that an exception was triggered in e. For example, $\sigma(x/0)$ yields *exc*. Furthermore, the evaluation $\sigma(y \neq 0 \wedge x/y = z)$ is different from *exc* because σ first evaluates $y \neq 0$ and then evaluates $x/y = z$ only if $y \neq 0$ evaluates to *true*. The state $\sigma[x := V]$ denotes the state obtained after the replacement of x by V in the state σ.

The transitions of the operational semantics have the form: $\langle \sigma, S \rangle \rightarrow \sigma', \chi$ where σ and σ' are states, S is an instruction, and χ is the current status of the program. The value of χ can be either the constant *normal* or *exc*. The variable χ is required to treat abrupt termination. The transition $\sigma, S \quad \rightarrow \sigma', \mathit{normal}$ expresses that executing the instruction S in the state σ terminates normally in the state σ'. The transition $\sigma, S \quad \rightarrow \sigma', \mathit{exc}$ expresses that executing the instruction S in the state σ terminales with an exception in the state σ'.

In the following, we present the operational semantics. First, we present the operational semantics for basic instructions. Second, we define the semantics for routine invocation. Finally, we show the semantics for exception handling and once routines. The operational semantics for exception handling and once routines is one of the contributions of this paper.

Basic Instructions. Figure 2 presents the operational semantics for basic instructions such as assignment, compound, conditional, and loop instructions. The operational semantics for assignment, conditional, and loop instructions is standard. Compound is defined by two rules: in rule (2.3) the instruction s_1 is executed and an exception is triggered. The instruction s_2 is not executed, and the state of the compound is the state produced by s_1. In rule (2.4), s_1 is executed and terminates normally. The state of the compound is the state produced by s_2.

The check instruction helps to express a property that one believes will be satisfied. If the property is satisfied then the system does not change. If the property is not satisfied then an exception is triggered. The semantics for check consist of two rules: if the condition of the check instruction evaluates to true, then the instruction terminates normally, rule (2.5); otherwise the check instruction triggers an exception, rule (2.6).

Routine Invocations. Poetzsch-Heffter and Müller [18] have developed an operational and axiomatic semantics for a Java-like languages which handle inheritance, dynamic binding, subtyping and abstract types. However, the source language used in their work has single inheritance. In this section, we extend their logic to support multiple inheritance.

Poetzsch-Heffter and Müller distinguish between virtual routines and routine implementation. A class T has a *virtual routine* $T{:}m$ for every routine m that it declares or inherits. A class T has a *routine implementation* $T@m$ for every routine m that it defines (or redefines). We assume in the following that every

Assignment Instruction

$$\frac{\sigma(e) = \mathit{exc}}{\langle \sigma, x := e \rangle \to \sigma, \mathit{exc}} \quad (2.1) \qquad \frac{\sigma(e) \neq \mathit{exc}}{\langle \sigma, x := e \rangle \to \sigma[x := \sigma(e)], \mathit{normal}} \quad (2.2)$$

Compound

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', \mathit{exc}}{\langle \sigma, s_1; s_2 \rangle \to \sigma', \mathit{exc}} \quad (2.3) \qquad \frac{\langle \sigma, s_1 \rangle \to \sigma', \mathit{normal} \qquad \langle \sigma', s_2 \rangle \to \sigma'', \chi}{\langle \sigma, s_1; s_2 \rangle \to \sigma'', \chi} \quad (2.4)$$

Check Instruction

$$\frac{\sigma(e) = \mathit{true}}{\langle \sigma, \texttt{check } e \texttt{ end} \rangle \to \sigma, \mathit{normal}} \quad (2.5) \qquad \frac{\sigma(e) = \mathit{false}}{\langle \sigma, \texttt{check } e \texttt{ end} \rangle \to \sigma, \mathit{exc}} \quad (2.6)$$

Conditional Instruction

$$\frac{\sigma(e) = \mathit{true} \qquad \langle \sigma, s_1 \rangle \to \sigma', \chi}{\langle \sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \rangle \to \sigma', \chi} \quad (2.7)$$

$$\frac{\sigma(e) = \mathit{false} \qquad \langle \sigma, s_2 \rangle \to \sigma', \chi}{\langle \sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \rangle \to \sigma', \chi} \quad (2.8)$$

Loop Instruction

$$\frac{\sigma(e) = \mathit{true}}{\langle \sigma, \texttt{from invariant } I \texttt{ until } e \texttt{ loop } s_1 \texttt{ end} \rangle \to \sigma, \mathit{normal}} \quad (2.9)$$

$$\frac{\sigma(e) = \mathit{false} \qquad \langle \sigma, s_1 \rangle \to \sigma', \mathit{exc}}{\langle \sigma, \texttt{from invariant } I \texttt{ until } e \texttt{ loop } s_1 \texttt{ end} \rangle \to \sigma', \mathit{exc}} \quad (2.10)$$

$$\frac{\begin{array}{c}\sigma(e) = \mathit{false} \qquad \langle \sigma, s_1 \rangle \to \sigma', \mathit{normal} \\ \langle \sigma', \texttt{from invariant } I \texttt{ until } e \texttt{ loop } s_1 \texttt{ end} \rangle \to \sigma'', \chi\end{array}}{\langle \sigma, \texttt{from invariant } I \texttt{ until } e \texttt{ loop } s_1 \texttt{ end} \rangle \to \sigma'', \chi} \quad (2.11)$$

Fig. 2. Operational Semantics for Basic Instructions

invocation is decorated with the virtual method being invoked. The semantics of routine invocations uses two functions: *body* and *impl*. The function $\mathit{impl}(T, m)$ yields the implementation of routine m in class T. This implementation could be defined by T or inherited from a superclass. The function *body* yields the instruction constituting the body of a routine implementation. The signatures of these functions are as follows:

$$\begin{array}{l} \mathit{impl} : \mathit{Type} \times \mathit{RoutineId} \to \mathit{RoutineImpl} \cup \{\mathit{undef}\} \\ \mathit{body} : \mathit{RoutineImpl} \to \mathit{Instr} \end{array}$$

The complications of multiple inheritance can be elegantly captured by a revised definition of *impl*. While $\mathit{impl}(T, m)$ traverses T's parent classes, it can take redefinition, undefinition, and renaming into account. In particular, *impl* is undefined for deferred routines (abstract methods) or when an inherited routine has been undefined.

```
class A                              class B
    feature m do ... end                 feature m do ... end
end                                  end

class C
inherit A
        B rename m as n end
end

                                     class E
class D                              inherit C rename m as m2 end
inherit C redefine m end
    feature m do ... end             end
end
```

Fig. 3. Example of Inheritance using Rename and Redefine.

Figure 3 shows an example of inheritance using the features `rename` and `redefine`. Table 1 presents an example of the application of the function *impl* using the class declarations of Figure 3. For brevity, refer to our technical report [11] for a definition of *impl*.

Table 1. Example of the Application of the Function *impl*.

impl(A,m)	= A@m	impl(D,m)	= D@m
impl(B,m)	= B@m	impl(D,n)	= B@m
impl(C,m)	= A@m	impl(E,m)	= undefined
impl(C,n)	= B@m	impl(E,m2)	= A@m
		impl(E,n)	= B@m

The operational semantics of routine invocation for non-once routines is defined with the following rules:

$$\frac{\begin{array}{c} T{:}m \textit{ is not a once routine} \\ \sigma(y) = \textit{voidV} \end{array}}{\langle \sigma, x := y.T{:}m(e)\rangle \to \sigma, \textit{exc}} \tag{1}$$

$$\frac{\begin{array}{c} T{:}m \textit{ is not a once routine} \\ \sigma(y) \neq \textit{voidV} \qquad \langle \sigma[\textit{Current} := \sigma(y), p := \sigma(e)], \textit{body}(\textit{impl}(\tau(\sigma(y)), m))\rangle \to \sigma', \chi \end{array}}{\langle \sigma, x := y.T{:}m(e)\rangle \to \sigma'[x := \sigma'(\textit{Result})], \chi} \tag{2}$$

In rule (1), since the target y is *Void*, the state σ is not changed and an exception is triggered. In rule (2), the target is not Void, thus, the *Current* object is updated with y, and the argument p by the expression e, and then the body of the routine is executed. To handle dynamic dispatch, first, the dynamic type of y is obtained using the function τ. Then, the routine implementation is determined by the function *impl*. Finally, the body of the routine is obtained by the function *body*. Note that multiple inheritance is handled using the function *impl*, which yields the appropriate routine implementation for m.

Exception Handling. Exceptions [7] raise some of the most interesting problems addressed in this paper. A routine execution either succeeds - meaning that it terminates normally - or fails, triggering an exception. An exception is an abnormal event, which occurred during the execution. To treat exceptions, each routine may contain a `rescue` clause. If the routine body is executed and terminates normally, the `rescue` clause is ignored. However, if the routine body triggers an exception, control is transferred to the `rescue` clause. Each routine defines a boolean local variable *Retry* (in a similar way as for *Result*). If at the end of the clause the variable *Retry* has value true, the routine body (`do` clause) is executed again. Otherwise, the routine fails, and triggers an exception. If the `rescue` clause triggers another exception, the second one takes precedence and it can be handled through the `rescue` clause of the caller. The *Retry* variable can be assigned to in either a `do` clause or a `rescue` clause.

The operational semantics for the exception handling mechanism is defined by rules 3-6 below. If the execution of s_1 terminates normally, then the `rescue` block is not executed, and the returned state is the one produced by s_1 (rule 3). If s_1 terminates with an exception, and s_2 triggers another exception, the `rescue` terminates in an exception returning the state produced by s_2 (rule 4). If s_1 triggers an exception and s_2 terminates normally but the *Retry* variable is false, then the `rescue` terminates with an exception returning the state produced by s_2 (rule 5). In the analogous situation with *Retry* being true, the rescue is executed again and the result is the one produced by the new execution of the `rescue` (rule 6). Note that the `rescue` is a loop that iterates over $s_2; s_1$ until s_1 terminates normally or *Retry* is false.

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', \mathit{normal}}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow \sigma', \mathit{normal}} \tag{3}$$

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', \mathit{exc} \qquad \langle \sigma', s_2 \rangle \rightarrow \sigma'', \mathit{exc}}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow \sigma'', \mathit{exc}} \tag{4}$$

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', \mathit{exc} \qquad \langle \sigma', s_2 \rangle \rightarrow \sigma'', \mathit{normal} \qquad \neg\sigma''(\mathit{Retry})}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow \sigma'', \mathit{exc}} \tag{5}$$

$$\frac{\begin{array}{c}\langle \sigma, s_1 \rangle \rightarrow \sigma', \mathit{exc} \qquad \langle \sigma', s_2 \rangle \rightarrow \sigma'', \mathit{normal} \qquad \sigma''(\mathit{Retry}) \\ \langle \sigma'', s_1 \texttt{ rescue } s_2 \rangle \rightarrow \sigma''', \chi\end{array}}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow \sigma''', \chi} \tag{6}$$

Once Routines. The mechanism used in Eiffel to access a shared object is *once routines.* This section focuses on once functions; once procedures are similar. The semantics of once functions is as follows. When a once function is invoked for the first time in a program execution, its body is executed and the outcome is cached. This outcome may be a result in case the body terminates normally or an exception in case the body triggers an exception. For subsequent invocations, the body is not executed; the invocations produce the same outcome (result or exception) like the first invocation. Note that whether an invocation is the first or a subsequent one is determined solely by the routine implementation name.

To be able to develop a semantics for once functions, finally, we also need to consider recursive invocations. As described in the Eiffel ECMA standard [8], a recursive call may start before the first invocation finished. In that case, the recursive call will return the result that has been obtained so far. The mechanism is not so simple. For example the behavior of following recursive factorial function might be surprising:

```
factorial (i: INTEGER): INTEGER
  require i>=0
  once
     if i<=1 then Result := 1
     else
        Result := i
        Result := Result * factorial (i–1)
     end
  end
```

This example is a typical factorial function but it is also a once function, and the assignment $Result := i{*}factorial(i-1)$ is split into two separate assignments. If one invokes $factorial(3)$ we observe that the returned result is 9. The reason is that the first invocation, $factorial(3)$, assigns 3 to $Result$. This result is stored for a later invocation since the function is a once function. Then, the recursive call is invoked with argument 2. But this invocation is not the first invocation, so the second invocation returns the stored value (in this case 3). Thus, the result of invoking $factorial(3)$ is $3 * 3$. If we do not split the assignment, the result would be 0 because $factorial(2)$ would return the result obtained so far which is the default value of $Result$, 0. This corresponds to a semantics where recursive calls are replaced by $Result$.

To be able to develop a sound semantics for once functions, we need to consider all the possible cases described above. To fulfil this goal, we present a pseudo-code of once functions. Given a once function m with body b, the pseudo-code is the following:

```
if not m_done then m_done := true; execute the body b
   if body triggers an exception e then m_exception := e end
end
if m_exception /= Void then throw m_exception else Result := m_result
end
```

We assume the variables m_done, $m\_exception$ and m_result are global variables, which exist one per routine implementation and can be shared by all invocations of that function. Furthermore, we assume the body of the function sets the result variable m_result. Now, we can see more clearly why the invocation of *factorial* (3) returns 9. In the first invocation, first the global variable m_done is set to false, and then the function's body is executed. The second invocation returns the stored value 3 because m_done is false.

To define the semantics for once functions, we introduce global variables to store the information whether the function was invoked before or not, to store whether it triggers an exception or not, and to store its result. These variables are $T@m\_done$, $T@m\_result$, and $T@m\_exc$. There is only one variable $T@m\_done$ per every routine implementation $T@m$. Descendants of the class T that do not redefine the routine m inherit the routine implementation $T@m$, therefore they share the same global variable $T@m\_done$.

Given a once function m implemented in the class T, $T@m\_done$ returns true if the once function was executed before, otherwise it returns false; $T@m\_result$ returns the result of the first invocation of m; and $T@m\_exc$ returns true if the first invocation of m produced an exception, otherwise it returns false. Since the type of the exception is not used in the exception mechanism, we use a boolean variable $T@m\_exc$, instead of a variable of type *EXCEPTION*. We omit the definition of a global initialization phase $T@m\_done = false$, $T@m\_result = default\ value$, and $T@m\_exc = false$. This initialization is performed in the *make* routine of the *ROOT* class.

The invocation of a once function is defined in four rules (rules 7-10, Figure 4). Rule (7) describes the normal execution of the first invocation of a once function. Before its execution, the global variable $T@m\_done$ is set to true. Then, the function body is executed. We assume here that the body updates the variable $T@m\_result$ whenever it assigns to *Result*. Rule (8) models the first invocation of an once function that terminates with an exception. The function is executed and terminates in the state σ'. The result of the once function m is the state σ' where the variable $T@m\_exc$ is set to true to express that an exception was triggered. In rule 9, the first invocation of the once function terminates normally, the remaining invocations restore the stored value using the variable $T@m\_result$. In rule 10, the first invocation of m terminates with an exception, so the subsequent invocations of m trigger an exception, too.

3 A Program Logic for Eiffel

The logic for Eiffel is based on the programming logic presented by Poetzsch-Heffter and Müller [18, 19]. We take over many of the rules, especially all the language-independent rules such as the rules of consequence. We do not repeat those rules here. We have developed new rules to model exceptions and once routines. Poetzsch-Heffter et al. [19] uses a special variable χ to capture the status of the program such as normal or exceptional status. We instead use Hoare triples with two postconditions to encode the status of the program execution.

$$\frac{\begin{array}{c} T@m = \mathit{impl}(\tau(\sigma(y)), m) \quad T@m \ \mathit{is\ a\ once\ routine} \\ \sigma(T@m\_done) = \mathit{false} \\ \langle \sigma[T@m\_done := \mathit{true}, \mathit{Current} := y, p := \sigma(e)],\ \mathit{body}(T@m) \rangle \rightarrow \sigma', \mathit{normal} \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \rightarrow \sigma'[x := \sigma'(\mathit{Result})], \mathit{normal}} \tag{7}$$

$$\frac{\begin{array}{c} T@m = \mathit{impl}(\tau(\sigma(y)), m) \quad T@m \ \mathit{is\ a\ once\ routine} \\ \sigma(T@m\_done) = \mathit{false} \\ \langle \sigma[T@m\_done := \mathit{true}, \mathit{Current} := y, p := \sigma(e)],\ \mathit{body}(T@m) \rangle \rightarrow \sigma', \mathit{exc} \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \rightarrow \sigma'[T@m\_exc := \mathit{true}], \mathit{exc}} \tag{8}$$

$$\frac{\begin{array}{c} T@m = \mathit{impl}(\tau(\sigma(y)), m) \quad T@m \ \mathit{is\ a\ once\ routine} \\ \sigma(T@m\_done) = \mathit{true} \\ \sigma(T@m\_exc) = \mathit{false} \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \rightarrow \sigma[x := \sigma(T@m\_result)], \mathit{normal}} \tag{9}$$

$$\frac{\begin{array}{c} T@m = \mathit{impl}(\tau(\sigma(y)), m) \quad T@m \ \mathit{is\ a\ once\ routine} \\ \sigma(T@m\_done) = \mathit{true} \\ \sigma(T@m\_exc) = \mathit{true} \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \rightarrow \sigma, \mathit{exc}} \tag{10}$$

Fig. 4. Operational Semantics for Once Routines

The logic is a Hoare-style logic. Properties of routines and routine bodies are expressed by Hoare triples of the form $\{P\}\ \ S\ \ \{Q_n\ ,\ Q_e\}$, where P, Q_n, Q_e are formulas in first order logic, and S is a routine or an instruction. The third component of the triple consists of a normal postcondition (Q_n), and an exceptional postcondition (Q_e).

The triple $\{P\}\ \ S\ \ \{Q_n\ ,\ Q_e\}$ defines the following refined partial correctness property: if S's execution starts in a state satisfying P, then (1) S terminates normally in a state where Q_n holds, or (2) S throws an exception and Q_e holds, or (3) S aborts due to errors or actions that are beyond the semantics of the programming language, e.g., memory allocation problems, or (4) S runs forever.

Boolean Expressions. Preconditions and postcondition are formulas in first order logic. Since expressions in assignments can trigger exceptions, we cannot always use these expressions in pre- and postconditions of Hoare triples. For example, if we want to apply the substitution $P[e/x]$ where e is an *ExpE* expression, first, we need to check that e does not trigger any exception, and then we can apply the substitution. To do this, we introduce a function *safe* that takes an expression, and yields a *safe* expression. A safe expression is an expression whose evaluation does not trigger an exception. The definition of safe expression is the following:

Definition 1 (Safe Expression). *An expression e is a* ***safe expression*** *if and only if* $\forall\ \sigma :\ \sigma(e) \neq \mathit{exc}$.

Definition 2 (Function Safe). *The function safe* : *ExpE* $\rightarrow$ *Exp returns an expression that expresses if the input expression is safe or not. The definition of this function is the following:*

$$\begin{array}{l} \mathit{safe} : \mathit{ExpE} \rightarrow \mathit{Exp} \\ \mathit{safe}\ (e_1\ \mathit{oper}\ e_2) = \mathit{safe}(e_1)\ \wedge\ \mathit{safe}(e_2)\ \wedge\ \mathit{safe\_op}\ (\mathit{oper}, e_1, e_2) \\ \\ \mathit{safe\_op} : \mathit{op} \times \mathit{ExpE} \times \mathit{ExpE} \rightarrow \mathit{Exp} \\ \mathit{safe\_op}\ (\mathit{oper},\ e_1,\ e_2) = \textbf{if}\ (\mathit{oper} = //)\ \textbf{then}\ (e_2 \neq 0)\ \textbf{else}\ \mathit{true} \end{array}$$

Lemma 1. *For each expression e, safe(e) satisfies:*

– *safe*(e) *is a* ***safe expression***
– $\sigma(\mathit{safe}(e)) = \mathit{true}\ \Leftrightarrow \sigma(e) \neq \mathit{exc}$

Lemma 2 (Substitution). *If the expression e is a* ***safe expression****, then:*

$$\forall\ \sigma : (\sigma \models P[e/x]\ \Leftrightarrow\ \sigma[x := \sigma(e)]\ \models P)$$

We define $\sigma \models P$ as the usual definition of $\models$ in first order logic but with the restriction that the expressions in P are safe expressions.

Signatures of Contracts. Contracts refer to attributes, variables and types. We introduce a signature Σ that represents the constant symbols of these entities. Given an Eiffel program, Σ denotes the signature of sorts, functions and constant symbols as described in Section 2.1. Arguments, program variables, and the current object store \$ are treated syntactically as constants of *Value* and *ObjectStore*. Preconditions and postconditions of Hoare triples are formulas over $\Sigma\ \cup\ \{\mathit{Current}, p, \mathit{Result}, \mathit{Retry}\}\ \cup\ \mathit{Var}(r)$ where r is a routine, and $\mathit{Var}(r)$ denotes the set of local variables of r. Note that we assume $\mathit{Var}(r)$ does not include the *Result* variable and the *Retry* variable, it only includes the local variables declared by the programmer. Routine preconditions are formulas over $\Sigma\ \cup\ \{\mathit{Current}, p, \$\}$, and routine postconditions are formulas over $\Sigma\ \cup\ \{\mathit{Current}, p, \mathit{Result}, \mathit{Retry}, \$\}$.

We treat recursive routines in the same way as Poetzsch-Heffter and Müller [18]. We use *sequents* of the form $\mathcal{A} \rhd \mathbf{A}$ where $\mathcal{A}$ is a set of routine annotations and $\mathbf{A}$ is a triple. Triples in $\mathcal{A}$ are called assumptions of the sequent, and $\mathbf{A}$ is called the consequent of the sequent. Thus, a sequent expresses that we can prove a triple based on the assumptions about routines.

In the following, we present the logic for Eiffel. First, we present the basic instructions. Second, we define the logic for invocation. Finally, we present the semantics for exception handling and once routines. The logic for exception handling and once routines is another contribution of this paper.

3.1 Basic Rules

Figure 5 presents the axiomatic semantics for basic instructions such as assignment, compound, loop, and conditional instructions. In the assignment rule, if the expression e is safe (it does not throw any exception) then the precondition is obtained replacing x by e in P. Otherwise the precondition is the exceptional postcondition.

In the compound instruction, first the instruction s_1 is executed. If s_1 triggers an exception, s_2 is not executed, and R_e holds. If s_1 terminates normally, s_2 is executed, and the postcondition of the compound is the postcondition of s_2. Note that for conditionals, check instructions, and loops, the expression e is syntactically restricted not to trigger an exception, which simplifies the rules significantly.

Assignment Instruction

$$\frac{}{\rhd \left\{ \begin{array}{l} (safe(e) \ \wedge \ P[e/x]) \ \vee \\ (\neg safe(e) \ \wedge \ Q_e) \end{array} \right\} \ x \ := \ e \ \left\{ P \ , \ Q_e \right\}}$$

Compound

$$\frac{\mathcal{A} \rhd \left\{ P \right\} \ s_1 \ \left\{ Q_n \ , \ R_e \right\} \qquad \mathcal{A} \rhd \left\{ Q_n \right\} \ s_2 \ \left\{ R_n \ , \ R_e \right\}}{\mathcal{A} \rhd \left\{ P \right\} \ s_1 ; s_2 \ \left\{ R_n \ , \ R_e \right\}}$$

Conditional Instruction

$$\frac{\begin{array}{c} \mathcal{A} \rhd \left\{ P \ \wedge \ e \right\} \ s_1 \ \left\{ Q_n \ , \ Q_e \right\} \\ \mathcal{A} \rhd \left\{ P \ \wedge \ \neg e \right\} \ s_2 \ \left\{ Q_n \ , \ Q_e \right\} \end{array}}{\mathcal{A} \rhd \left\{ P \right\} \ \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \ \left\{ Q_n \ , \ Q_e \right\}}$$

Check Instruction

$$\frac{}{\mathcal{A} \rhd \left\{ P \right\} \ \texttt{check } e \texttt{ end} \ \left\{ (P \ \wedge \ e \ ) \ , \ (P \ \wedge \ \neg e \ ) \right\}}$$

Loop Instruction

$$\frac{\mathcal{A} \rhd \left\{ \neg e \ \wedge \ I \ \right\} \ s_1 \ \left\{ I \ , \ R_e \right\}}{\mathcal{A} \rhd \left\{ I \right\} \ \texttt{from invariant } I \texttt{ until } e \texttt{ loop } s_1 \texttt{ end} \ \left\{ (I \ \wedge \ e) \ , \ R_e \right\}}$$

Fig. 5. Axiomatic Semantics for Basic Instructions

3.2 Routine and Routine Invocation Rules

Routine Invocation. Routine invocations of non-once and once routines are verified based on properties of the the virtual method being called:

$$\frac{\mathcal{A} \rhd \left\{ P \right\} \ T{:}m \ \left\{ Q_n \ , \ Q_e \right\}}{\mathcal{A} \rhd \left\{ \begin{array}{l} (y \neq Void \wedge P[y/Current, e/p]) \vee \\ (y = Void \wedge Q_e) \end{array} \right\} \ x := y.T{:}m(e) \ \left\{ Q_n[x/Result] \ , \ Q_e \right\}}$$

In this rule, if the target y must not be *Void*, the current object is replaced by y and the formal parameter p by the expression e in the precondition P.

Then, in the postcondition Q_n, *Result* is replaced by x to assign the result of the invocation. If y is *Void* the invocation triggers and exception, and Q_e holds.

To prove a triple for a virtual method $T : m$, one has to derive the property for all possible implementations, that is, $\mathit{impl}(m, T)$ and $S : m$ for all sublasses S of T. The corresponding rule is identical to the logic we extend [18].

Routine Implementation. The following rule is used to derive properties of routine implementations from their bodies.

$$\frac{\mathcal{A}, \{P\} \quad T@m \quad \{Q_n,\ Q_e\} \rhd \left\{\, P \,\right\} \quad \mathit{body}(T@m) \quad \left\{\, Q_n \ ,\ Q_e \,\right\}}{\mathcal{A} \rhd \left\{\, P \,\right\} \quad T@m \quad \left\{\, Q_n \ ,\ Q_e \,\right\}}$$

Eiffel pre and postconditions are often too weak for verification, for instance because they cannot contain quantifiers. Therefore, our logic allows one to use stronger conditions. To handle recursion, we add the assumption $\{P\} \quad T@m(p) \quad \{Q_n,\ Q_e\}$ to the set of routine annotations $\mathcal{A}$.

3.3 Exception Handling

The operational semantics presented in Section 2.3 shows that a `rescue` clause is a loop. The loop body $s_2; s_1$ iterates until no exception is thrown in s_1, or *Retry* is false. To be able to reason about this loop, we introduce an invariant I_r. We call this invariant *rescue invariant*. The rule is defined as follows:

$$\frac{\begin{array}{c} P \ \Rightarrow\ I_r \\ \mathcal{A} \rhd \left\{\, I_r \,\right\} \quad s_1 \quad \left\{\, Q_n \ ,\ Q_e \,\right\} \\ \mathcal{A} \rhd \left\{\, Q_e \,\right\} \quad s_2 \quad \left\{\, \mathit{Retry} \Rightarrow I_r \ \wedge\ \neg \mathit{Retry} \Rightarrow R_e \ ,\ R_e \,\right\} \end{array}}{\mathcal{A} \rhd \left\{\, P \,\right\} \quad s_1 \ \texttt{rescue} \ s_2 \quad \left\{\, Q_n \ ,\ R_e \,\right\}}$$

This rule is applied to any routine with a `rescue` clause. If the do block, s_1, terminates normally then the `rescue` block is not executed and the postcondition is Q_n . If s_1 triggers an exception, the `rescue` block executes. If the `rescue` block, s_2, terminates normally, and the *Retry* variable is true then control flow transfers back to the beginning of the routine and I_r holds. If s_2 terminates normally and *Retry* is false, the routine triggers an exception and R_e holds. If both s_1 and s_2 trigger an exception, the last one takes precedence, and R_e holds.

3.4 Once Routines

To define the logic for once routines, we use the global variables $T@m\_done$, $T@m\_result$, and $T@m\_exc$, which store if the once routine was executed before or not, the result, and the exception. Let P be the following precondition, where T_M_RES is a logical variable:

$$P \equiv \left\{ \begin{array}{l} (\neg T@m\_done \wedge P')\vee \\ \big(T@m\_done \wedge P'' \wedge T@m\_result = T\_M\_RES \wedge \neg T@m\_exc \big) \ \vee \\ (T@m\_done \wedge P''' \wedge T@m\_exc) \end{array} \right\}$$

and let Q'_n and Q'_e be the following postconditions:

$$Q'_n \equiv \left\{ \begin{array}{l} T@m\_done \ \wedge\ \neg T@m\_exc \ \wedge \\ \big(Q_n \vee (\, P'' \ \wedge\ \mathit{Result} = T\_M\_RES \ \wedge\ T@m\_result = T\_M\_RES\,)\big) \end{array} \right\}$$

$$Q'_e \equiv \left\{ \ T@m\_done \ \wedge\ T@m\_exc \ \wedge\ (Q_e \ \vee P''') \ \right\}$$

The rule for once functions is defined as follows:

$$\frac{\begin{array}{l} \mathcal{A}, \{P\} \ \ T@m\ \{Q'_n,\ Q'_e\} \vartriangleright \\ \big\{\, P'[\mathit{false}/T@m\_done] \wedge\ T@m\_done \,\big\} \ \ \mathit{body}(T@m) \ \ \big\{\, Q_n \ ,\ Q_e \,\big\} \end{array}}{\mathcal{A} \vartriangleright \big\{\, P \,\big\} \ \ T@m \ \ \big\{\, Q'_n \ ,\ Q'_e \,\big\}}$$

In the precondition of the body of $T@m$, $T@m\_done$ is true to model recursive call as illustrated in the example presented in Section 2.3. In the postcondition of the rule, under normal termination, either the function $T@m$ is executed and Q_n holds, or the function is not executed since it was already executed and P'' holds. In both cases, $T@m\_done$ is true and $T@m\_exc$ false. In the case an exception is triggered, $Q_e \vee P'''$ holds.

4 Example

Figure 6 presents an example of the application of the logic. The function *safe_division* implements an integer division which always terminates normally. If the second operand, y, is zero, this function returns the first operand, x; otherwise it returns the integer division $x//y$. This function is implemented in Eiffel using a `rescue` clause. If the division triggers an exception, this exception is handled by the `rescue` block setting z to 1 and retrying.

To verify this example, we first apply the routine implementation rule (Section 3.2). Then, we apply the rescue rule (Section 3.3) to verify the rescue block. The *retry invariant* is $(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0))$. Finally, we verify the body of the do block and the body of the rescue block using the assignment, and the compound rule.

5 Soundness and Completeness

We have proved soundness and completeness of the logic. The soundness proof runs by induction on the structure of the derivation tree for $\big\{\, P \,\big\} \ \ s \ \ \big\{\, Q_n \ ,\ Q_e \,\big\}$. The completeness proof runs by induction on the structure of the instruction s using a sequent which contains a complete specification for every routine implementation $T@m$. In this section, we state the theorems. The proofs are presented in a technical report [11].

```
safe_division (x,y: INTEGER): INTEGER
  local
     z: INTEGER
  do
     { $(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0))$ }
     Result := x // (y+z)
     $\left\{ \begin{pmatrix} (y = 0 \Rightarrow \mathit{Result} = x) \wedge \\ (y \neq 0 \Rightarrow \mathit{Result} = x/y) \end{pmatrix}, (y = 0 \wedge z = 0) \right\}$
  ensure
     zero: y = 0 implies Result = x
     not_zero: y /= 0 implies Result = x // y
  rescue
     { $y = 0 \wedge z = 0$ }
     z := 1
     { $(y = 0 \wedge z = 1)$, false }
     Retry := true
     $\{(\mathit{Retry} \wedge (y = 0 \wedge z = 1)), \mathit{false}\}$
  end
```

Fig. 6. Example of an Eiffel source proof.

Definition 3. *The triple* $\models \{P\} \; s \; \{Q_n \, , \, Q_e\}$ *if and only if:* *for all* $\sigma \models P : \langle \sigma, s \rangle \rightarrow \sigma', \chi$ *then*

- $\chi = \mathit{normal} \Rightarrow \sigma' \models Q_n$, *and*
- $\chi = \mathit{exc} \Rightarrow \sigma' \models Q_e$

Theorem 1 (Soundness Theorem)

$$\rhd \{P\} \; s \; \{Q_n \, , \, Q_e\} \Rightarrow \models \{P\} \; s \; \{Q_n \, , \, Q_e\}$$

Theorem 2 (Completeness Theorem)

$$\models \{P\} \; s \; \{Q_n \, , \, Q_e\} \Rightarrow \rhd \{P\} \; s \; \{Q_n \, , \, Q_e\}$$

6 Related Work

Huisman and Jacobs [4] have developed a Hoare-style logic with abrupt termination. It includes not only exception handling but also while loops which may contain exceptions, breaks, continues, returns and side-effects. The logic is formulated in a general type theoretical language and not in a specific language such as PVS or Isabelle. Oheimb [23] has developed a Hoare-style calculus for a subset of JavaCard. The language includes side-effecting expressions, mutual recursion, dynamic method binding, full exception handling and static class initialization. These logics formalize a Java-like exception handling which is different to the exception handling presented in this paper.

Logics such as separation logic [20, 13], dynamic frames [5, 22], and regions [1] have been proposed to solve a key issue for reasoning about imperative programs: framing. Separation logic has been adapted to verify object-oriented programs [14, 15, 3]. Parkinson and Bierman [14, 15] introduce abstract predicates: a powerful means to abstract from implementation details and to support information hiding and inheritance. Distefano and Parkinson [3] develop a tool to verify Java programs based on the ideas of abstract predicates.

Logics have been also developed for bytecode languages. Bannwart and Müller [2] have developed a Hoare-style logic a bytecode similar to Java Bytecode and CIL. This logic is based on Poetzsch-Heffter and Müller's logic [17, 18], and it supports object-oriented features such as inheritance and dynamic binding. The Mobius project [9] has also developed a program logic for bytecodes. This logic has been proved sound with respect the operational semantics, and it has been formalized in Coq.

With the goal of verifying bytecode programs, Pavlova [16] has developed an operational semantics, and a verification condition generator (VC) for Java Bytecode. Furthermore, she has shown the equivalence between the verification condition generated from the source program and the one generated from the bytecode. Furthermore, Müller and Nordio [10] present a logic for Java and its proof-transformation for programs with abrupt termination. The language considered includes instructions such as `while`, `try-catch`, `try-finally`, `throw`, and `break`.

An operational semantics and a verification methodology for Eiffel has been presented by Schöller [21]. The methodology uses dynamic frame contracts to be able to address the frame problem, and applies to a substantial subset of Eiffel. However, Schöller's work only presents an operational semantics, and it does not include exceptions.

Our logic is based on Poetzsch-Heffter and Müller's work [17, 18], which we extended by new rules for Eiffel instructions. The new rules support Eiffel's exception handling, once routines, and multiple inheritance. This work is based on our earlier effort [12] on proof-transforming compilation from Eiffel to CIL. In this earlier work, we have developed an axiomatic semantics for the exception handling mechanism, and its proof transformation to CIL. This earlier work does not present the operational semantics, and the logic was neither proved sound nor complete. Furthermore, once routines and multiple inheritance were not covered.

7 Lessons Learned

We have presented a sound and complete logic for a subset of Eiffel. Here we report on some lessons on programming language design learned in the process.

Exception Handling. During the development of this work, we have formalized the current Eiffel exception handling mechanism. In the current version of Eiffel, `retry` is an instruction that can only be used in a `rescue` block. When `retry` is executed, the control flow is transferred to the beginning of the routine. If the execution of the `rescue` block finishes without invoking a `retry`, an exception is triggered. Developing a logic for the current Eiffel would require the addition of a third postcondition, to model the execution of `retry` (since `retry` is another way of transferring control flow). Thus, we would use Hoare triples of the form $\left\{ P \right\} \quad s \quad \left\{ Q_n \ , \ Q_r, \ Q_e \right\}$ where s is an instruction, Q_n is the postcondition under normal termination, Q_r the postcondition after the execution of a `retry`, and Q_e the exceptional postcondition.

Such a formalization would make verification harder than with the formalization we use in this paper, because the extra postcondition required by the `retry` instruction would have to be carried throughout the whole reasoning. In this paper, we have observed that a `rescue` block behaves as a loop that iterates until no exception is triggered, and that `retry` can be modeled simply as a variable which guards the loop. Since the `retry` instruction transfers control flow to the beginning of the routine, a `retry` instruction has a similar behavior to a `continue` in Java or C#. Our proposed change of the `retry` instruction to a variable will be introduced in the next revision of the language standard [8].

Since Eiffel does not have `return` instructions, nor `continue`, nor `break` instructions, Eiffel programs can be verified using Hoare triples with only two postconditions. To model object-oriented programs with abrupt termination in languages such as Java or C#, one needs to introduce extra postconditions for `return`, `break` or `continue` (or we could introduce a variable to model abrupt termination). If we wanted to model the current version of Java, for example, we would also need to add postconditions for labelled breaks and labelled continues. Thus, one would need to add as many postcondition as there are labels in the program. These features for abrupt termination make the logic more complex and harder to use.

Another difference between Eiffel and Java and C# is that Eiffel supports exceptions using `rescue` clauses, and Java and C# using `try-catch` and `try-finally` instructions. The use of `try-finally` makes the logic harder as pointed out by Müller and Nordio [10]. The combination of `try-finally` and `break` instructions makes the rules more complex and harder to apply because one has to consider all possible cases in which the instructions can terminate (normal, break, return, exception, etc).

However, we cannot conclude that the Eiffel's exception handling mechanism is always simpler for verification; although it eliminates the problems produced by `try-finally`, `break`, and `return` instructions. Since the rescue block is a loop, one needs a retry invariant. When the program is simple, and it does not trigger many different exceptions, defining this *retry invariant* is simple. But, if the program triggers different kinds of exception at different locations, finding this invariant can be more complicated. Note that finding this *retry invariant* is more complicated than finding a loop invariant since in a loop invariant one has to consider only normal termination (and in Java and C#, also `continue` instructions), but in *retry invariants* one needs to consider all possible executions and all possible exceptions.

Multiple Inheritance. Introducing multiple inheritance to a programing language is not an easy task. The type system has to be extended, and this extension is complex. However, since the resolution of a routine name can be done syntactically, extending Poetzsch-Heffter and Müller's logic [18] to handle multiple inheritance was not a complicated task. The logic was easily extended by giving a new definition of the function *impl*. This function returns the body of a routine by searching the definition in the parent classes, and considering the clauses redefine, undefine, and rename. The experience with this paper indicates that the complexity of a logic for multiple inheritance is similar to a logic for single inheritance.

Once Routines. To verify once routines, we introduce global variables to express whether the once routine has been executed before or not, and whether the routine triggered an exception or not. With the current mechanism, the use of recursion in once functions does not increase the expressivity of the language. In fact, every recursive call can be equivalently replaced by *Result*. However, the rule for once functions is more complicated than it could be if recursion were omitted.

Recursive once function would be more interesting if we changed the semantics of once routines. Instead of setting the global variable *done* before the execution of the body of the once function, we could set it after the invocation. Then the recursive once function would be invoked until the last recursive call finishes. Thus, for example, the result of the first invocation of $\mathit{factorial}(n)$ would be $n!$ (the function *factorial* is presented in Section 2.3). Later invocations of *factorial* would return the stored result. However, this change would not simplify the logic, and we would need to use global variables to mark whether the once function was invoked before or not.

Analyzing the EiffelBase libraries, and the source code of the EiffelStudio compiler, we found that the predominant use of once functions is without arguments, which makes sense because arguments of subsequent calls are meaningless. Even though our rules for once functions are not overly complicated, verification of once functions is cumbersome because one has to carry around the outcome of the first invocation in proofs. It is unclear whether this is any simpler than reasoning about static methods and fields [6].

References

1. A. Banerjee, D. Naumann, and S. Rosenberg. Regional Logic for Local Reasoning about Global Invariants. In *ECOOP*, volume 5142 of *LNCS*, pages 387–411. Springer-Verlag, 2008.
2. F. Y. Bannwart and P. Müller. A Logic for Bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141(1) of *ENTCS*, pages 255–273. Elsevier, 2005.
3. D. Distefano and M. J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 213–226, 2008.

4. M. Huisman and B. Jacobs. Java program verification via a hoare logic with abrupt termination. In E. Maibaum, editor, *Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
5. I. T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*, pages 268–283, 2006.
6. K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag, 2005.
7. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
8. B. Meyer (editor). ISO/ECMA Eiffel standard (Standard ECMA-367: Eiffel: Analysis, Design and Programming Language), June 2006. available at http://www.ecma-international.org/publications/standards/Ecma-367.htm.
9. MOBIUS Consortium. Deliverable 3.1: Byte code level specification language and program logic. Available online from http://mobius.inria.fr, 2006.
10. P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, 2007.
11. M. Nordio, C. Calcagno, P. Müller, and B. Meyer. Soundness and Completeness of a Program Logic for Eiffel. Technical Report 617, ETH Zurich, 2009.
12. M. Nordio, P. Müller, and B. Meyer. Proof-Transforming Compilation of Eiffel Programs. In R. Paige and B. Meyer, editors, *TOOLS-EUROPE*, Lecture Notes in Business and Information Processing. Springer-Verlag, 2008.
13. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, pages 268–280, 2004.
14. M. J. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL '05*, volume 40, pages 247–258. ACM, 2005.
15. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75–86. ACM, 2008.
16. M. Pavlova. *Java Bytecode verification and its applications*. PhD thesis, University of Nice Sophia-Antipolis, 2007.
17. A. Poetzsch-Heffter and P. Müller. Logical Foundations for Typed Object-Oriented Languages . In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, pages 404–423, 1998.
18. A. Poetzsch-Heffter and P. Müller. A Programming Logic for Sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.
19. A. Poetzsch-Heffter and N. Rauch. Soundness and Relative Completeness of a Programming Logic for a Sequential Java Subset. Technical report, Technische Universität Kaiserlautern, 2004.
20. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
21. B. Schoeller. *Making classes provable through contracts, models and frames*. PhD thesis, ETH Zurich, 2007.
22. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *Formal Techniques for Java-like Programs*, 2008.
23. D. von Oheimb. *Analyzing Java in Isabelle/HOL - Formalization, Type Safety and Hoare Logic -*. PhD thesis, Universität München, 2001.