Soundness and Completeness of a 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

(3) Imperial College, London, UK
ccris@doc.ic.ac.uk

ETH Technical Report 617

March 2009

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.

Contents

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 the 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 [21]. This logic includes the most important features of object-oriented languages such as abstract types, dynamic binding, subtyping, and inheritance. However, the exception handling is not treated in their work. Huisman and Jacobs [6] developed a Hoare-stype logic which treats abrupt termination. It includes not only exception handling but also break, continue, and return statements. This logic has been 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. 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, and Section 7 summarizes the result and describes future developments. Appendix A presents the soundness and completeness proofs.

2 A Semantics for Eiffel

2.1 The Source Language

The source language is a subset of Eiffel which includes the most important Eiffel's features, although agents are omitted. The most interesting concepts supported by this subset are: (1) multiple inheritance, (2) exception handling, and (3) once routines. Multiple inheritance is supported using the clauses undefine, redefine and rename. The exception handling mechanism is developed using rescue clauses. Instructions can throw exceptions either in the routine body or the rescue clause,.

An Eiffel program is a sequence of class declaration. A class declaration consist of an optional inheritance clause, and a class body. The inheritance clause supports multiple inheritance and allow us to undefine, redefine and rename routines. If the routine is redefined, preconditions of subclasses can be weaken, and postconditions can be stronger. A class body is a sequence of attributes declaration or routine declaration. For simplicity, routines are functions that take always one argument and return a result. Routines are once routine or non-once routines (normal routines). Once routines are routines that always return the same result after their first execution.

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 functions $*$ and $+$ are defined as usual, and *list_of* denotes a comma-separated list.

Figure **??** presents the syntax of expressions. Boolean expression and expressions (*boolExp* and *exp*) are side-effect-free and do not trigger exceptions. Furthermore, *expE* denotes expressions that are side-effect-free but can trigger exceptions. For simplicity, expressions that can trigger exceptions (*expE*) are only allowed in assignments. This assumption simplifies the presentation of the logic, specially the rules for routine invocation, read and write attribute and `if then else` and `loop` instructions. However, the logic could 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 contracts are side-effect-free and do not throw exceptions.

```
Program            ::= ClassDeclaration*
ClassDeclaration   ::= class ClassId [Inheritance] ClassBody end
Type               ::= BoolT | IntT | ClassId | VoidT
Inheritance        ::= inherit Parent+
Parent             ::= Type [Undefine] [Redefine] [Rename]
Undefine           ::= undefine list_of RoutineId
Redefine           ::= redefine list_of RoutineId
Rename             ::= rename list_of (RoutineId as RoutineId)
ClassBody          ::= MemberDeclaration*
MemberDeclaration  ::= 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
                    |  from invariant BoolExp until BoolExp loop Instr end
                    |  if BoolExp then Instr else Instr end
                    |  check BoolExp end
                    |  VarId := create {Type}.make (Exp)
                    |  VarId := VarId.Type@AttributeId
                    |  VarId.Type@AttributeId := Exp
                    |  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             ::= < | > | <= | >= | = | /=
```

Figure 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 object store $. A value is a boolean, an integer, 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, and it is defined as follows:

$$\begin{aligned}\textbf{datatype } \mathit{Value} \;&= \textbf{boolV } \mathit{Bool}\\ &\mid \textbf{intV } \mathit{Int}\\ &\mid \textbf{objV } \mathit{ClassId}\ \mathit{ObjId}\\ &\mid \textbf{voidV}\end{aligned}$$

The function τ returns the dynamic type of a value. Its definition is the following:

$$\begin{aligned}&\tau : \mathit{Value} \to \mathit{Type}\\ &\quad \tau(\textbf{boolV}\ b) &&= \mathit{BoolT}\\ &\quad \tau(\textbf{intV}\ n) &&= \mathit{IntT}\\ &\quad \tau(\textbf{objV}\ cId\ oId) &&= cId\\ &\quad \tau(\textbf{voidV}) &&= \mathit{VoidT}\end{aligned}$$

The function *init* initializes default values to types. The default value of boolean is *false*, the default value of integer is zero, and the default value of reference objects is *voidV*. Its definition is as follows:

$$\begin{aligned}&\mathit{init} : \mathit{Type} \to \mathit{Value}\\ &\quad \mathit{init}(\mathit{BoolT}) &&= (\textbf{boolV}\ \mathit{false})\\ &\quad \mathit{init}(\mathit{IntT}) &&= (\textbf{intV}\ 0)\\ &\quad \mathit{init}(cId) &&= \textbf{voidV}\\ &\quad \mathit{init}(\mathit{VoidT}) &&= \textbf{voidV}\end{aligned}$$

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.

$$\textbf{datatype } \mathit{Attribute} \;= \mathit{Type}\ \mathit{AttributeId}$$

We use a sort *Location* and a function *instvar* where $\mathit{instvar}(V, T@a)$ returns the instance of the attribute $T@a$ if V is an object reference and the object has an attribute $T@a$; otherwise it returns *undef*. The datatype definition and the signature of *instvar* are the following:

$$\textbf{datatype } \mathit{Location} \;= \mathit{ObjId}\ \mathit{AttributeId}$$

$$\mathit{instvar} : \mathit{Value} \times \mathit{Attribute} \to \mathit{Location} \cup \{\mathit{undef}\}$$

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 [19]. The following operations apply to the object store: $\$(f)$ denotes reading the location l in store $\$$; $\mathit{alive}(o, os)$ yields true if and only if object o is allocated in os; $\mathit{new}(os, C)$ yields a reference to a new object in the store os of type C; $\mathit{alloc}(os, C)$ denotes the store after allocating the object store $\mathit{new}(os, C)$; $\mathit{update}(os, l, v)$ updates the object store os at the location l with the value v:

$$\begin{aligned}\_(\_) \;&:\; \mathit{ObjectStore} \times \mathit{Location} \to \mathit{Value}\\ \mathit{alive} \;&:\; \mathit{Value} \to \mathit{ObjectStore} \to \mathit{Bool}\\ \mathit{new} \;&:\; \mathit{ObjectStore} \times \mathit{ClassId} \to \mathit{Value}\\ \_<\_:=\_> \;&:\; \mathit{ObjectStore} \times \mathit{Location} \times \mathit{Value} \to \mathit{ObjectStore}\\ \_<\_> \;&:\; \mathit{ObjectStore} \times \mathit{ClassId} \to \mathit{ObjectStore}\end{aligned}$$

Following we present Poetzsch-Heffter's axiomatization [19] of these functions with a brief description. The function $\mathit{obj} : \mathit{Location} \to \mathit{Value}$ takes a location and yields its value. The function $\mathit{ltyp} : \mathit{Location} \to \mathit{Type}$ yields the dynamic type of a location.

Axiom 1 *Updating a location does not affect the values of other locations:*
$\forall\ OS \in ObjectStore,\ L_1, L_2 \in Location,\ X \in Value : L_1 \neq L_2\ \Rightarrow OS < L_1 := X > (L_2) = OS(L_2)$

Axiom 2 *Reading a location updated with a value produces the same value if both the location and the value are alive:*
$\forall\ OS \in ObjectStore,\ L \in Location,\ X \in Value :$
$alive(obj(L), OS)\ \wedge\ alive(X, OS)\ \Rightarrow\ OS < L := X > (L) = X$

Axiom 3 *Reading a location that is not alive produces the default value of the type of the location:*
$\forall\ OS \in ObjectStore,\ L \in Location : \neg alive(obj(L), OS)\ \Rightarrow OS(L) = init(ltyp(L))$

Axiom 4 *Updating a location that is not alive does not modify the object store:*
$\forall\ OS \in ObjectStore,\ L \in Location,\ X \in Value :\ \neg alive(X, OS)\ \Rightarrow\ OS < L := X >= E$

Axiom 5 *Allocating a type in the object store does not change their values:*
$\forall\ OS \in ObjectStore,\ L \in Location,\ cId \in ClassId :\ OS < cId > (L) = OS(L)$

Axiom 6 *Updating a location does not affect the aliveness property:*
$\forall\ OS \in ObjectStore,\ L \in Location,\ X, Y \in Value :\ alive(X, OS < L := Y >) \Leftrightarrow alive(X, OS)$

Axiom 7 *An object is alive if only if the object was alive before or the object is a new object:*
$\forall\ OS \in ObjectStore,\ X \in Value,\ cId \in ClassId :$
$alive(X, OS < cId >) \Leftrightarrow alive(X, OS)\ \vee\ X = new(OS, cId)$

Axiom 8 *Objects held by locations are alive:*
$\forall\ OS \in ObjectStore,\ L \in Location :\ alive(OS(L), OS)$

Axiom 9 *A created object is not alive in the object store from which it was created:*
$\forall\ OS \in ObjectStore,\ cId \in ClassId :\ \neg alive(new(OS, cId), OS)$

Axiom 10 *The dynamic type of a creation object of class id cId is cId:*
$\forall\ OS \in ObjectStore,\ cId \in ClassId :\ \tau(new(OS, cId)) = cId$

Axiom 11 *Two object store are equal if we cannot distinguish them by the alive and the reading location functions:*
$\forall\ OS_1, OS_2 \in ObjectStore,\ L \in Location,\ X \in Value :$
$(\forall X : alive(X, OS_1) \Leftrightarrow alive(X, OS_2))\ \wedge\ (\forall L : OS_1(L) = OS_2(L)) \Rightarrow\ OS_1 = OS_2$

2.3 Operational Semantics

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

$$\begin{array}{lcl} State & \equiv & Local\ \times\ Heap \\ Local & \equiv & VarId \cup \{Current, p, Result, Retry\} \rightarrow Value \cup \{Undef\} \\ Heap & \equiv & \{\$\} \rightarrow ObjectStore \cup \{Undef\} \end{array}$$

The current object store is denoted by \$. *Local* maps local variables, *Current*, 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 $\sigma \in State$, $\sigma(e)$ denotes the evaluation of the expression e in the state σ. Its signature is the following:

$$\sigma : Local \rightarrow Value \cup \{exc\}$$

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

The operation semantics rules have the following form:

$$\langle \sigma, S \rangle \rightarrow \sigma', \chi$$

where σ and σ' are states, S instructions 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 $\sigma, S \ \rightarrow \sigma', normal$ expresses that executing the instruction S in the state σ terminates normally in the state σ'. The transition $\sigma, S \ \rightarrow \sigma', exc$ expresses that executing the instruction S in the state σ terminales with an exception in the state σ'.

Following, we present the operational semantics. Subsection 2.3.1 presents the basic instructions, which are an adaptation of Müller and Poetzsch-Heffter's work [12, 20, 21] to Eiffel. Subsection 2.3.2 presents routine invocation and object creation. Subsections 2.3.3 and 2.3.4 presents exception handling and once routines. The operation semantics for exception handling and once routines is one of the contributions of this paper.

2.3.1 Basic Instructions

Figure 2 presents the operational semantics for basic instructions such as assingment and compound. Following, we describe this semantics:

Assignment Instruction. The semantics for assignments consists of two rules: one when the expression e throws an exception and one when it does not. In rule (2.1), if the expression e throws an exception, then the assignment terminates with an exception and the state is unchanged. The state does not change since expressions are side-effect free. In rule (2.2), if e does not throw any exception, after the execution of the assignment instruction, the variable x is updated with the value of the expression e in the state σ.

Compound. Compound is defined with two rules: in (2.3) the instruction s_1 is executed and an exception is triggered. Then, the instruction s_2 is not executed, and the state of the compound is the state produced by s_1. In (2.4), s_1 is executed an terminates normally. The state of the compound is the state produced by s_2.

Conditional Instruction. In rule (2.5) the resulting state is the produced by the execution of s_1 since e evaluates to true. In rule (2.6) the state produced by the execution of the conditional is the one produced by s_2 due to the evaluation of e yields false.

Check Instruction. The check instruction helps to express a property that you believe will be satisfied. If the property is satisfied then the system does not change. If the property is not satisfied then an exception is thrown. The semantics for check consist of two rules: if the condition of the check instruction evaluates to true, then the instruction terminates normally, rule (2.7); otherwise the check instruction triggers an exception, rule (2.8).

Loop Instruction. The operational semantics for the loop instruction is divided into four rules. In rule (2.9), s_1 triggers an exception, the loop is not executed and the state is the state produced by s_1. In rule (2.10) the body of the `from` terminates normally, and since the condition is *true*, then the body of the loop is not executed producing the state σ'. If the until expression is *false*, in rule (2.11), the instruction s_2 is executed, but it triggers and exception. Thus, the state of the loop is σ'' and in the status is *exc*. Finally, in (2.12), s_2 terminates normally and the condition is evaluated to false, the returned state is the one produced by the new execution of the loop.

Read Attribute Instruction. The semantics of read attribute is defined by two rules depending if the target object is void or not. In rule (2.13), if the value of y is not *Void*, x is updated with the value of the attribute a. In (2.14), if y is *Void*, the instruction terminates with an exception.

Write Attribute Instruction. Similar to read attribute, the semantics for write attribute is defined with two rules. Let $y.T@a := e$ denotes the updating of the attribute a of the object y with the value e. In (2.15) the attribute a of the object y is updated with the value e if y is not void. If y is *Void*, the instruction terminates with an exception (2.16).

2.3.2 Creation, Routines, Routine Bodies and Routine Invocations

Poetzsch-Heffter and Müller [21] 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 invocation is decorated with the virtual method being invoked. The semantics of routine invocations uses two functions: *body* and *impl*. The function $impl(T, m)$ 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:

$$\begin{aligned} &body : RoutineDeclId \rightarrow Instruction \\ &impl : Type \times RoutineId \rightarrow RoutineDeclId \cup \{undef\} \end{aligned}$$

The complications of multiple inheritance can be elegantly captured by a revised definition of *impl*. While $impl(T, m)$ 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.

Figure 3 shows an example of inheritance using the features `rename` and `redefine`. Table 1 presents an example of the application of the function *imp* using the class declarations of Figure 3. Note that if an object o of declared type C is attached to an object of type E, then the invocation $o.m$ would produce a catcall. This problem is detected using our logic since *imp* yields *undefined*. The definition of this function is presented in Appendix B.

Table 1: Example of the Application of the Function *imp*.

imp(A,m)	= A@m
imp(B,m)	= B@m
imp(C,m)	= A@m
imp(C,n)	= B@m

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

Following, we present the operational semantics for creation, routine invocation and local variables declaration. This semantics is an adaptation of Müller's work [12].

Given a routine declaration *rId* (contracts omitted):

$$\begin{array}{l} rId\ (x : T) :\ T' \\ \quad \texttt{local} \\ \quad\quad v_1 : T_1;\ \dots\ v_n : T_n \\ \quad \texttt{do} \\ \quad\quad s \\ \quad \texttt{end} \end{array}$$

the function *body* returns the following result:

$$\texttt{local}\ \ v_1 : T_1;\ \dots\ v_n : T_n; Result : T'; Retry : Boolean;\ s$$

Assignment Instruction

$$\frac{\sigma(e) = exc}{\langle \sigma, x := e \rangle \rightarrow \sigma, exc} \quad (2.1)$$

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

Compound

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', exc}{\langle \sigma, s_1; s_2 \rangle \rightarrow \sigma', exc} \quad (2.3)$$

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

Conditional Instruction

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', \chi \qquad \sigma(e) = True}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \rangle \rightarrow \sigma', \chi} \quad (2.5)$$

$$\frac{\langle \sigma, s_2 \rangle \rightarrow \sigma', \chi \qquad \sigma(e) = False}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \rangle \rightarrow \sigma', \chi} \quad (2.6)$$

Check Instruction

$$\frac{\sigma(e) = True}{\langle \sigma, \texttt{check}\ e\ \texttt{end} \rangle \rightarrow \sigma, normal} \quad (2.7)$$

$$\frac{\sigma(e) = False}{\langle \sigma, \texttt{check}\ e\ \texttt{end} \rangle \rightarrow \sigma, exc} \quad (2.8)$$

Loop Instruction

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', exc}{\langle \sigma, \texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end} \rangle \rightarrow \sigma', exc} \quad (2.9)$$

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', normal \qquad \sigma'(e) = True}{\langle \sigma, \texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end} \rangle \rightarrow \sigma', normal} \quad (2.10)$$

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', normal \qquad \sigma'(e) = False \qquad \langle \sigma', s_2 \rangle \rightarrow \sigma'', exc}{\langle \sigma, \texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end} \rangle \rightarrow \sigma'', exc} \quad (2.11)$$

$$\frac{\begin{array}{c}\langle \sigma, s_1 \rangle \rightarrow \sigma', normal \qquad \sigma'(e) = False \qquad \langle \sigma', s_2 \rangle \rightarrow \sigma'', normal \\ \langle \sigma'', \texttt{from}\ skip\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end} \rangle \rightarrow \sigma''', \chi\end{array}}{\langle \sigma, \texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end} \rangle \rightarrow \sigma''', \chi} \quad (2.12)$$

Read Attribute Instruction

$$\frac{\sigma(y) \neq voidV}{\langle \sigma, x := y.T@a \rangle \rightarrow \sigma[x := \sigma(\$)\ (instvar(\sigma(y), T@a))], normal} \quad (2.13)$$

$$\frac{\sigma(y) = voidV}{\langle \sigma, x := y.T@a \rangle \rightarrow \sigma, exc} \quad (2.14)$$

Write Attribute Instruction

$$\frac{\sigma(y) \neq voidV}{\langle \sigma, y.T@a := e \rangle \rightarrow \sigma[\$:= \sigma(\$) < instvar(\sigma(y), T@a) := \sigma(e) >], normal} \quad (2.15)$$

$$\frac{\sigma(y) = voidV}{\langle \sigma, y.T@a := e \rangle \rightarrow \sigma, exc} \quad (2.16)$$

Figure 2: Operational Semantics for Basic Instructions

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

Figure 3: Example of Inheritance using Rename and Redefine.

Note the function *body* adds the declaration of the variables *Result* and *Retry*. This declaration allows us to prove properties about the *Result*. In particular, it allows us to initialize the variables *Result* and *Retry* with their initial values.

Local Variables Declaration

Local variables are initialized using rule (1). The values of the variables $v_1 ... v_n$ are updated with their default value according to their types. The function *init*, given a type T returns its default value; $init(INTEGER)$ returns 0; $init(BOOLEAN)$ returns *false* and $init(T)$ where T is a reference type returns *Void*. The rule is the following:

$$\frac{\langle \sigma[v_1 := init(T_1), ..., v_n := init(T_n)],\ s\rangle \rightarrow \sigma', normal}{\langle \sigma, \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\rangle \rightarrow \sigma', normal} \tag{1}$$

Routine Invocation

In this section, we define the operational semantics of routine invocation for non-once routines. Once routine invocations are defined in Section 2.3.4. The semantics for routine invocation is defined as follows:

$$\frac{\begin{array}{c} T{:}m\ \ is\ \ not\ \ a\ \ once\ \ routine \\ \sigma(y) = voidV \end{array}}{\langle \sigma, x := y.T{:}m(e)\rangle \rightarrow \sigma, exc} \tag{2}$$

$$\frac{\begin{array}{c} T{:}m\ \ is\ \ not\ \ a\ \ once\ \ routine \\ \sigma(y) \neq voidV \qquad \langle \sigma[Current := \sigma(y), p := \sigma(e)],\ body(impl(\tau(\sigma(y)), m))\rangle \rightarrow \sigma', \chi \end{array}}{\langle \sigma, x := y.T{:}m(e)\rangle \rightarrow \sigma'[x := \sigma'(Result)], \chi} \tag{3}$$

If the target y is *Void*, then the state σ is not change and an exception is triggered (2). Otherwise, the *Current* object is updated with y, and the argument p by the expression e, and then the body of the routine is executed (3). To handle dynamic dispatch, first, the dynamic type of y is obtained using the function τ. Then, the routine declaration is returned by the function *impl*. Finally, the body of the routine is returned by the function *body*.

Creation Instruction

In the creation instruction, first, a new object of type T is created and assigned to *Current*. The current object store \$ and the argument p are updated in the state σ. Then, the routine *make* is invoked. Finally, the object x is updated with the *Current* object in σ'. The semantics is defined as follows:

$$\frac{\langle \sigma[Current := new(\sigma(\$), T), \$:= \sigma(\$) < T >, p := \sigma(e)],\ body(impl(T, make))\rangle \rightarrow \sigma', \chi}{\langle \sigma, x := \texttt{create}\ T.make(e)\rangle \rightarrow \sigma'[x := \sigma'(Current)], \chi} \quad (4)$$

In Eiffel, when a new object is created, its attributes are initialized with the default value. In the semantics, this is done by the function *new* which creates the new object initializing its attributes.

2.3.3 Exception Handling

Exceptions [9] raise some of the most interesting problems in this paper. A routine execution either succeeds - meaning terminates normally - or fails, triggering an exception. An exception is an abnormal event occurred during the execution. To treat exceptions, each routine contains one `rescue` clause (either explicit or default). If the routine body is executed an terminates normally, the `rescue` clause is ignored. However, if the routine body triggers an exception, control is transfer to the `rescue` clause. Each routine defines a boolean local variable *Retry* (in a similar way as *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 triggering 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.

This specification slightly departs from the current Eiffel standard, where *Retry* is an instruction, not a variable. The change was suggested by this work and will be adopted by a future revision of the language standard. The *Retry* variable can be assigned in either a `do` clause or a `rescue` clause; if its value is true at the end of the `rescue` clause the routine re-executes its body, otherwise it fails, triggering a new exception.

The operation semantics for the exception mechanism is defined by rules 5-8. 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 5). 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 6). 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 7). In a similar situation but when the *Retry* variable is true, the rescue is executed again and the result is the one produced by the new execution of the `rescue` (rule 8).

$$\frac{\langle \sigma, s_1\rangle \rightarrow \sigma', normal}{\langle \sigma, s_1\ \texttt{rescue}\ s_2\rangle \rightarrow \sigma', normal} \quad (5)$$

$$\frac{\langle \sigma, s_1\rangle \rightarrow \sigma', exc \qquad \langle \sigma', s_2\rangle \rightarrow \sigma'', exc}{\langle \sigma, s_1\ \texttt{rescue}\ s_2\rangle \rightarrow \sigma'', exc} \quad (6)$$

$$\frac{\langle \sigma, s_1\rangle \rightarrow \sigma', exc \qquad \langle \sigma', s_2\rangle \rightarrow \sigma'', normal \qquad \neg\sigma''(Retry)}{\langle \sigma, s_1\ \texttt{rescue}\ s_2\rangle \rightarrow \sigma'', exc} \quad (7)$$

$$\frac{\langle \sigma, s_1\rangle \rightarrow \sigma', exc \qquad \langle \sigma', s_2\rangle \rightarrow \sigma'', normal \qquad \sigma''(Retry) \qquad \langle \sigma'', s_1\ \texttt{rescue}\ s_2\rangle \rightarrow \sigma''', \chi}{\langle \sigma, s_1\ \texttt{rescue}\ s_2\rangle \rightarrow \sigma''', \chi} \quad (8)$$

2.3.4 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 function name, irrespective of its arguments.

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 [10], 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 function 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 $T@m\_exc$. Given a once function $T@m$ defined 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 9-12, Figure 4). Rule (9) describes the normal execution of the first invocation of a once function. Before its execution, the

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \ is \ a \ once \ routine \\ \sigma(T@m\_done) = false \\ \langle \sigma[T@m\_done := true, Current := y, p := \sigma(e)],\ body(T@m)\rangle \rightarrow \sigma', normal \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow \sigma'[x := \sigma'(Result)], normal} \tag{9}$$

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \ is \ a \ once \ routine \\ \sigma(T@m\_done) = false \\ \langle \sigma[T@m\_done := true, Current := y, p := \sigma(e)],\ body(T@m)\rangle \rightarrow \sigma', exc \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow \sigma'[T@m\_exc := true], exc} \tag{10}$$

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \ is \ a \ once \ routine \\ \sigma(T@m\_done) = true \\ \sigma(T@m\_exc) = false \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow \sigma[x := \sigma(T@m\_result)], normal} \tag{11}$$

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \ is \ a \ once \ routine \\ \sigma(T@m\_done) = true \\ \sigma(T@m\_exc) = true \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow \sigma, exc} \tag{12}$$

Figure 4: Operational Semantics for Once Routines

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 (10) 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 11, the first invocation of the once function terminates normally, the remaining invocations restore the stored value using the variable $T@m\_result$. In rule 12, 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 developed by Müller and Poetzsch-Heffter [12, 20, 21, 22]. We have added new rules to model exceptions and once routines. Poetzsch-Heffter et al. [22] 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.

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). We call such a triple *routine specification*.

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 S throws an exception and Q_e holds, or (2) S aborts due to errors or actions that are beyond the semantics of the programming language, e.g., memory allocation problems, or (3) 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* $\forall\ \sigma : \sigma(e) \neq exc$.

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

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

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

- *$safe(e)$ is a* ***safe expression***
- $\sigma(safe(e)) = true\ \Leftrightarrow \sigma(e) \neq exc$

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. The introduce a signature Σ that represent 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 are formulas over $\Sigma\ \cup\ \{Current, p, Result, Retry\}\ \cup\ Var(r)$ where r is a routine and $Var(r)$ denotes the set of local variables of r. Note we assume $Var(r)$ does not denote the result variable and the retry variable, it only denotes the local variables declared by the programmer. Preconditions are formulas over $\Sigma \cup \{Current, p, \$\}$, and postconditions are formulas over $\Sigma\ \cup\ \{Current, p, Result, Retry, \$\}$.

We treat recursive routines in the same way as Müller and Poetzsch-Heffter [21]. 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. We assume the sequent $\mathcal{A}$ is constructed before applying the logic. The sequents keep unchanged when the rules are applied. The sequent is constructed with the precondition and postcondition for verification. The user pre and postconditions cannot be used because they are too weak (for example, for the lack of quantifiers). However, we do not discard the user's pre and postcondition. We show that the user's contracts implies the contract used for verifying the program. We denote $pre(T@m)$ and $post(T@m)$ the user's pre and postconditions of the routine m.

Following, we present the logic for Eiffel. Subsection 3.1 presents the basic instructions, which are an adaptation of Müller and Poetzsch-Heffter's work [12, 20, 21] to Eiffel. Subsection 3.2

presents routine invocation and object creation. Subsections 3.3 and 3.4 presents exception handling and once routines. The logic for exception handling and once routines is another contribution of this paper.

3.1 Base Rules

Figure 5 presents the axiomatic semantics for basic instructions such as compound, loop and conditional et al. Following, we describe these rules.

Assignment Instruction. 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. The function *safe* is used to ensure that the precondition of the Hoare triple does not trigger any exception. The evaluation of $\mathit{safe}(e) \ \wedge \ P[e/x]$ is left to right, $P[e/x]$ is evaluated if only if $\mathit{safe}(e)$ is true. Thus, the expression e might trigger and exception but the pre and postconditions of the Hoare triple not.

Compound. In the composition instruction, first the instruction s_1 is executed. If it 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.

Conditional Instruction. In the conditional instruction, s_1 is executed if e evaluates to true, and the result of the conditional is the postcondition of s_1. If e evaluates to false, s_2 is executed.

Check Instruction. If the condition of the check instruction evaluates to true, then the instruction terminates normally and $P \ \wedge \ e$ holds. If e is false, an exception is triggered and $P \ \wedge \ \neg e$ holds.

Loop Instruction. In the loop instruction, first the body of the `from` (s_1) is executed. If s_1 throws an exception, then the postcondition of the loop is the postcondition of s_1 (R_e). If s_1 finishes in normal execution, then the body of the loop (s_2) is executed. If s_2 finishes in normal execution then the invariant I holds, and if s_2 throws an exception, R_e holds. The implication $I \ \Rightarrow I'$ shows that the loop invariant used in the proof (I) implies the user's invariant (I').

Read Attribute Instruction. If y is not *Void* the value of the attribute a defined in the class T of the object y is assigned to x. Otherwise, an exception is triggered and Q_e holds.

Write Attribute Instruction. Similar to read attribute, if y is not *Void*, the attribute a defined in the class T of the object y is updated with the expression e. Otherwise, an exception is triggered and Q_e holds.

3.2 Creation, Routines, Routine Bodies and Routine Invocations Rules

This section presents the adaptation of new, read field, write field and invocations rules from Müller and Poetzsch-Heffter [12, 20, 21, 22] to Eiffel.

Local

In Eiffel, local variables have default values. To initialize local variables we use the function *init*. The following rule is used to initializes default values:

$$\frac{\mathcal{A} \rhd \left\{ \ P \ \wedge \ v_1 = \mathit{init}(T_1) \ \wedge \ ... \wedge \ v_n = \mathit{init}(T_n) \ \right\} \ \ s \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}}{\mathcal{A} \rhd \left\{ \ P \ \right\} \ \ \texttt{local} \ \ v_1 : T_1; \ ... \ v_n : T_n; \ s \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}}$$

Assignment Instruction

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

Compound

$$\frac{\begin{array}{c} \mathcal{A} \triangleright \left\{ \ P \ \right\} \ s_1 \ \left\{ \ Q_n \ , \ R_e \ \right\} \\ \mathcal{A} \triangleright \left\{ \ Q_n \ \right\} \ s_2 \ \left\{ \ R_n \ , \ R_e \ \right\} \end{array}}{\mathcal{A} \triangleright \left\{ \ P \ \right\} \ s_1 ; s_2 \ \left\{ \ R_n \ , \ R_e \ \right\}}$$

Conditional Instruction

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

Check Instruction

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

Loop Instruction

$$\frac{\begin{array}{c} I \ \Rightarrow I' \\ \mathcal{A} \triangleright \left\{ \ P \ \right\} \ s_1 \ \left\{ \ I \ , \ R_e \ \right\} \\ \mathcal{A} \triangleright \left\{ \ \neg e \ \wedge \ I \ \right\} \ s_2 \ \left\{ \ I \ , \ R_e \ \right\} \end{array}}{\mathcal{A} \triangleright \left\{ \ P \ \right\} \ \texttt{from} \ s_1 \ \texttt{invariant} \ I' \ \texttt{until} \ e \ \texttt{loop} \ s_2 \ \texttt{end} \ \left\{ \ (I \ \wedge \ e) \ , \ R_e \ \right\}}$$

Read Attribute Instruction

$$\frac{}{\triangleright \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[\$(\mathit{instvar}(y, T@a))/x]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e) \end{array} \right\} \ x := y.T@a \ \left\{ \ P \ , \ Q_e \ \right\}}$$

Write Attribute Instruction

$$\frac{}{\triangleright \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[\$ < \mathit{instvar}(y, T@a) := e > /\$]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e) \end{array} \right\} \ y.T@a := e \ \left\{ \ P \ , \ Q_e \ \right\}}$$

Figure 5: Axiomatic Semantics for Basic Instructions

where P does not contain $v_1, ..., v_n$.

Routine Invocation.

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

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

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 [21].

The following rule expresses the fact that local variables different from the left-hand-side variable are not modified by an invocation. This rule allows one to substitute logical variables Z in preconditions and postconditions by local variables w (w different from x).

$$\frac{\mathcal{A} \rhd \{\ P\ \}\quad x := y.T{:}m(e)\quad \{\ Q_n\ ,\ Q_e\ \}}{\mathcal{A} \rhd \{\ P[w/Z]\ \}\quad x := y.T{:}m(e)\quad \{\ Q_n[w/Z]\ ,\ Q_e[w/Z]\ \}}$$

Routine Implementation.

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

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

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 $\mathcal{A}$.

Creation

The creation instruction creates an object of type T and then invokes the routine *make*. In the following rule, the object creation is expressed by the replacement $\mathit{new}(\$, T)/\mathit{Current}, \$ < T > /\$$. The replacement e/p is added due to the routine invocation. Finally, in the postcondition, the *Current* object is replaced by x .

$$\frac{\mathcal{A} \rhd \{\ P\ \}\quad T : \mathit{make}\quad \{\ Q_n\ ,\ Q_e\ \}}{\mathcal{A} \rhd \left\{\ P \left[\begin{array}{l} \mathit{new}(\$, T)/\mathit{Current}, \\ \$ < T > /\$, \\ e/p \end{array} \right] \right\}\quad x := \texttt{create}\ \{T\}.\mathit{make}(e)\quad \{\ Q_n[x/\mathit{Current}]\ ,\ Q_e[x/\mathit{Current}]\ \}}$$

Subtype

$$\frac{\begin{array}{c} S \preceq T \\ \mathcal{A} \rhd \left\{\ P\ \right\}\quad S{:}m\quad \left\{\ Q_n\ ,\ Q_e\ \right\} \end{array}}{\mathcal{A} \rhd \left\{\ \tau(\mathit{Current}) \preceq S\ \wedge\ P\ \right\}\quad T{:}m\quad \left\{\ Q_n\ ,\ Q_e\ \right\}}$$

Class Rule

$$\frac{\begin{array}{l}\mathcal{A} \rhd \left\{ \tau(Current) = T \ \wedge \ P \right\} \ imp(T,m) \ \left\{ Q_n \ , \ Q_e \right\} \\ \mathcal{A} \rhd \left\{ \tau(Current) \prec T \ \wedge \ P \right\} \ T{:}m \ \left\{ Q_n \ , \ Q_e \right\}\end{array}}{\mathcal{A} \rhd \left\{ \tau(Current) \preceq T \ \wedge \ P \right\} \ T{:}m \ \left\{ Q_n \ , \ Q_e \right\}}$$

3.3 Exception Handling

The operation semantics presented in Section 2.3.3 shows that a rescue clause and the *Retry* 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 prove this loop, we use an invariant I_r. We call this invariant *rescue invariant*. The rule is the following:

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

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:

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

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

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

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

The rule for once functions is defined as follows:

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

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 2.3.4. 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.

3.5 Language-Independent Rules

The rules we have presented in the above sections depend from the specific instructions and features of the programming language, in this case Eiffel. Figure 6 presents rules that can be applied to any programming language. The *false axiom* allows us to prove anything assuming false. The *strength rule* allows us to proof a Hoare triple with an stronger precondition if the precondition P' implies the precondition P, and the Hoare triple can be proved using the precondition P. The *weak rule* is similar but it weakens the postcondition. This rule can be used to weaken both the normal postcondition Q_n, and the exceptional postcondition Q_e.

The *conjunction* and *disjunction rule* given the two proofs for the same instruction but using possible different pre- and postconditions, it concludes the conjunction and disjunction of the pre- and postcondition respectively. The *invariant rule* conjuncts W in the precondition and postcondition assuming that W does not contain neither program variables or $. The *substitution rule* substitutes Z by t in the precondition and postcondition. Finally, the *all-rule* and *ex-rule* introduces universal and existential quantifiers respectively.

4 Application

Figure 7 presents an example of the application of the logic. The function *safe_division* implements an integer division which terminates always normally. If the second operand is zero, this function returns the first operand; 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 prove this example, we first apply the routine implementation rule (Section 3.2). Then, we prove the initialization $z = 0$ using the local rule presented in Section 3.2. Next, we apply the rescue rule (Section 3.3) to prove the rescue block. Finally, we prove the body of the do block and the body of the rescue block using the assignment rule.

5 Soundness and Completeness Theorems

We have proved soundness and completeness of the logic. The proofs run by induction of the structure of the derivation tree for $\mathcal{A} \mid\!\triangleright \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$. In this section, we present the theorems. The soundness proof is presented in Appendix A, and the completeness proof is presented in Appendix A.3.

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

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

Theorem 1 (Soundness Theorem)

$$\triangleright \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \} \Rightarrow \models \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

Theorem 2 (Completeness Theorem)

$$\models \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \} \Rightarrow \triangleright \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

6 Related Work

Huisman and Jacobs [6] 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

Assumpt-axiom

$$\frac{}{\mathbf{A} \rhd \mathbf{A}}$$

False axiom

$$\frac{}{\rhd \left\{ \mathit{false} \right\} \; s_1 \; \left\{ \mathit{false} \; , \; \mathit{false} \right\}}$$

Assumpt-intro-axiom

$$\frac{\mathcal{A} \rhd \mathbf{A}}{\mathbf{A_0}, \mathcal{A} \rhd \mathbf{A}}$$

Assumpt-elim-axiom

$$\frac{\mathcal{A} \rhd \mathbf{A_0} \quad \mathbf{A_0}, \mathcal{A} \rhd \mathbf{A}}{\mathcal{A} \rhd \mathbf{A}}$$

Strength

$$\frac{P' \Rightarrow P \quad \mathcal{A} \rhd \left\{ P \right\} \; s_1 \; \left\{ Q_n \; , \; Q_e \right\}}{\mathcal{A} \rhd \left\{ P' \right\} \; s_1 \; \left\{ Q_n \; , \; Q_e \right\}}$$

Weak

$$\frac{\mathcal{A} \rhd \left\{ P \right\} \; s_1 \; \left\{ Q_n \; , \; Q_e \right\} \quad Q_n \Rightarrow Q_n' \quad Q_e \Rightarrow Q_e'}{\mathcal{A} \rhd \left\{ P \right\} \; s_1 \; \left\{ Q_n' \; , \; Q_e' \right\}}$$

Conjunction

$$\frac{\mathcal{A} \rhd \left\{ P^1 \right\} \; s_1 \; \left\{ Q_n^1 \; , \; Q_e^1 \right\} \quad \mathcal{A} \rhd \left\{ P^2 \right\} \; s_1 \; \left\{ Q_n^2 \; , \; Q_e^2 \right\}}{\mathcal{A} \rhd \left\{ P^1 \wedge P^2 \right\} \; s_1 \; \left\{ Q_n^1 \wedge Q_n^2 \; , \; Q_e^1 \wedge Q_e^2 \right\}}$$

Disjunction

$$\frac{\mathcal{A} \rhd \left\{ P^1 \right\} \; s_1 \; \left\{ Q_n^1 \; , \; Q_e^1 \right\} \quad \mathcal{A} \rhd \left\{ P^2 \right\} \; s_1 \; \left\{ Q_n^2 \; , \; Q_e^2 \right\}}{\mathcal{A} \rhd \left\{ P^1 \vee P^2 \right\} \; s_1 \; \left\{ Q_n^1 \vee Q_n^2 \; , \; Q_e^1 \vee Q_e^2 \right\}}$$

Invariant

$$\frac{\mathcal{A} \rhd \left\{ P \right\} \; s_1 \; \left\{ Q_n \; , \; Q_e \right\}}{\mathcal{A} \rhd \left\{ P \wedge W \right\} \; s_1 \; \left\{ Q_n \wedge W \; , \; Q_e \wedge W \right\}}$$

where W is a $\Sigma -$ formula, i.e. does not contain program variables or \$.

Substitution

$$\frac{\mathcal{A} \rhd \left\{ P \right\} \; s_1 \; \left\{ Q_n \; , \; Q_e \right\}}{\mathcal{A} \rhd \left\{ P[t/Z] \right\} \; s_1 \; \left\{ Q_n[t/Z] \; , \; Q_e[t/Z] \right\}}$$

where Z is an arbitrary logical variable and t a $\Sigma -$ term.

all-rule

$$\frac{\mathcal{A} \rhd \left\{ P[Y/Z] \right\} \; s_1 \; \left\{ Q_n \; , \; Q_e \right\}}{\mathcal{A} \rhd \left\{ P[Y/Z] \right\} \; s_1 \; \left\{ \forall Z : Q_n \; , \; \forall Z : Q_e \right\}}$$

where Z, Y are arbitrary, but distinct logical variables.

ex-rule

$$\frac{\mathcal{A} \rhd \left\{ P[Y/Z] \right\} \; s_1 \; \left\{ Q_n \; , \; Q_r \right\} Q_e}{\mathcal{A} \rhd \left\{ P[Y/Z] \right\} \; s_1 \; \left\{ \exists Z : Q_n \; , \; \exists Z : Q_e \right\}}$$

where Z, Y are arbitrary, but distinct logical variables.

Figure 6: Language-Independent Rules

```
safe_division (x,y: INTEGER): INTEGER
  local
    z: INTEGER
  do
    { z=0 or z=1 }
    Result := x // (y+z)
    { y = 0 implies Result = x and y /= 0 implies Result = x // y , z = 0 }
  ensure
    zero: y = 0 implies Result = x
    not_zero: y /= 0 implies Result = x // y
  rescue
    { z=0 }
    z := 1
    { z=1 , false }
    Retry := true
    { Retry implies z=1 and not Retry implies false, false }
  end
```

Figure 7: Example of an Eiffel source proof.

a specific language such as PVS or Isabelle. Oheimb [26] 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 [23, 15], dynamic frames [7, 25], and regions [2] have been proposed to solve a key issue for reasoning about imperative programs: framing. Separation logic has been adapted to verify object-oriented programs [16, 17, 4]. Parkinson and Bierman [16, 17] introduce abstract predicates: a powerful means to abstract from implementation details and to support information hiding and inheritance. Distefano and Parkinson [4] 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 [3] 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 [20, 21], and it supports object-oriented features such as inheritance and dynamic binding. The Mobius project [11] 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 [18] 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 [13] 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 [24]. 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 [20, 21], 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 [14] 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 $\{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_r,\ Q_e\ \}$ 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 [10].

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 [13]. 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 [21] 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 $factorial(n)$ 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 [8].

References

[1] K. R. Apt. Ten Years of Hoare's Logic: A Survey—Part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[2] 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.

[3] 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.

[4] 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.

[5] G. Gorelick. A Complete Axiomatic System for Proving Assertions about Recursive and Non-Recursive Programs. Technical Report TR-75, Department of Computer Science, University of Toronto, 1975.

[6] 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.

[7] I. T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*, pages 268–283, 2006.

[8] 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.

[9] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

[10] 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.

[11] MOBIUS Consortium. Deliverable 3.1: Byte code level specification language and program logic. Available online from http://mobius.inria.fr, 2006.

[12] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[13] 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.

[14] 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.

[15] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04*, pages 268–280, 2004.

[16] M. J. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL '05*, volume 40, pages 247–258. ACM, 2005.

[17] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08*, pages 75–86. ACM, 2008.

[18] M. Pavlova. *Java Bytecode verification and its applications.* PhD thesis, University of Nice Sophia-Antipolis, 2007.

[19] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

[20] 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.

[21] 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.

[22] 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.

[23] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[24] B. Schoeller. *Making classes provable through contracts, models and frames.* PhD thesis, ETH Zurich, 2007.

[25] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *Formal Techniques for Java-like Programs*, 2008.

[26] D. von Oheimb. *Analyzing Java in Isabelle/HOL - Formalization, Type Safety and Hoare Logic -*. PhD thesis, Universität München, 2001.

[27] D. von Oheimb. Hoare Logic for Java in Isabelle/HOL. In *special issue of Concurrency and Computation: Practice and Experience*, volume 13, pages 1173–1214, November 2001.

A Appendix: Soundness and Completeness Proof

To handle recursive calls, we define a richer semantic relation $\to_N$ where N captures the maximal depth of nested method calls which is allowed during the execution of the instruction. The transition $\sigma, S \ \to_N \sigma', normal$ expresses that executing the instruction S in the state σ does not lead to more than N nested calls, and terminates normally in the state σ'. The transition $\sigma, S \ \to_N \sigma', exc$ expresses that executing the instruction S in the state σ does not lead to more than N nested calls, and terminales with an exception in the state σ'.

The rules defining $\to_N$ are similar to the rule of $\to$ presented in Section 2.3 except for the additional parameter N. For the rules that do not describe the semantics of neither a routine call, nor a once routine, nor a creation procedure, we replace $\to$ by $\to_N$. For example, the compound rule (2.4) is defined as follows:

$$\frac{\langle \sigma, s_1 \rangle \to_N \sigma', normal \qquad \langle \sigma', s_2 \rangle \to_N \sigma'', \chi}{\langle \sigma, s_1; s_2 \rangle \to_N \sigma'', \chi}$$

Routine Invocations. The routine invocation rule described in Section 2.3.2 is extended using the transition $\to_N$ as follows:

$$\frac{\begin{array}{c} T{:}m \ \textit{is not a once routine} \\ \sigma(y) \neq voidV \qquad \langle \sigma[Current := \sigma(y), p := \sigma(e)],\ body(impl(\tau(\sigma(y)), m)) \rangle \to_N \sigma', \chi \end{array}}{\langle \sigma, x := y.T{:}m(e) \rangle \to_{N+1} \sigma'[x := \sigma'(Result)], \chi} \tag{13}$$

Once Routines. The only rules of once routines that are extended using the transition $\to_N$ are the rules that describe the execution of the first invocation. These rules is extended as follows:

$$\frac{\begin{array}{c} T'@m = impl(\tau(\sigma(y)), m) \quad T'@m \ \textit{is a once routine} \\ \sigma(T'@m\_done) = false \\ \langle \sigma[T'@m\_done := true, Current := y, p := \sigma(e)],\ body(T'@m) \rangle \to_N \sigma', normal \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \to_{N+1} \sigma'[x := \sigma'(Result)], normal}$$

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \ \textit{is a once routine} \\ \sigma(T@m\_done) = false \\ \langle \sigma[T@m\_done := true, Current := y, p := \sigma(e)],\ body(T@m) \rangle \to_N \sigma', exc \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \to_{N+1} \sigma'[T@m\_exc := true], exc}$$

A.1 Definitions and Theorems

To handle recursion, following we extend the semantics of Hoare triples, and the soundness and completeness theorems.

Definition 4 (Triple $\models$) $\models \{ \ P \ \} \quad s \quad \{ \ Q_n \ , \ Q_e \ \}$ *is defined as follows:*

- *If s is an instruction, then*
  $\models \{ \ P \ \} \quad s \quad \{ \ Q_n \ , \ Q_e \ \}$ *if only if:*
  for all $\sigma \models P : \langle \sigma, s \rangle \to \sigma', \chi$ *then*
  - $\chi = normal \ \Rightarrow \sigma' \models Q_n$*, and*
  - $\chi = exc \ \Rightarrow \sigma' \models Q_e$
- *If s is the routine implementation $T@m$, then*
  for all $\sigma \models P : \langle \sigma, body(T@m) \rangle \to \sigma', \chi$ *then*
  - $\chi = normal \ \Rightarrow \sigma' \models Q_n$*, and*

- $\chi = exc \Rightarrow \sigma' \models Q_e$

- *If s is the virtual routine $T{:}m$, then*
  for all $\sigma \models P : \langle \sigma, body(imp(\tau(\sigma(Current)), m)) \rangle \rightarrow \sigma', \chi$ *then*
  - $\chi = normal \Rightarrow \sigma' \models Q_n$, *and*
  - $\chi = exc \Rightarrow \sigma' \models Q_e$

The definition of $\models \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$ (Definition 4) uses the transition $\rightarrow$. To handle recursive routine calls, we extend this definition using the transition $\rightarrow_N$ as follows:

Definition 5 (Triple $\models_N$) $\models_N \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$ *is defined as follows:*

- *If s is an instruction, then*
  $\models_N \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$ *if only if:*
  for all $\sigma \models P : \langle \sigma, s \rangle \rightarrow_N \sigma', \chi$ *then*
  - $\chi = normal \Rightarrow \sigma' \models Q_n$, *and*
  - $\chi = exc \Rightarrow \sigma' \models Q_e$

- *If s is the routine implementation $T@m$, then*
  $\models_0 \{\ P\ \}\ \ T@m\ \ \{\ Q_n\ ,\ Q_e\ \}$ ***always holds****; and*
  $\models_{N+1} \{\ P\ \}\ \ T@m\ \ \{\ Q_n\ ,\ Q_e\ \}$ *if only if* $\models_N \{\ P\ \}\ \ body(T@m)\ \ \{\ Q_n\ ,\ Q_e\ \}$

- *If s is the virtual routine $T{:}m$, then*
  $\models_N \{\ P\ \}\ \ T{:}m\ \ \{\ Q_n\ ,\ Q_e\ \}$ *if only if* $\models_N \{\ P\ \}\ \ imp(\tau(Current), m)\ \ \{\ Q_n\ ,\ Q_e\ \}$

The above definition presents the semantics for Hoare Triples with empty assumptions. The following definition introduces the semantics of sequent:

Definition 6 (Sequent Holds)

$\{P^1\}s_1\{Q^1_n\ ,\ Q^1_e\}, ..., \{P^j\}s_j\{Q^j_n\ ,\ Q^j_e\} \models \{P\}\ \ s\ \ \{Q_n\ ,\ Q_e\}$ *if only if:*

for all N: $\models_N \{P^1\}s_1\{Q^1_n\ ,\ Q^1_e\}$ *and ... and* $\models_N \{P^j\}s_j\{Q^j_n\ ,\ Q^j_e\}$ *implies*

$$\models_N \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

Now, the theorems can be presented using the definition of sequent holds (Definition 6). The theorems are the followings:

Theorem 3 (Soundness Theorem)

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

Theorem 4 (Completeness Theorem)

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

Following, we present the proofs of the soundness and completeness theorems.

A.2 Soundness Proof

The followings auxiliary lemmas are used to prove soundness:

Lemma 3 (Triple $\models$, Triple $\models_N$)

$$\models \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ \textit{if only if}\ \ \forall N : \models_N\ \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

Lemma 4 (Monotone $\to_N$)

$$\langle \sigma,\ s_1 \rangle \to_N \sigma', \chi\ \Rightarrow \langle \sigma,\ s_1 \rangle \to_{N+1} \sigma', \chi$$

Lemma 5 ($\to$ iff $\to_N$)

$$\langle \sigma,\ s_1 \rangle \to \sigma', \chi\ \ \textit{if only if}\ \ \exists N : \langle \sigma,\ s_1 \rangle \to_N \sigma', \chi$$

Lemma 6 (Monotone $\models_N$)

$$\models_{N+1} \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ \textit{implies}\ \ \models_N \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

The proof of soundness runs by induction on the structure of the derivation tree for:

$$\mathcal{A} \rhd \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

and the operational semantics. Following, we present the proof for the most interesting rules.

A.2.1 Assignment Axiom

We have to prove:

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

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

Let P' be $(\textit{safe}(e)\ \wedge\ P[e/x])\ \vee\ (\neg \textit{safe}(e)\ \wedge\ Q_e)$. Applying Definition 5, and Definition 6 to the consequence of the rule, we have to prove:
$\forall \sigma \models P' : \langle \sigma,\ x := e \rangle \to_N \sigma', \chi$ *then*

$$\begin{array}{rcl} \chi = \textit{normal} & \Rightarrow & \sigma' \models P,\ \textit{and} \\ \chi = \textit{exc} & \Rightarrow & \sigma' \models Q_e \end{array}$$

We prove it doing case analysis on χ:
Case 1: $\chi =$ **exc**. By the definition of the operational semantics, we have:

$$\frac{\sigma(e) = \textit{exc}}{\langle \sigma, x := e \rangle \to_N \sigma, \textit{exc}}$$

Thus, we have to prove $\sigma \models Q_e$. Since $\sigma(e) = \textit{exc}$, applying Lemma 1, we know $\sigma \models \neg \textit{safe}(e)$. Since $\sigma \models P'$, and σ does not change, then $\sigma \models Q_e$.
Case 2: $\chi =$ **normal**. By the definition of the operational semantics, we get:

$$\frac{\sigma(e) \neq \textit{exc}}{\langle \sigma, x := e \rangle \to_N \sigma[x := \sigma(e)], \textit{normal}}$$

Thus, we have to prove $\sigma[x := \sigma(e)] \models P$. Since $\sigma(e) \neq \textit{exc}$, applying Lemma 1, we know $\sigma \models \textit{safe}(e)$. Since $\sigma \models P'$, then $\sigma \models \textit{safe}(e) \wedge P[e/x]$. Applying Lemma 2, then $\sigma[x := \sigma(e)] \models P$.
$\square$

A.2.2 Compound Rule

We have to prove:

$$\mathcal{A} \rhd \{\ P\ \}\quad s_1; s_2\quad \{\ R_n\ ,\ R_e\ \}\ \mathit{implies}\ \mathcal{A} \models \{\ P\ \}\quad s_1; s_2\quad \{\ R_n\ ,\ R_e\ \}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \rhd \{\ P\ \}\quad s_1\quad \{\ Q_n\ ,\ R_e\ \}\quad \mathit{implies}\ \mathcal{A} \models \{\ P\ \}\quad s_1\quad \{\ Q_n\ ,\ R_e\ \} \\ \mathcal{A} \rhd \{\ Q_n\ \}\quad s_2\quad \{\ R_n\ ,\ R_e\ \}\quad \mathit{implies}\ \mathcal{A} \models \{\ Q_n\ \}\quad s_2\quad \{\ R_n\ ,\ R_e\ \} \end{array}$$

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies} \models_N \{\ P\ \}\quad s_1; s_2\quad \{\ R_n\ ,\ R_e\ \}$$

using the hypotheses:

$$\begin{array}{l} \mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies} \models_N \{\ P\ \}\quad s_1\quad \{\ Q_n\ ,\ R_e\ \} \\ \mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies} \models_N \{\ Q_n\ \}\quad s_2\quad \{\ R_n\ ,\ R_e\ \} \end{array}$$

Since the sequent $\mathcal{A}$ is the same in the hypotheses and the conclusion, and since s_1 and s_2 are instructions, applying Definition 5, we have to show:

$$\begin{array}{l} \text{for all } \sigma \models P : \langle \sigma,\ s_1; s_2 \rangle \rightarrow_N \sigma'', \chi\ \ \mathit{then} \\ \qquad\qquad \chi = \mathit{normal}\ \Rightarrow \sigma'' \models R_n,\ \mathit{and} \\ \qquad\qquad \chi = \mathit{exc}\ \Rightarrow \sigma'' \models R_e \end{array} \tag{14}$$

using the hypotheses:

$$\begin{array}{l} \text{for all } \sigma \models P : \langle \sigma,\ s_1 \rangle \rightarrow_N \sigma', \chi\ \ \mathit{then} \\ \qquad\qquad \chi = \mathit{normal}\ \Rightarrow \sigma' \models Q_n,\ \mathit{and} \\ \qquad\qquad \chi = \mathit{exc}\ \Rightarrow \sigma' \models R_e \end{array} \tag{15}$$

and

$$\begin{array}{l} \text{for all } \sigma' \models Q_n : \langle \sigma',\ s_2 \rangle \rightarrow_N \sigma'', \chi\ \ \mathit{then} \\ \qquad\qquad \chi = \mathit{normal}\ \Rightarrow \sigma'' \models R_n,\ \mathit{and} \\ \qquad\qquad \chi = \mathit{exc}\ \Rightarrow \sigma'' \models R_e \end{array} \tag{16}$$

We prove it doing case analysis on χ:
Case 1: $\chi =$ **exc**. By the definition of the operational semantics for compound we get:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \mathit{exc}}{\langle \sigma, s_1; s_2 \rangle \rightarrow_N \sigma', \mathit{exc}}$$

Since $\sigma \models P$, then by the first hypothesis (33) we get $\sigma' \models R_e$.
Case 2: $\chi =$ **normal**. By the definition of the operational semantics for compound we have:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \mathit{normal} \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \chi}{\langle \sigma, s_1; s_2 \rangle \rightarrow_N \sigma'', \chi}$$

We can apply the first induction hypothesis (33) we get $\sigma' \models Q_n$ since $\sigma \models P$. Then, we can apply the second induction hypothesis (16) and get:

$$\chi = \mathit{normal}\ \Rightarrow \sigma'' \models R_n,\ \mathit{and}\ \chi = \mathit{exc}\ \Rightarrow \sigma'' \models R_e$$

□

A.2.3 Conditional Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \rhd \left\{\ P\ \right\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}\ \left\{\ Q_n\ ,\ Q_e\ \right\} \mathit{implies} \\ \mathcal{A} \models \left\{\ P\ \right\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}\ \left\{\ Q_n\ ,\ Q_e\ \right\} \end{array}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \rhd \left\{\ P\ \wedge\ e\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_e\ \right\}\quad \mathit{implies}\ \mathcal{A} \models \left\{\ P\ \wedge\ e\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_e\ \right\} \\ \mathcal{A} \rhd \left\{\ P\ \wedge\ \neg e\ \right\}\ s_2\ \left\{\ Q_n\ ,\ Q_e\ \right\}\quad \mathit{implies}\ \mathcal{A} \models \left\{\ P\ \wedge\ \neg e\ \right\}\ s_2\ \left\{\ Q_n\ ,\ Q_e\ \right\} \end{array}$$

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies} \models_N \left\{\ P\ \right\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}\ \left\{\ Q_n\ ,\ Q_e\ \right\}$$

using the hypotheses:

$$\begin{array}{l} \mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies} \models_N \left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_e\ \right\} \\ \mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies} \models_N \left\{\ P\ \right\}\ s_2\ \left\{\ Q_n\ ,\ Q_e\ \right\} \end{array}$$

Since the sequent $\mathcal{A}$ is the same in the hypotheses and the conclusions, and s_1 and s_2 are instructions, applying Definition 5 we have to prove:

$$\begin{array}{r} \forall \sigma \models P : \langle \sigma,\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \rangle \rightarrow_N \sigma', \chi\ \mathit{then} \\ \chi = \mathit{normal}\ \Rightarrow \sigma' \models Q_n,\ \mathit{and} \\ \chi = \mathit{exc}\ \Rightarrow \sigma' \models Q_e \end{array} \tag{17}$$

using the hypotheses:

$$\begin{array}{r} \forall \sigma \models (P\ \wedge\ e) : \langle \sigma,\ s_1\ \rangle \rightarrow_N \sigma', \chi\ \mathit{then} \\ \chi = \mathit{normal}\ \Rightarrow \sigma' \models Q_n,\ \mathit{and} \\ \chi = \mathit{exc}\ \Rightarrow \sigma' \models Q_e \end{array} \tag{18}$$

and

$$\begin{array}{r} \forall \sigma \models (P\ \wedge\ \neg e) : \langle \sigma,\ s_2\ \rangle \rightarrow_N \sigma', \chi\ \mathit{then} \\ \chi = \mathit{normal}\ \Rightarrow \sigma' \models Q_n,\ \mathit{and} \\ \chi = \mathit{exc}\ \Rightarrow \sigma' \models Q_e \end{array} \tag{19}$$

We prove this rule doing case analysis on $\sigma(e)$:

Case 1: $\sigma(\mathbf{e}) = \mathbf{True}$. If $\sigma(e) = \mathit{True}$ then by the definition of the operational semantics we get:

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

Then applying the first hypothesis (18) we prove $\chi = \mathit{normal}\ \Rightarrow \sigma' \models Q_n$, and $\chi = \mathit{exc}\ \Rightarrow \sigma' \models Q_e$.

Case 2: $\sigma(\mathbf{e}) = \mathbf{False}$. If $\sigma(e) = \mathit{False}$ then by the definition of the operational semantics we get:

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

Then applying the second hypothesis (19) we prove $\chi = \mathit{normal}\ \Rightarrow \sigma' \models Q_n$, and $\chi = \mathit{exc}\ \Rightarrow \sigma' \models Q_e$.

□

A.2.4 Check Axiom

We have to prove:

$$\triangleright \{ P \} \texttt{ check } e \texttt{ end } \{ (P \wedge e), (P \wedge \neg e) \} \Rightarrow$$
$$\models \{ P \} \texttt{ check } e \texttt{ end } \{ (P \wedge e), (P \wedge \neg e) \}$$

Applying Definition 5 and Definition 6 to the consequence of the rule, we have to prove:

$$\begin{aligned} \forall \sigma \models P : \langle \sigma, \texttt{ check } e \texttt{ end} \rangle \rightarrow_N \sigma, \chi \textit{ then} & \\ \chi = \textit{normal} \Rightarrow \sigma' \models (P \wedge e), \textit{ and} & \\ \chi = \textit{exc} \Rightarrow \sigma' \models (P \wedge \neg e) & \end{aligned} \tag{20}$$

To prove it, we do case analysis on $\sigma(e)$:
Case 1: $\sigma(\mathbf{e}) = \mathbf{True}$. By the definition of the operational semantics we have:

$$\frac{\sigma(e) = \textit{True}}{\langle \sigma, \texttt{check } e \texttt{ end} \rangle \rightarrow_N \sigma, \textit{normal}}$$

Since the state σ is unchanged then $\sigma \models P$. Furthermore, $\sigma(e) = \textit{True}$ by this case analysis, then applying the definition of $\models$ we prove $\sigma \models (P \wedge e))$
Case 2: $\sigma(\mathbf{e}) = \mathbf{False}$. By the definition of the operational semantics we have:

$$\frac{\sigma(e) = \textit{False}}{\langle \sigma, \texttt{check } e \texttt{ end} \rangle \rightarrow_N \sigma, \textit{exc}}$$

Similar to the above case, $\sigma \models P$ since the state is unchanged and $\sigma(e) = \textit{False}$ by the case analysis. Then $\sigma \models (P \wedge \neg e)$ holds using the definition of $\models$.
□

A.2.5 Loop Rule

We have to prove:

$$\begin{aligned} &\mathcal{A} \triangleright \{ P \} \texttt{ from } s_1 \texttt{ invariant } I' \texttt{ until } e \texttt{ loop } s_2 \texttt{ end } \{ (I \wedge e), R_e \} \textit{ implies} \\ &\mathcal{A} \models \{ P \} \texttt{ from } s_1 \texttt{ invariant } I' \texttt{ until } e \texttt{ loop } s_2 \texttt{ end } \{ (I \wedge e), R_e \} \end{aligned}$$

using the induction hypotheses:

$$\begin{array}{ll} \mathcal{A} \triangleright \{ P \} \; s_1 \; \{ I, R_e \} & \textit{implies } \mathcal{A} \models \{ P \} \; s_1 \; \{ I, R_e \} \\ \mathcal{A} \triangleright \{ \neg e \wedge I \} \; s_2 \; \{ I, R_e \} & \textit{implies } \mathcal{A} \models \{ \neg e \wedge I \} \; s_2 \; \{ I, R_e \} \\ I \Rightarrow I' & \end{array}$$

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \models_N \mathcal{A}_n \textit{ implies} \models_N \{ P \} \texttt{ from } s_1 \ldots \{ (I \wedge e), R_e \}$$

using the hypotheses:

$$\begin{aligned} \textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \models_N \mathcal{A}_n \textit{ implies} \models_N \{ P \} \; s_1 \; \{ I, R_e \} \\ \textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \models_N \mathcal{A}_n \textit{ implies} \models_N \{ \neg e \wedge I \} \; s_2 \; \{ I, R_e \} \end{aligned}$$

Since the sequent $\mathcal{A}$ is the same in the hypotheses and the conclusions, and s_1 and s_2 are instructions, applying Definition 5 we have to prove:

$$\forall \sigma \models P : \forall \sigma \models P : \langle \sigma,\ \texttt{from}\ s_1\ \texttt{invariant}\ I'\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\rangle \rightarrow_N \sigma', \chi\ \mathit{then}$$
$$\begin{aligned} \chi = \mathit{normal} \quad &\Rightarrow \quad \sigma' \models (I \wedge e),\ \mathit{and} \\ \chi = \mathit{exc} \quad &\Rightarrow \quad \sigma' \models R_e \end{aligned}$$

using the hypotheses:

$$\begin{aligned} &\forall \sigma \models P : \langle \sigma,\ s_1\ \rangle \rightarrow_N \sigma', \chi\ \mathit{then} \\ &\qquad \chi = \mathit{normal} \ \Rightarrow \sigma' \models I,\ \mathit{and} \\ &\qquad \chi = \mathit{exc} \ \Rightarrow \sigma' \models R_e \end{aligned} \tag{21}$$

and

$$\begin{aligned} &\forall \sigma \models (\neg e\ \wedge I) : \langle \sigma,\ s_2\ \rangle \rightarrow_N \sigma', \chi\ \mathit{then} \\ &\qquad \chi = \mathit{normal} \ \Rightarrow \sigma' \models I,\ \mathit{and} \\ &\qquad \chi = \mathit{exc} \ \Rightarrow \sigma' \models R_e \end{aligned} \tag{22}$$

We prove this rule doing case analysis on χ:

Case 1: $\chi =$ **exc**. Since s_1 trigger an exception, by the definition of the operational semantics we get:

$$\frac{\langle \sigma, s_1\rangle \rightarrow_N \sigma', \mathit{exc}}{\langle \sigma, \texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\rangle \rightarrow_N \sigma', \mathit{exc}}$$

Since $\sigma \models P$, then by the first hypothesis (21) we prove $\sigma' \models R_e$.

Case 2: $\chi =$ **normal**. If s_1 terminates normally, by the operational semantics we have:

$$\langle \sigma,\ s_1\rangle \rightarrow_N \sigma', \mathit{normal}$$

We have several cases depending if e evaluates to true or not and if s_2 terminates normally or not:

Case 2.a: $\sigma'(\mathbf{e}) =$ **True**. By the operational semantics, we get:

$$\frac{\langle \sigma, s_1\rangle \rightarrow_N \sigma', \mathit{normal} \qquad \sigma'(e) = \mathit{True}}{\langle \sigma, \texttt{from}\ \ s_1\ \ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\rangle \rightarrow_N \sigma', \mathit{normal}}$$

Applying the first hypothesis (21), we get $\sigma' \models I$ and $\sigma' \models e$. Then by the definition of $\models$ we prove $\sigma' \models (I\ \wedge\ e)$.

Case 2.b: $\sigma'(\mathbf{e}) =$ **False**. Then we do case analysis on χ:

Case 2.b.1: $\chi =$ **exc**. By the definition of the operational semantics we have:

$$\frac{\langle \sigma, s_1\rangle \rightarrow_N \sigma', \mathit{normal} \qquad \sigma'(e) = \mathit{False} \qquad \langle \sigma', s_2\rangle \rightarrow_N \sigma'', \mathit{exc}}{\langle \sigma, \texttt{from}\ \ s_1\ \ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\rangle \rightarrow_N \sigma'', \mathit{exc}}$$

By the first hypothesis (21), we prove $\sigma' \models I$. Then since $\sigma'(e) = \mathit{False}$ and $\chi = \mathit{exc}$, we prove $\sigma'' \models R_e$.

Case 2.b.2: $\sigma'(\mathbf{e}) =$ **False and** $\chi =$ **normal**. By the definition of the operational semantics we have:

$$\frac{\begin{array}{c}\langle \sigma, s_1\rangle \rightarrow_N \sigma', \mathit{normal} \qquad \sigma'(e) = \mathit{False} \qquad \langle \sigma', s_2\rangle \rightarrow_N \sigma'', \mathit{normal} \\ \langle \sigma'', \texttt{from}\ \ \mathit{skip}\ \ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\rangle \rightarrow_N \sigma''', \chi\end{array}}{\langle \sigma, \texttt{from}\ \ s_1\ \ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\rangle \rightarrow_N \sigma''', \chi}$$

By the first hypothesis (21), we prove $\sigma' \models I$. Then since $\sigma'(e) = \mathit{False}$, we can apply the definition of $\models$ and the second hypothesis (22), and we get $\sigma'' \models I$. Now we can apply the induction hypothesis and prove

$$\chi = \mathit{normal} \ \Rightarrow \sigma' \models (I \wedge e),\ \mathit{and}\ \chi = \mathit{exc} \ \Rightarrow \sigma' \models R_e$$

□

A.2.6 Read Attribute Axiom

We have to prove:

$$\triangleright \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[\$(\mathit{instvar}(y, T@a))/x]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e) \end{array} \right\} \ x := y.T@a \ \left\{ \ P \ , \ Q_e \ \right\} \qquad \Rightarrow$$

$$\models \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[\$(\mathit{instvar}(y, T@a))/x]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e) \end{array} \right\} \ x := y.T@a \ \left\{ \ P \ , \ Q_e \ \right\}$$

Applying Definition 5, and Definition 6 to the consequence of the rule, we have to prove:

$$\begin{array}{r} \forall \sigma \models P' : \langle \sigma, \ x := y.T@a \rangle \rightarrow_N \sigma', \chi \ \mathit{then} \qquad\qquad\qquad \\ \chi = \mathit{normal} \ \Rightarrow \sigma' \models P, \ \mathit{and} \\ \chi = \mathit{exc} \ \Rightarrow \sigma' \models Q_e \qquad\quad \end{array} \tag{23}$$

where P' is defined as follows:

$$P' \equiv (y \neq \mathit{Void} \ \wedge \ P[\$(\mathit{instvar}(y, T@a))/x]) \ \vee \ (y = \mathit{Void} \ \wedge \ Q_e)$$

To prove it, we do case analysis on χ:
Case 1: $\chi =$ **normal**. Applying the definition of the operational semantics we have:

$$\frac{\sigma(y) \neq \mathit{voidV}}{\langle \sigma, x := y.T@a \rangle \rightarrow_N \sigma[x := \sigma(\$) \ (\mathit{instvar}(\sigma(y), T@a))], \mathit{normal}}$$

Then applying lemma 2 we get $\sigma \models P$.
Case 2: $\chi =$ **exc**. Applying the definition of the operational semantics:

$$\frac{\sigma(y) = \mathit{voidV}}{\langle \sigma, x := y.T@a \rangle \rightarrow_N \sigma, \mathit{exc}}$$

we get $\sigma \models Q_e$.
$\square$

A.2.7 Write Attribute Axiom

We have to prove:

$$\triangleright \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[\$ < \mathit{instvar}(y, T@a) := e > /\$]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e) \end{array} \right\} \ y.T@a := e \ \left\{ \ P \ , \ Q_e \ \right\} \qquad \Rightarrow$$

$$\models \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[\$ < \mathit{instvar}(y, T@a) := e > /\$]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e) \end{array} \right\} \ y.T@a := e \ \left\{ \ P \ , \ Q_e \ \right\}$$

Applying Definition 5, and Definition 6 to the consequence of the rule, we have to prove:

$$\begin{array}{r} \forall \sigma \models P' : \langle \sigma, \ y.T@a := e \rangle \rightarrow_N \sigma', \chi \ \mathit{then} \qquad\qquad\qquad \\ \chi = \mathit{normal} \ \Rightarrow \sigma' \models P, \ \mathit{and} \\ \chi = \mathit{exc} \ \Rightarrow \sigma' \models Q_e \qquad\quad \end{array} \tag{24}$$

where P' is defined as follows:

$$P' \equiv (y \neq \mathit{Void} \ \wedge \ P[\$ < \mathit{instvar}(y, T@a) := e > /\$]) \ \vee (y = \mathit{Void} \ \wedge \ Q_e)$$

To prove it, we do case analysis on χ:
Case 1: $\chi =$ **normal**. Applying the definition of the operational semantics we have:

$$\frac{\sigma(y) \neq voidV}{\langle \sigma, y.T@a := e \rangle \rightarrow_N \sigma[\$:= \sigma(\$) < instvar(\sigma(y), T@a) := \sigma(e) >], normal}$$

Then applying lemma 2 we get $\sigma \models P$.
Case 2: $\chi =$ **exc**. Applying the definition of the operational semantics:

$$\frac{\sigma(y) = voidV}{\langle \sigma, y.T@a := e \rangle \rightarrow_N \sigma, exc}$$

we get $\sigma \models Q_e$.
□

A.2.8 Local Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \rhd \{\ P\ \}\ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ implies \\ \mathcal{A} \models \{\ P\ \}\ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

using the induction hypothesis:

$$\begin{array}{l} \mathcal{A} \rhd \{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ implies \\ \mathcal{A} \models \{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

Let $\mathcal{A}$ be the sequent $\mathcal{A} = \mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$for\ all\ N : \models_N \mathcal{A}_1, and...,\ \models_N \mathcal{A}_n\ implies \models_N \{\ P\ \}\ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

using the hypothesis:

$$\begin{array}{c} for\ all\ N : \models_N \mathcal{A}_1, and...,\ \models_N \mathcal{A}_n\ implies \\ \models_N \{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

Since the sequent $\mathcal{A}$ is the same in the hypothesis and the conclusion, and since s is an instruction, applying Definition 5, we have to show:
$\forall \sigma \models P : \langle \sigma, \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s \rangle \rightarrow_N \sigma', \chi\ then$

$$\begin{array}{rcl} \chi = normal & \Rightarrow & \sigma' \models Q_n,\ and \\ \chi = exc & \Rightarrow & \sigma' \models Q_e \end{array}$$

using the hypothesis:

$$\begin{array}{l} \forall \sigma \models P' : \langle \sigma,\ s\ \rangle \rightarrow_N \sigma', \chi\ then \\ \qquad \chi = normal\ \Rightarrow \sigma' \models Q_n,\ and \\ \qquad \chi = exc\ \Rightarrow \sigma' \models Q_e \end{array} \tag{25}$$

where P' is defined as follows:

$$P' \equiv P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)$$

Applying the definition of the operational semantics for locals, we get:

$$\frac{\langle \sigma[v_1 := init(T_1), ..., v_n := init(T_n)],\ s \rangle \rightarrow_N \sigma', normal}{\langle \sigma, \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s \rangle \rightarrow_N \sigma', normal}$$

Then, applying Lemma 2 we know $\sigma \models P'$. Finally, applying the hypothesis (25) we prove:

$$\chi = normal\ \Rightarrow \sigma' \models Q_n,\ and \chi = exc\ \Rightarrow \sigma' \models Q_e$$

□

A.2.9 Creation Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \rhd \left\{ P \left[\begin{array}{l} new(\$, T)/Current, \\ \$ < T > /\$, e/p \end{array} \right] \right\} \; x := \texttt{create}\ \{T\}.make(e) \; \left\{ Q_n[x/Current] \ , \ Q_e[x/Current] \right\} \; implies \\ \mathcal{A} \models \left\{ P \left[\begin{array}{l} new(\$, T)/Current, \\ \$ < T > /\$, e/p \end{array} \right] \right\} \; x := \texttt{create}\ \{T\}.make(e) \; \left\{ Q_n[x/Current] \ , \ Q_e[x/Current] \right\} \end{array}$$

using the induction hypothesis:

$$\mathcal{A} \rhd \{ \ P \ \} \quad T : make \quad \{ \ Q_n \ , \ Q_e \ \} \quad implies\ \mathcal{A} \models \{ \ P \ \} \quad T : make \quad \{ \ Q_n \ , \ Q_e \ \}$$

Applying Definition 5 and Definition 6 to the consequence of the rule, we have to prove:
$\forall \sigma \models P' : \langle \sigma,\ x := \texttt{create}\ \{T\}.make(e) \rangle \rightarrow_{N+1} \sigma', \chi\ then$

$$\begin{array}{rcl} \chi = normal & \Rightarrow & \sigma' \models Q_n[x/Current],\ and \\ \chi = exc & \Rightarrow & \sigma' \models Q_e[x/Current] \end{array}$$

where P' is defined as follows:

$$P' \equiv P \left[\ new(\$, T)/Current,\ \$ < T > /\$, e/p \ \right]$$

using the hypothesis:
$\forall \sigma \models P : \langle \sigma,\ body(imp(T, make)) \rangle \rightarrow_N \sigma', \chi\ then$

$$\begin{array}{rcl} \chi = normal & \Rightarrow & \sigma' \models Q_n,\ and \\ \chi = exc & \Rightarrow & \sigma' \models Q_e \end{array}$$

We prove soundness of this rule with respect to the operational semantics of creation instruction (defined in Section 2.3 on page 12) for an arbitrary N.

Since $\sigma \models P[new(\$, T)/Current,\ \$ < T > /\$,\ e/p]$, then by lemma 2, we know

$$\sigma[Current := new(\sigma(\$), T), \$:= \sigma(\$) < T >, p := \sigma(e)] \models P$$

Applying the definition of the operational semantics we get

$$\begin{array}{ll} \chi = normal & \Rightarrow \sigma'[x := \sigma'(Current)] \models Q_n,\ and \\ \chi = exc & \Rightarrow \sigma'[x := \sigma'(Current)] \models Q_e \end{array}$$

Using lemma 2 we prove:

$$\chi = normal\ \Rightarrow \sigma' \models Q_n[x/Current],\ and\ \chi = exc\ \Rightarrow \sigma' \models Q_e[x/Current]$$

□

A.2.10 Rescue Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \rhd \{ \ P \ \} \quad s_1\ \texttt{rescue}\ s_2 \quad \{ \ Q_n \ , \ R_e \ \} \quad implies \\ \mathcal{A} \models \{ \ P \ \} \quad s_1\ \texttt{rescue}\ s_2 \quad \{ \ Q_n \ , \ R_e \ \} \end{array}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \rhd \{ \ I_r \ \} \quad s_1 \quad \{ \ Q_n \ , \ Q_e \ \}\ implies\ \mathcal{A} \rhd \{ \ I_r \ \} \quad s_1 \quad \{ \ Q_n \ , \ Q_e \ \} \\ and \\ \mathcal{A} \rhd \{ \ Q_e \ \} \quad s_2 \quad \{ \ Retry \Rightarrow I_r \ \wedge \ \neg Retry \Rightarrow R_e \ , \ R_e \ \} \quad implies \\ \mathcal{A} \models \{ \ Q_e \ \} \quad s_2 \quad \{ \ Retry \Rightarrow I_r \ \wedge \ \neg Retry \Rightarrow R_e \ , \ R_e \ \} \\ and \\ P \ \Rightarrow \ I_r \end{array}$$

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n \textit{ implies} \models_N \left\{\ P\ \right\}\ \ s_1 \texttt{ rescue } s_2\ \ \left\{\ Q_n\ ,\ R_e\ \right\}$$

using the hypotheses:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n \textit{ implies} \models_N \left\{\ I_r\ \right\}\ \ s_1\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}$$
$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n \textit{ implies} \models_N \left\{\ Q_e\ \right\}\ \ s_2\ \ \left\{\ \textit{Retry} \Rightarrow I_r\ \wedge\ \neg \textit{Retry} \Rightarrow R_e\ ,\ R_e\ \right\}$$

Since the sequent $\mathcal{A}$ is the same in the hypotheses and the conclusions, and s_1 and s_2 are instructions, applying Definition 5 we have to prove:
$\forall \sigma \models P : \langle \sigma,\ s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma', \chi \textit{ then}$

$$\begin{aligned} \chi = \textit{normal} \quad &\Rightarrow \quad \sigma' \models Q_n,\ \textit{and} \\ \chi = \textit{exc} \quad &\Rightarrow \quad \sigma' \models R_e \end{aligned}$$

using the hypotheses:

$$\begin{aligned} &\forall \sigma \models I_r : \langle \sigma,\ s_1\ \rangle \rightarrow_N \sigma', \chi \textit{ then} \\ &\qquad \chi = \textit{normal} \ \Rightarrow \sigma' \models Q_n,\ \textit{and} \\ &\qquad \chi = \textit{exc} \ \Rightarrow \sigma' \models Q_e \end{aligned} \tag{26}$$

and

$$\begin{aligned} &\forall \sigma \models Q_e : \langle \sigma,\ s_2\ \rangle \rightarrow_N \sigma'', \chi \textit{ then} \\ &\qquad \chi = \textit{normal} \ \Rightarrow \sigma'' \models (\textit{Retry} \Rightarrow I_r\ \wedge\ \neg \textit{Retry} \Rightarrow R_e),\ \textit{and} \\ &\qquad \chi = \textit{exc} \ \Rightarrow \sigma'' \models R_e \end{aligned} \tag{27}$$

We prove this rule doing case analysis on χ:
Case 1: $\chi =$ **normal**. Since s_1 terminates normally, by the definition of the operational semantics we get:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \textit{normal}}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma', \textit{normal}}$$

Then we can apply the first hypothesis (26) since $P \Rightarrow I_r$. Thus, we prove $\sigma' \models Q_n$.

Case 2: $\chi =$ **exc**. If s_1 triggers an exception, then by the operational semantics we have:

$$\langle \sigma,\ s_1 \rangle \rightarrow_N \sigma', \textit{exc}$$

We have several cases depending if *Retry* evaluates to true or not and if s_2 terminates normally or not:
Case 2.a: $\chi =$ **exc**. By the definition of the operational semantics we have:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \textit{exc} \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \textit{exc}}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma'', \textit{exc}}$$

By the first hypothesis (26), we prove $\sigma' \models Q_e$. Then, we can apply the second hypothesis (27) and prove $\sigma'' \models R_e$.

Case 2.b: $\chi =$ **normal**. Here we do case analysis on $\sigma''(\textit{Retry})$:
Case 2.b.1: σ''(**Retry**) $=$ **False**. By the definition of the operational semantics we have:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \textit{exc} \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \textit{normal} \qquad \neg \sigma''(\textit{Retry})}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma'', \textit{exc}}$$

By the first hypothesis (26), we prove $\sigma' \models Q_e$. Then, we can apply the second hypothesis (27) and we get $\sigma'' \models (\mathit{Retry} \Rightarrow I_r \;\wedge\; \neg \mathit{Retry} \Rightarrow R_e)$. Since $\sigma''(\mathit{Retry}) = \mathit{False}$ then by the definition of $\models$ we prove $\sigma'' \models R_e$.

Case 2.b.2: $\sigma''(\mathbf{Retry}) = \mathbf{True}$. By the definition of the operational semantics we have:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \mathit{exc} \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \mathit{normal} \qquad \sigma''(\mathit{Retry}) \qquad \langle \sigma'', s_1 \;\texttt{rescue}\; s_2 \rangle \rightarrow_N \sigma''', \chi}{\langle \sigma, s_1 \;\texttt{rescue}\; s_2 \rangle \rightarrow_N \sigma''', \chi}$$

By the first hypothesis (26), we prove $\sigma' \models Q_e$. Then, we can apply the second hypothesis (27) and we get $\sigma'' \models (\mathit{Retry} \Rightarrow I_r \;\wedge\; \neg \mathit{Retry} \Rightarrow R_e)$. Since $\sigma''(\mathit{Retry}) = \mathit{True}$ then by the definition of $\models$ we prove $\sigma'' \models I_r$. Now we can apply the induction hypothesis and we prove

$$\chi = \mathit{normal} \;\Rightarrow \sigma' \models Q_n, \;\; \mathit{and} \; \chi = \mathit{exc} \;\Rightarrow \sigma' \models R_e$$

□

A.2.11 Once Functions Rule

The proof of the rule for once functions (defined in Section 3.4 on page 19) is done in a similar way than the creation procedure. We use the once function rule and the invocation rule, and we prove they are sound with respect to the operation semantics of once (defined in Section 2.3.4 on page 12).

Let P be the following precondition, where T_M_RES is a logical variable:

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

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

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

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

To prove the once function rule, we have to prove:

$$\begin{array}{l} \mathcal{A} \rhd \{\; P \;\} \;\;\; T@m \;\;\; \{\; Q'_n \;,\; Q'_e \;\} \quad \mathit{implies} \\ \mathcal{A} \models \{\; P \;\} \;\;\; T@m \;\;\; \{\; Q'_n \;,\; Q'_e \;\} \end{array}$$

using the induction hypothesis:

$$\mathcal{A}, \{P\} \;\; T@m \; \{Q'_n, \; Q'_e\} \rhd$$

$$\left\{ \begin{array}{l} P'[\mathit{false}/T@m\_done] \wedge \\ T@m\_done \end{array} \right\} \; \mathit{body}(T@m) \;\; \{ \; (\;\; Q_n \;\wedge\; T@m\_done \;\;) \;,\; (\;\; Q_e \;\wedge\; T@m\_done \;\;) \;\; \} \quad \mathit{implies}$$

$$\mathcal{A}, \{P\} \;\; T@m \; \{Q'_n, \; Q'_e\} \models$$

$$\left\{ \begin{array}{l} P'[\mathit{false}/T@m\_done] \wedge \\ T@m\_done \end{array} \right\} \; \mathit{body}(T@m) \;\; \{ \; (\;\; Q_n \;\wedge\; T@m\_done \;\;) \;,\; (\;\; Q_e \;\wedge\; T@m\_done \;\;) \;\; \}$$

Let $\mathcal{A} = \mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\mathit{for \;\; all} \;\; N : \models_N \mathcal{A}_1, \mathit{and}..., \;\models_N \mathcal{A}_n \;\; \mathit{implies} \models_N \{\;\; P \;\;\} \quad T@m \quad \{\;\; Q'_n \;,\; Q'_e \;\;\}$$

using the hypothesis:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \ \models_N \mathcal{A}_n \textit{ and } \ \models_N \{P\} \quad T@m \ \{Q'_n, \ Q'_e\}$$

implies

$$\models_N \left\{ \begin{array}{l} P'[\textit{false}/T@m\_done] \wedge \\ T@m\_done \end{array} \right\} \ \textit{body}(T@m) \ \left\{ \left(\ Q_n \wedge T@m\_done \ \right), \left(\ Q_e \wedge T@m\_done \ \right) \right\} \tag{28}$$

We prove it by induction on N.
Base Case: $N = 0$. Holds by Definition 5.
Induction Case: $N \Rightarrow N + 1$. Assuming the induction hypothesis

$$\models_N \mathcal{A}_1, \textit{and...}, \ \models_N \mathcal{A}_n \textit{ implies} \models_N \{ \ P \ \} \quad T@m \quad \{ \ Q'_n \ , \ Q'_e \ \}$$

we have to show

$$\models_{N+1} \mathcal{A}_1, \textit{and...}, \ \models_{N+1} \mathcal{A}_n \textit{ implies} \models_{N+1} \{ \ P \ \} \quad T@m \quad \{ \ Q'_n \ , \ Q'_e \ \}$$

Then, we can prove this as follows:

$$\begin{array}{ll} \models_{N+1} \mathcal{A}_1, \textit{and...}, \ \models_{N+1} \mathcal{A}_n & \\ & \textit{implies } [\textit{Lemma } 6] \\ \models_N \mathcal{A}_1, \textit{and...}, \ \models_N \mathcal{A}_n & \\ & \textit{applying induction hypothesis} \\ \models_N \{ \ P \ \} \quad T@m \quad \{ \ Q'_n \ , \ Q'_e \ \} & \end{array}$$

Using $\models_N \ \mathcal{A}_1, \textit{and...}, \ \models_N \ \mathcal{A}_n$, and $\models_N \ \{ \ P \ \} \quad T@m \quad \{ \ Q'_n \ , \ Q'_e \ \}$, we can apply the hypothesis (28), and we get:

$$\models_N \left\{ \begin{array}{l} P'[\textit{false}/T@m\_done] \wedge \\ T@m\_done \end{array} \right\} \ \textit{body}(T@m) \ \{ \ (\ Q_n \ \wedge \ T@m\_done \ ) \ , \ (\ Q_e \ \wedge \ T@m\_done \ ) \ \} \tag{29}$$

Since we know (29) holds, we can prove $\models_{N+1} \{ \ P \ \} \quad T@m \quad \{ \ Q'_n \ , \ Q'_e \ \}$ using the hypothesis (29). Applying Definition 5, we have to prove

$$\models_N \{ \ P \ \} \quad \textit{body}(T@m) \quad \{ \ Q'_n \ , \ Q'_e \ \}$$

assuming

$$\models_N \left\{ \begin{array}{l} P'[\textit{false}/T@m\_done] \wedge \\ T@m\_done \end{array} \right\} \ \textit{body}(T@m) \ \{ \ (\ Q_n \ \wedge \ T@m\_done \ ) \ , \ (\ Q_e \ \wedge \ T@m\_done \ ) \ \}$$

Then, applying Definition 5 ($\models_N$), we have to prove:
$\forall \sigma \models P : \langle \sigma, \ \textit{body}(T@m) \rangle \rightarrow_N \sigma', \chi \textit{ then}$

$$\begin{array}{rcl} \chi = \textit{normal} & \Rightarrow & \sigma' \models Q'_n, \ \textit{and} \\ \chi = \textit{exc} & \Rightarrow & \sigma' \models Q'_e \end{array}$$

using the hypothesis:
$\forall \sigma \models P'[\textit{false}/T@m\_done] \wedge T@m\_done : \langle \sigma, \ \textit{body}(T@m) \rangle \rightarrow_N \sigma', \chi \textit{ then}$

$$\begin{array}{rl} \chi = \textit{normal} & \Rightarrow \sigma' \models Q_n \wedge \ T@m\_done, \ \textit{and} \\ \chi = \textit{exc} & \Rightarrow \sigma' \models Q_e \ \wedge \ T@m\_done \end{array} \tag{30}$$

We prove this with respect to the operational semantics of once routines (defined in page 12) by case analysis on χ and $T@m\_done$:

Case 1: σ(T@m_done) = false and χ = normal. By the definition of the operational semantics we have:

$$\frac{\begin{array}{c} T'@m = impl(\tau(\sigma(y)), m) \quad T'@m \textit{ is a once routine} \\ \sigma(T'@m\_done) = false \\ \langle \sigma[T'@m\_done := true, Current := y, p := \sigma(e)],\ body(T'@m)\rangle \rightarrow_N \sigma', normal \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow_{N+1} \sigma'[x := \sigma'(Result)], normal}$$

First, we show that $T'@m = T@m$ because the operational semantics assigns to $T'@m$ and the rule uses $T@m$. Since the rule is derived applying the invocation rule, and the class rule, we know $T@m = imp(T, m)$ and $\tau(Current) = T$. However, $T'@m = imp(\tau(y), m)$, and we now $\tau(y) = \tau(Current)$, then we can conclude that $T@m = T;@m$.

Then, $\sigma \models P'[false/T@m\_done] \wedge\ T@m\_done$ because $\sigma[T'@m\_done := true, Current := y, p := \sigma(e)] \models P'$.

Now, we can apply the induction hypothesis 30, and we get $\sigma' \models Q_n\ \wedge T@m\_done$. Since $Q_n\ \wedge T@m\_done\ \Rightarrow Q'_n$ then $\sigma' \models Q'_n$.

Case 2: σ(T@m_done) = false and χ = exc. By the definition of the operational semantics we have:

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \textit{ is a once routine} \\ \sigma(T@m\_done) = false \\ \langle \sigma[T@m\_done := true, Current := y, p := \sigma(e)],\ body(T@m)\rangle \rightarrow_N \sigma', exc \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow_{N+1} \sigma'[T@m\_exc := true], exc}$$

Applying a similar reasoning to Case 1, we know $T'@m = T@m$. Since $\sigma \models P'[false/T@m\_done] \wedge T@m\_done$ because $\sigma[T'@m\_done := true, Current := y, p := \sigma(e)] \models P'$, we can apply the induction hypothesis 30, and we get $\sigma' \models Q_e \wedge T@m\_done$. Then $\sigma' \models Q'_e$ because $Q_e \wedge T@m\_done \Rightarrow Q'_e$

Case 3: σ(T@m_done) = true and χ = normal. The definition of the operational semantics is the following:

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

We know $T'@m = T@m$. Since $\sigma \models P$, and the state is unchanged except for the variable x, and $\sigma(T@m\_done) = true$ and $\sigma(T@m\_exc) = false$, then $\sigma \models P''$. Then $\sigma \models Q'_n$.

Case 4: σ(T@m_done) = true and χ = exc. By the definition of the operational semantics we have:

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

We know $T'@m = T@m$. Since $\sigma \models P$, and the state is unchanged, and $\sigma(T@m\_done) = true$ and $\sigma(T@m\_exc) = true$, then $\sigma \models P'''$. Then $\sigma \models Q'_e$.

This concludes the proof.
□

A.2.12 Routine Implementation Rule

To prove this rule, we have to prove:

$$\mathcal{A} \rhd \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}\ \ implies\ \ \mathcal{A} \models \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$$

using the induction hypothesis:

$$\mathcal{A}, \{P\}\quad T@m\quad \{Q_n,\ Q_e\} \rhd \{\ P\ \}\quad body(T@m)\quad \{\ Q_n\ ,\ Q_e\ \}\quad implies$$

$$\mathcal{A}, \{P\}\quad T@m\quad \{Q_n,\ Q_e\} \models \{\ P\ \}\quad body(T@m)\quad \{\ Q_n\ ,\ Q_e\ \}$$

Let $\mathcal{A}$ be the sequent $\mathcal{A} = \mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$for\ all\ N : \models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n\ implies \models_N \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$$

using the hypothesis:

$$\begin{aligned} &for\ all\ N : \models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n,\ and\ \models_N \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \} \\ &\quad implies \\ &\qquad\qquad \models_N \{\ P\ \}\quad body(T@m)\quad \{\ Q_n\ ,\ Q_e\ \} \end{aligned} \tag{31}$$

We prove it by induction on N.
Base Case: $N = 0$. Holds by Definition 5.
Induction Case: $N \Rightarrow N + 1$. Assuming the induction hypothesis

$$\models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n\ implies \models_N \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$$

we have to show

$$\models_{N+1} \mathcal{A}_1, and..., \models_{N+1} \mathcal{A}_n\ implies \models_{N+1} \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$$

Then, we can prove this as follows:

$$\begin{aligned} &\models_{N+1} \mathcal{A}_1, and..., \models_{N+1} \mathcal{A}_n \\ &\qquad\qquad\qquad\qquad implies\ [Lemma\ 6] \\ &\models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n \\ &\qquad\qquad\qquad\qquad applying\ induction\ hypothesis \\ &\models_N \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \} \end{aligned}$$

Using $\models_N\ \mathcal{A}_1, and...,\ \models_N\ \mathcal{A}_n$, and $\models_N\ \{\ P\ \}\quad T@m\quad \{\ Q'_n\ ,\ Q'_e\ \}$, we can apply the hypothesis (31), and we get $\models_N \{\ P\ \}\quad body(T@m)\quad \{\ Q_n\ ,\ Q_e\ \}$. Then, by Definition 5 we prove $\models_{N+1} \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$
□

A.2.13 Routine Invocation Rule

To prove this rule, we have to prove:

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

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

using the induction hypothesis:

$$\mathcal{A} \rhd \{ \; P \; \} \;\; T{:}m \;\; \{ \; Q_n \; , \; Q_e \; \} \;\; implies \;\; \mathcal{A} \models \{ \; P \; \} \;\; T{:}m \;\; \{ \; Q_n \; , \; Q_e \; \}$$

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\begin{array}{l} for \; all \; N : \models_N \mathcal{A}_1, and..., \; \models_N \mathcal{A}_n \\ implies \\ \models_N \left\{ \begin{array}{l} (y \neq Void \wedge P[y/Current, e/p]) \vee \\ (y = Void \wedge Q_e) \end{array} \right\} \; x := y.T{:}m(e) \; \left\{ \; Q_n[x/Result] \; , \; Q_e \; \right\} \end{array}$$

using the hypothesis:

$$for \; all \; N : \models_N \mathcal{A}_1, and..., \; \models_N \mathcal{A}_n \; implies \models_N \{ \; P \; \} \;\; T{:}m \;\; \{ \; Q_n \; , \; Q_e \; \}$$

Let P' be $(y \neq Void \wedge P[y/Current, e/p]) \vee \; (y = Void \wedge Q_e)$. Since the sequent $\mathcal{A}$ is the same in the hypothesis and the conclusion, applying Definition 5, we have to show:

$$\begin{array}{rl} \text{for all } \sigma \models P' : \langle \sigma, \; x := y.T{:}m(e) \rangle \to_N \sigma'', \chi \;\; then & \\ \chi = normal \;\; \Rightarrow \sigma'' \models Q_n[x/Result], \;\; and & \\ \chi = exc \;\; \Rightarrow \sigma'' \models Q_e & \end{array} \tag{32}$$

using the hypothesis:

$$\begin{array}{rl} \text{for all } \sigma \models P : \langle \sigma, \; body(imp(\tau(Current), m)) \rangle \to_{N-1} \sigma', \chi \;\; then & \\ \chi = normal \;\; \Rightarrow \sigma' \models Q_n, \;\; and & \\ \chi = exc \;\; \Rightarrow \sigma' \models Q_e & \end{array} \tag{33}$$

We do case analysis on $\sigma(y)$:
Case 1: $\sigma(\mathbf{y}) = \mathbf{void}$. By the operational semantics we have:

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

Then, $\sigma \models Q_e$ since $\sigma \models P$ and $\chi = exc$.
Case 2: $\sigma(\mathbf{y}) \neq \mathbf{void}$. By the operational semantics we have:

$$\frac{\begin{array}{c} T{:}m \; is \; not \; a \; once \; routine \\ \sigma(y) \neq voidV \qquad \langle \sigma[Current := \sigma(y), p := \sigma(e)], \; body(impl(\tau(\sigma(y)), m)) \rangle \to_N \sigma', \chi \end{array}}{\langle \sigma, x := y.T{:}m(e) \rangle \to_{N+1} \sigma'[x := \sigma'(Result)], \chi}$$

Since $\sigma \models P'$, then applying Lemma 2, $\sigma \models P$. Then since $Current := \sigma(y)$, we can apply the induction hypothesis and Lemma 2 again, and we get
$\chi = normal \;\; \Rightarrow \sigma'' \models Q_n[x/Result], \;\; and$
$\chi = exc \;\; \Rightarrow \sigma'' \models Q_e$
$\square$

A.2.14 Class Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \rhd \{\ \tau(Current) \preceq T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \}\ implies \\ \mathcal{A} \models \{\ \tau(Current) \preceq T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \rhd \{\ \tau(Current) = T\ \wedge\ P\ \}\ \ imp(T,m)\ \ \{\ Q_n\ ,\ Q_e\ \}\quad implies \\ \mathcal{A} \models \{\ \tau(Current) = T\ \wedge\ P\ \}\ \ imp(T,m)\ \ \{\ Q_n\ ,\ Q_e\ \} \\ and \\ \mathcal{A} \rhd \{\ \tau(Current) \prec T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \}\quad implies \\ \mathcal{A} \models \{\ \tau(Current) \prec T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\begin{array}{l} for\ all\ N : \models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n\ implies \\ \qquad \models_N \mathcal{A} \models \{\ \tau(Current) \preceq T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

using the hypotheses:

$$\begin{array}{l} for\ all\ N : \models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n\ implies \\ \qquad \models_N \mathcal{A} \models \{\ \tau(Current) = T\ \wedge\ P\ \}\ \ imp(T,m)\ \ \{\ Q_n\ ,\ Q_e\ \} \\ for\ all\ N : \models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n\ implies \\ \qquad \models_N \mathcal{A} \models \{\ \tau(Current) \prec T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

We prove:

$$\begin{array}{l} \quad \mathcal{A} \rhd \{\ \tau(Current) \preceq T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \\ \Rightarrow [definition\ of\ \tau] \\ \quad \mathcal{A} \rhd \{\ (\tau(Current) \prec T\ \vee\ \tau(Current) = T)\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \\ \Rightarrow [hypothesis] \\ \quad \mathcal{A} \models \{\ \tau(Current) = T\ \wedge\ P\ \}\ \ imp(T,m)\ \ \{\ Q_n\ ,\ Q_e\ \} \\ \quad and \\ \quad \mathcal{A} \models \{\ \tau(Current) \prec T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \\ \Rightarrow [definition\ of\ \models_N] \\ \quad \mathcal{A} \models \{\ \tau(Current) = T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \\ \quad and \\ \quad \mathcal{A} \models \{\ \tau(Current) \prec T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \\ \Rightarrow \\ \quad \mathcal{A} \models \{\ \tau(Current) \preceq T\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \\ \square \end{array}$$

A.2.15 Subtype Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \rhd \{\ \tau(Current) \preceq S\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \}\ implies \\ \mathcal{A} \models \{\ \tau(Current) \preceq S\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \rhd \{\ P\ \}\quad S{:}m\quad \{\ Q_n\ ,\ Q_e\ \}\quad implies\ \mathcal{A} \models \{\ P\ \}\quad S{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \\ S \preceq T \end{array}$$

We have to prove:

$$\begin{array}{l} \mathcal{A} \models \{\ \tau(Current) \preceq S\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \} \\ \textit{iff}\,[\textit{definition of}\ \models_N] \\ \mathcal{A} \models \{\ \tau(Current) \preceq S\ \wedge\ P\ \}\quad imp(\tau(Current), m)\quad \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

Since $\tau(Current) \preceq S$ and from the hypothesis we know $\mathcal{A} \models \{\ P\ \}\quad S{:}m\quad \{\ Q_n\ ,\ Q_e\ \}$, then applying the induction hypothesis we prove:

$$\mathcal{A} \models \{\ \tau(Current) \preceq S\ \wedge\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \}$$

□

A.2.16 Language-Independent Rules

In this subsection, we prove the soundness of the language-independent rules.

Assumpt-axiom

We have to show that $\textit{for all } N : \models_N \mathcal{A}\ \textit{implies}\ \models_N \mathcal{A}$, which is true.

□

False-axiom

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n\ \textit{implies}\ \models_N \{\ \textit{false}\ \}\quad s\quad \{\ \textit{false}\ ,\ \textit{false}\ \}$$

This holds by the definition of $\models_N$.

□

Assumpt-intro-rule

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n, \textit{and}\ \mathbf{A_0}\ \textit{implies}\ \models_N \mathbf{A}$$

using the hypothesis:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n\ \textit{implies}\ \models_N \mathbf{A}$$

This holds by the hypothesis.

□

Assumpt-elim-rule

Let $\mathcal{A}$ be $\mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n\ \textit{implies}\ \models_N \mathbf{A}$$

using the hypotheses:

$$\begin{array}{l} \textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n\ \textit{implies}\ \models_N \mathbf{A_0} \\ \textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n,\ \textit{and}\ \mathbf{A_0}\ \textit{implies}\ \models_N \mathbf{A} \end{array}$$

We prove it as follows:

$$\begin{array}{l} \quad \models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n \quad (1) \\ \Rightarrow [\textit{applying the first hypothesis}] \\ \quad \models_N \mathbf{A_0} \quad (2) \\ \Rightarrow [\textit{applying second hypothesis to } (1) \textit{ and } (2)] \\ \quad \models_N \mathbf{A} \end{array}$$

□

Strength Rule

We have to prove:

$$\mathcal{A} \rhd \{\ P'\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ implies\ \ \mathcal{A} \models \{\ P'\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \rhd \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ implies\ \mathcal{A} \models \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \} \\ and \\ P' \Rightarrow P \end{array}$$

Applying the definition of $\models_N$, we have to prove:

$$\begin{array}{rl} \text{for all } \sigma \models P' : \langle \sigma,\ s_1 \rangle \rightarrow_N \sigma'', \chi & then \\ \chi = normal & \Rightarrow \sigma'' \models Q_n,\ and \\ \chi = exc & \Rightarrow \sigma'' \models Q_e \end{array}$$

Since $P' \Rightarrow P$, then we get $\sigma \models P$, then by hypothesis we prove $\mathcal{A} \models \{\ P'\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}$
□

Weak Rule

We have to prove:

$$\mathcal{A} \rhd \{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}\ \ implies\ \ \mathcal{A} \models \{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \rhd \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ implies\ \mathcal{A} \models \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \} \\ and \\ Q_n \Rightarrow Q'_n \\ Q_e \Rightarrow Q'_e \end{array}$$

Applying the definition of $\models_N$, we have to prove:

$$\begin{array}{rl} \text{for all } \sigma \models P : \langle \sigma,\ s_1 \rangle \rightarrow_N \sigma'', \chi & then \\ \chi = normal & \Rightarrow \sigma'' \models Q'_n,\ and \\ \chi = exc & \Rightarrow \sigma'' \models Q'_e \end{array}$$

Applying the hypothesis we get:

$$\begin{array}{ll} \chi = normal & \Rightarrow \sigma'' \models Q_n,\ and \\ \chi = exc & \Rightarrow \sigma'' \models Q_e \end{array}$$

Since $Q_n \Rightarrow Q'_n$ and $Q_e \Rightarrow Q'_e$ we get then we get

$$\begin{array}{ll} \chi = normal & \Rightarrow \sigma'' \models Q'_n,\ and \\ \chi = exc & \Rightarrow \sigma'' \models Q'_e \end{array}$$

and we prove $\mathcal{A} \models \{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}$
□

Conjunction Rule

We have to prove:

$$\mathcal{A} \rhd \{\ P^1 \wedge P^2\ \}\quad s_1 \quad \{\ Q_n^1 \wedge Q_n^2\ ,\ Q_e^1 \wedge Q_e^2\ \}\ \textit{implies}\ \mathcal{A} \models \{\ P^1 \wedge P^2\ \}\quad s_1 \quad \{\ Q_n^1 \wedge Q_n^2\ ,\ Q_e^1 \wedge Q_e^2\ \}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \rhd \{\ P^1\ \}\quad s_1 \quad \{\ Q_n^1\ ,\ Q_e^1\ \}\quad \textit{implies}\ \mathcal{A} \models \{\ P^1\ \}\quad s_1 \quad \{\ Q_n^1\ ,\ Q_e^1\ \} \\ \mathcal{A} \rhd \{\ P^2\ \}\quad s_1 \quad \{\ Q_n^2\ ,\ Q_e^2\ \}\quad \textit{implies}\ \mathcal{A} \models \{\ P^2\ \}\quad s_1 \quad \{\ Q_n^2\ ,\ Q_e^2\ \} \end{array}$$

This holds applying the definition of $\models_N$, and the hypotheses.
□

Disjunction Rule

The proof is similar to the conjunction rule proof.
□

A.3 Completeness Proof

As pointed out by Oheimb [27], the approach using weakest precondition cannot be used to prove completeness of recursive method calls. The postcondition of recursive method calls changes such that the induction does not go through. Here, we use the *Most General Formula (MGF)* approach introduced by Gorelick [5], and promoted by Apt [1] and others. The *MGF* of a instruction s gives for the most general precondition the strongest poscondition, which is the operational semantics of s.

Following, we prove Theorem 4 by induction on the structure of the instruction s. In this section, we present the proof for the most important cases.

Lemma 7 (Completeness Routine Imp) *Let* $\$$ *and* $\$'$ *be object stores, and let* $\{Q_n^{T@m}, Q_e^{T@m}\}$ *be the strongest postcondition defined as follows:*

$$\{Q_n^{T@m}, Q_e^{T@m}\} \triangleq SP(T@m, \$ = \$')$$

Let $\mathcal{A}_0$ *be the sequent defined as follows:*

$$\mathcal{A}_0 = \bigwedge_{T@m} \{\ \$ = \$'\ \}\quad T@m \quad \{\ Q_n^{T@m}\ ,\ Q_e^{T@m}\ \}$$

$$\textit{If}\quad \models \{\ P\ \}\quad s \quad \{\ Q_n\ ,\ Q_e\ \}\quad \textit{then}\quad \mathcal{A}_0 \rhd \{\ P\ \}\quad s \quad \{\ Q_n\ ,\ Q_e\ \}$$

Before proving Lemma 7, we use it to prove the completeness theorem.

Lemma 8 (Sequent T@m)

$$\mathcal{A}, \left\{\ \$ = \$'\ \right\}\quad T@m \quad \left\{\ Q_n^{T@m}\ ,\ Q_e^{T@m}\ \right\} \rhd \left\{\ P\ \right\}\quad s \quad \left\{\ Q_n\ ,\ Q_e\ \right\}\quad \textit{implies}$$

$$\mathcal{A} \rhd \left\{\ P\ \right\}\quad s \quad \left\{\ Q_n\ ,\ Q_e\ \right\}$$

Now, we prove the completeness theorem:
Proof of Completeness Theorem We have to prove:

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

Assume $\models \{\ P\ \}\quad s \quad \{\ Q_n\ ,\ Q_e\ \}$. Then, applying the Lemma 7 we get:

$$\mathcal{A}_0 \rhd \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

Then by repeated application of Lemma 8 we obtain:

$$\rhd \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

In the rest of this section, we prove Lemma 7 by induction on the measure of s, defined as follows:

- If s is an instruction, the measure is the size of s
- The measure of $T{:}m$ is 0
- The measure of $T@m$ is -1

With this definition of measure, one can reason about instructions using induction hypotheses about their sub-parts, about a routine invocation using induction hypotheses of the form $T{:}m$, and about $T{:}m$ using induction hypotheses of the form $T@m$.

A.3.1 Assignment Axiom

We have to prove:

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

$$\mathcal{A}_0 \rhd \left\{ \begin{array}{l} (\mathit{safe}(e)\ \wedge\ P[e/x]) \vee \\ (\neg \mathit{safe}(e)\ \wedge\ Q_e) \end{array} \right\}\ x\ :=\ e\ \ \{\ P\ ,\ Q_e\ \}$$

Let P' be $(\mathit{safe}(e)\ \wedge\ P[e/x])\ \vee\ (\neg \mathit{safe}(e)\ \wedge\ Q_e)$.

Assume $\models\ \{\ P'\ \}\ \ x\ :=\ e\ \ \{\ Q'_n\ ,\ Q'_e\ \}$, then $\models P' \Rightarrow Q'_n[e/x]$. Then by the assignment axiom and the consequence rule we prove:

$$\mathcal{A}_0 \rhd \{\ P'\ \}\ \ x\ :=\ e\ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

□

A.3.2 Compound Rule

We have to prove:

$$\models \{\ P\ \}\ \ s_1; s_2\ \ \{\ R_n\ ,\ R_e\ \} \Rightarrow \mathcal{A}_0 \rhd \{\ P\ \}\ \ s_1; s_2\ \ \{\ R_n\ ,\ R_e\ \}$$

using the hypotheses

$$\begin{array}{l} \models \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ R_e\ \}\ \ \Rightarrow \mathcal{A}_0 \rhd \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ R_e\ \} \\ \mathit{and} \\ \models \{\ Q_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \}\ \ \Rightarrow \mathcal{A}_0 \rhd \{\ Q_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \} \end{array}$$

Assume $\models \{\ P\ \}\ \ s_1; s_2\ \ \{\ R_n\ ,\ R_e\ \}$. Then

$$\begin{array}{ll} \models \{\ P\ \}\ \ s_1\ \ \{\ T_n\ ,\ T_e\ \} & \mathit{and} \\ \models \{\ T_n\ \}\ \ s_2\ \ \{\ R'_n\ ,\ R'_e\ \} & \end{array}$$

where $\{T_n, T_e\}$ and $\{R'_n, R'_e\}$ are the strongest postconditions defined as follows:

$$\begin{array}{l} \{T_n, T_e\} \triangleq s_1(P) \\ \{R_n, R'_e\} \triangleq s_2(T_n) \end{array}$$

By induction hypotheses we have:

$$\begin{array}{l} \mathcal{A}_0 \rhd \{ \ P \ \} \ \ s_1 \ \ \{ \ T_n \ , \ T_e \ \} \quad \textit{and} \\ \mathcal{A}_0 \rhd \{ \ T_n \ \} \ \ s_2 \ \ \{ \ R'_n \ , \ R'_e \ \} \end{array}$$

By the semantics of $s_1; s_2$ we have that $R'_n \Rightarrow R_n$ and $R'_e \Rightarrow R_e$ and $T_e \Rightarrow R_e$. By the rule of consequence applied with implications $R'_n \Rightarrow R_n$ and $T_e \Rightarrow R_e$ and $R'_e \Rightarrow R_e$, we obtain:

$$\begin{array}{l} \mathcal{A}_0 \rhd \{ \ P \ \} \ \ s_1 \ \ \{ \ T_n \ , \ R_e \ \} \ \textit{and} \\ \mathcal{A}_0 \rhd \{ \ T_n \ \} \ \ s_2 \ \ \{ \ R_n \ , \ R_e \ \} \end{array}$$

The conclusion $\mathcal{A}_0 \rhd \{ \ P \ \} \ \ s_1; s_2 \ \ \{ \ R_n \ , \ R_e \ \}$ follows by the compound rule.
□

A.3.3 Conditional Rule

We have to prove:

$$\begin{array}{l} \models \{ \ P \ \} \ \ \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \ \ \{ \ Q_n \ , \ Q_e \ \} \Rightarrow \\ \mathcal{A}_0 \rhd \{ \ P \ \} \ \ \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \ \ \{ \ Q_n \ , \ Q_e \ \} \end{array}$$

using the hypotheses

$$\begin{array}{ll} \models \{ \ P \ \wedge \ e \ \} \ \ s_1 \ \ \{ \ Q_n \ , \ Q_e \ \} & \Rightarrow \mathcal{A}_0 \rhd \{ \ P \ \wedge \ e \ \} \ \ s_1 \ \ \{ \ Q_n \ , \ Q_e \ \} \\ \textit{and} & \\ \models \{ \ Q_n \ \} \ \ s_2 \ \ \{ \ R_n \ , \ R_e \ \} & \Rightarrow \mathcal{A}_0 \rhd \{ \ P \ \wedge \ \neg e \ \} \ \ s_2 \ \ \{ \ Q_n \ , \ Q_e \ \} \end{array}$$

Assume $\models \{ \ P \ \} \ \ \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \ \ \{ \ Q'_n \ , \ Q'_e \ \}$. Then,

$$\begin{array}{l} \models \{ \ P \ \wedge \ e \ \} \ \ s_1 \ \ \{ \ Q'_n \ , \ Q'_e \ \} \\ \models \{ \ P \ \wedge \ \neg e \ \} \ \ s_2 \ \ \{ \ Q'_n \ , \ Q'_e \ \} \end{array}$$

where $\{T_n, T_e\}$ is the strongest postcondition $\{Q'_n, Q'_e\} \triangleq s_1(P \ \wedge \ e) \cup s_2(P \ \wedge \ \neg e)$
Then by induction hypotheses, we know:

$$\begin{array}{l} \mathcal{A}_0 \rhd \{ \ P \ \wedge \ e \ \} \ \ s_1 \ \ \{ \ Q'_n \ , \ Q'_e \ \} \\ \mathcal{A}_0 \rhd \{ \ P \ \wedge \ \neg e \ \} \ \ s_2 \ \ \{ \ Q'_n \ , \ Q'_e \ \} \end{array}$$

Since $Q'_n \Rightarrow Q_n$ and $Q'_e \Rightarrow Q_e$, applying the conditional rule and the rule of consequence we obtain:

$$\mathcal{A}_0 \rhd \{ \ P \ \} \ \ \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \ \ \{ \ R_n \ , \ R_e \ \}$$

□

A.3.4 Check Axiom

We have to prove:

$$\models \ \{ \ P \ \} \ \ \texttt{check } e \texttt{ end} \ \ \{ \ (P \ \wedge \ e \ ) \ , \ (P \ \wedge \ \neg e \ ) \ \} \qquad \Rightarrow$$

$$\mathcal{A}_0 \rhd \{ \ P \ \} \ \ \texttt{check } e \texttt{ end} \ \ \{ \ (P \ \wedge \ e \ ) \ , \ (P \ \wedge \ \neg e \ ) \ \}$$

Assume $\models \{ \ P \ \} \ \ \texttt{check } e \texttt{ end} \ \ \{ \ Q'_n \ , \ Q'_e \ \}$. For $Q'_n \triangleq (P \wedge e)$ and $Q'_e \triangleq (P \wedge \neg e)$, the conclusion follows by applying the rule of consequence, and the check axiom.
□

A.3.5 Loop Rule

We have to prove:

$$\models \{\ P\ \}\ \texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\ \ \{\ (I\ \wedge\ e)\ ,\ R_e\ \} \Rightarrow$$
$$\mathcal{A}_0 \rhd \{\ P\ \}\ \texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\ \ \{\ (I\ \wedge\ e)\ ,\ R_e\ \}$$

using the hypotheses

$$\models \{\ P\ \}\ \ s_1\ \ \{\ I\ ,\ R_e\ \} \qquad \Rightarrow \mathcal{A}_0 \rhd \{\ P\ \}\ \ s_1\ \ \{\ I\ ,\ R_e\ \}$$
and
$$\models \{\ \neg e\ \wedge\ I\ \ \}\ \ s_2\ \ \{\ I\ ,\ R_e\ \}\ \ \Rightarrow \mathcal{A}_0 \rhd \{\ \neg e\ \wedge\ I\ \ \}\ \ s_2\ \ \{\ I\ ,\ R_e\ \}$$

Assume $\models \{\ P\ \}$ $\texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}$ $\{\ T_n\ ,\ T_e\ \}$

Let $\{P_n^0, R_e'\}$ be the strongest postcondition $\{P_n^0, P_e^0\} \triangleq s_1(P)$. Let $\{P_n^{i+1}, P_e^{i+1}\}$ be the strongest postcondition $\{P_n^{i+1}, P_e^{i+1}\} \triangleq s_2(P_n^i)$. Let I' be the invariant $I' \triangleq \cup_i P_n^i$ and R_e' be $R_e' \triangleq \cup_i P_e^i$. Then by induction hypotheses we get:

$$\mathcal{A}_0 \rhd \{\ P\ \}\ \ s_1\ \ \{\ I'\ ,\ R_e'\ \}$$
$$\mathcal{A}_0 \rhd \{\ \neg e\ \wedge\ I'\ \ \}\ \ s_2\ \ \{\ I'\ ,\ R_e'\ \}$$

Finally, since $I' \triangleq \cup_i P_n^i$ and $R_e' \triangleq \cup_i P_e^i$ then $I' \Rightarrow I$ and $R_e' \Rightarrow R_e$. Then applying the loop rule and the rule of consequence we prove:

$\mathcal{A}_0 \rhd \{\ P\ \}$ $\texttt{from}\ s_1\ \texttt{invariant}\ I\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}$ $\{\ (I \wedge e)\ ,\ R_e\ \}$

□

A.3.6 Read Attribute Axiom

We have to prove:

$$\models \left\{ \begin{array}{l} (y \neq \mathit{Void}\ \wedge\ P[\$(\mathit{instvar}(y, S@a))/x]) \vee \\ (y = \mathit{Void}\ \wedge\ Q_e) \end{array} \right\}\ x := y.S@a\ \ \{\ P\ ,\ Q_e\ \} \qquad \Rightarrow$$

$$\mathcal{A}_0 \rhd \left\{ \begin{array}{l} (y \neq \mathit{Void}\ \wedge\ P[\$(\mathit{instvar}(y, S@a))/x]) \vee \\ (y = \mathit{Void}\ \wedge\ Q_e) \end{array} \right\}\ x := y.S@a\ \ \{\ P\ ,\ Q_e\ \}$$

Let P' be $P' \triangleq (y \neq \mathit{Void}\ \wedge\ P[\$(\mathit{instvar}(y, S@a))/x])\ \vee\ (y = \mathit{Void}\ \wedge\ Q_e)$.

Assume that $\models \{\ P'\ \}\ \ x := y.S@a\ \ \{\ Q_n'\ ,\ Q_e'\ \}$ holds, then

$\models P' \Rightarrow Q_n'[(\mathit{instvar}(y, S@a))/x]$. Then, by the read attribute axiom and the consequence rule, we prove:

$$\mathcal{A}_0 \rhd \{\ P'\ \}\ \ x := y.S@a\ \ \{\ Q_n'\ ,\ Q_e'\ \}$$

□

A.3.7 Write Attribute Axiom

We have to prove:

$$\models \left\{ \begin{array}{l} (y \neq \mathit{Void}\ \wedge\ P[\$ < \mathit{instvar}(y, S@a) := e > /\$]) \vee \\ (y = \mathit{Void}\ \wedge\ Q_e) \end{array} \right\}\ y.S@a := e\ \ \{\ P\ ,\ Q_e\ \} \Rightarrow$$

$$\mathcal{A}_0 \rhd \left\{ \begin{array}{l} (y \neq \mathit{Void}\ \wedge\ P[\$ < \mathit{instvar}(y, S@a) := e > /\$]) \vee \\ (y = \mathit{Void}\ \wedge\ Q_e) \end{array} \right\}\ y.S@a := e\ \ \{\ P\ ,\ Q_e\ \}$$

Let P' be $P' \triangleq (y \neq \mathit{Void}\ \wedge\ P[\$ < \mathit{instvar}(y, S@a) := e > /\$])\ \vee\ (y = \mathit{Void}\ \wedge\ Q_e)$.

Assume $\models \{\ P'\ \}\ \ y.S@a := e\ \ \{\ Q'_n\ ,\ Q'_e\ \}$ *holds*, then $\models P' \Rightarrow Q'_n[\$ < instvar(y, S@a) := e > /\$]$. Then, by the write attribute axiom and the consequence rule, we prove:

$$\mathcal{A}_0 \rhd \{\ P'\ \}\ \ y.S@a := e\ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

□

A.3.8 Local Rule

We have to prove:

$$\begin{array}{l} \models \{\ P\ \}\ \ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ \textit{implies} \\ \mathcal{A}_0 \rhd \{\ P\ \}\ \ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

using the induction hypotheses:

$$\begin{array}{l} \models \{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \} \qquad \textit{implies} \\ \mathcal{A}_0 \rhd \{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

Assume $\models \{\ P\ \}\ \ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\ \ \{\ Q_n\ ,\ Q_e\ \}$, then

$$\models \{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)\ \}\ \ s\ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

where $\{Q'_n, Q'_e\}$ is the strongest postcondition:

$$\{Q'_n, Q'_e\} \triangleq s(P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n))$$

By induction hypothesis we have:

$$\mathcal{A}_0 \rhd \{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)\ \}\ \ s\ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

Since $\{Q'_n, Q'_e\}$ is the strongest postcondition, then $Q_n \Rightarrow Q'_n$ and $Q_e \Rightarrow Q'_e$. Then, by the consequence rule and the local rule, we prove:

$$\mathcal{A}_0 \rhd \{\ P\ \}\ \ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\ \ \{\ Q_n\ ,\ Q_e\ \}$$

□

A.3.9 Rescue Rule

Figure 8 shows a diagram of the states produced by the execution of the rescue clause. The instruction is s_1 `rescue` s_2. The arrow with label s_1 means that the execution of the instruction s_1 starting in the state P^i_n terminates in the state $\{P'^i_n,\ P'^i_e\}$ where P'^i_n is the postcondition after normal termination, and P'^i_e is the postcondition when s_1 triggers an exception. In a similar way, the execution of the instruction s_2 stating in the state P'^i_e terminates in the state $\{Q^i_n,\ Q^i_e\}$ where Q^i_n is the postcondition after normal termination, and Q^i_e is the postcondition when s_1 triggers an exception. If $Retry = True$ then the postcondition Q^i_n implies P^i_n. If $Retry = False$ then the postcondition Q^i_n implies $Q^i_n\ \wedge\ \neg Retry\ \wedge\ Q^i_e$. Furthermore, Q^i_e implies $Q^i_n\ \wedge\ \neg Retry\ \wedge\ Q^i_e$

We have to prove:

$$\models \{\ P\ \}\ \ s_1\ \texttt{rescue}\ s_2\ \ \{\ Q_n\ ,\ R_e\ \} \Rightarrow \mathcal{A}_0 \rhd \{\ P\ \}\ \ s_1\ \texttt{rescue}\ s_2\ \ \{\ Q_n\ ,\ R_e\ \}$$

using the hypotheses

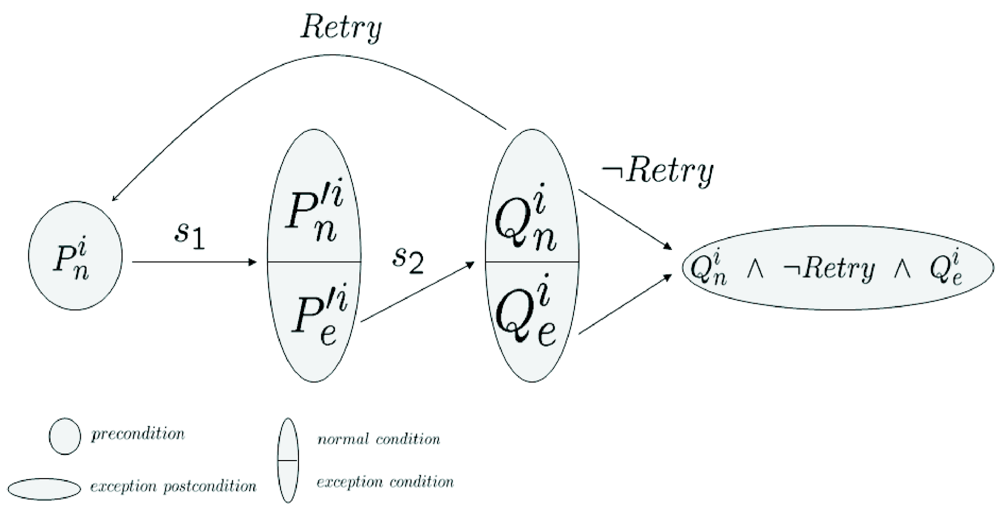

Figure 8: Completeness proof

$$\models \{\ I_r\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \} \Rightarrow \mathcal{A}_0 \rhd \{\ I_r\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}\quad \textit{and}$$

$$\models \{\ Q_e\ \}\ \ s_2\ \ \{\ Retry \Rightarrow I_r\ \wedge\ \neg Retry \Rightarrow R_e\ ,\ R_e\ \}\ \Rightarrow$$
$$\mathcal{A}_0 \rhd \{\ Q_e\ \}\ \ s_2\ \ \{\ Retry \Rightarrow I_r\ \wedge\ \neg Retry \Rightarrow R_e\ ,\ R_e\ \}\quad \textit{and}$$

$$P\ \Rightarrow\ I_r$$

Assume $\models \{\ P\ \}\ \ s_1\ \texttt{rescue}\ s_2\ \ \{\ Q_n\ ,\ Q_e\ \}$. Let

$$\begin{aligned}
P_n^0 &\triangleq P && (34)\\
\{P'^i_n, P'^i_e\} &\triangleq s_1(P_n^i) && (35)\\
\{Q_n^i, Q_e^i\} &\triangleq s_2(P'^i_e) && (36)\\
P_n^{i+1} &\triangleq Q_n^i\ \wedge\ Retry && (37)\\
I_r &\triangleq \cup_i P_n^i && (38)\\
T_n &\triangleq \cup_i P'^i_n && (39)\\
T_e &\triangleq \cup_i P'^i_e && (40)\\
R_e &\triangleq \cup_i((Q_n^i\ \wedge\ \neg Retry)\ \vee Q_e^i) && (41)
\end{aligned}$$

We have $T_n \Rightarrow Q_n$ and $R_e \Rightarrow Q_e$. Then,

$$\models \{\ P_n^i\ \}\ \ s_1\ \ \{\ P'^i_n\ ,\ P'^i_e\ \},\ \textit{and}$$
$$\models \{\ P'^i_e\ \}\ \ s_2\ \ \{\ Q_n^i\ ,\ Q_e^i\ \},\ \textit{for all } i$$

Therefore,

$$\models \{\ \cup_i\ P_n^i\ \}\ \ s_1\ \ \{\ \cup_i\ P'^i_n\ ,\ \cup_i\ P'^i_e\ \}\ \textit{and}$$
$$\models \{\ \cup_i\ P'^i_e\ \}\ \ s_2\ \ \{\ \cup_i\ Q_n^i\ ,\ \cup_i\ Q_e^i\ \}$$

Then by induction hypotheses

$$\mathcal{A}_0 \rhd \{\ \cup_i\ P_n^i\ \}\ \ s_1\ \ \{\ \cup_i\ P'^i_n\ ,\ \cup_i\ P'^i_e\ \}\ \textit{and}$$
$$\mathcal{A}_0 \rhd \{\ \cup_i\ P'^i_e\ \}\ \ s_2\ \ \{\ \cup_i\ Q_n^i\ ,\ \cup_i\ Q_e^i\ \}$$

Since $\cup_i\ P_n^i \Rightarrow P$, and $\cup_i\ P_n'^i \Rightarrow Q_n$, and $\cup_i\ Q_e^i \Rightarrow Q_e$, and the rule of consequence, we get

$$\mathcal{A}_0 \rhd \{\ I_r\ \}\ \ s_1\ \ \{\ Q_n\ ,\ T_e\ \} \tag{42}$$

By (37) we know $Q_n^i \Rightarrow (\mathit{Retry} \Rightarrow P_n^i)$ and by (41) $Q_n^i \Rightarrow ((\neg \mathit{Retry}) \Rightarrow R_e)$. Then, since $I_r = \cup_i P_n^i$ we get $Q_n^i \Rightarrow (\mathit{Retry} \Rightarrow I_r)$ and since $R_e \Rightarrow Q_e$ we get $Q_n^i \Rightarrow ((\neg \mathit{Retry}) \Rightarrow Q_e)$. Then $\cup_i\ Q_n^i \Rightarrow (\mathit{Retry} \Rightarrow I_r \wedge \neg \mathit{Retry} \Rightarrow Q_e$.

Then, by (41) $\cup_i\ Q_e^i \Rightarrow R_e$, $R_e \Rightarrow Q_e$, and the rule of consequence, we prove:

$$\mathcal{A}_0 \rhd \{\ T_e\ \}\ \ s_2\ \ \{\ (\mathit{Retry} \Rightarrow I_r \wedge \neg \mathit{Retry} \Rightarrow Q_e\ ,\ Q_e\ \} \tag{43}$$

To finish the proof, we need to prove $P \Rightarrow I_r$. This holds because $P = P_n^0$ and $I_r = \cup_i P_n^i$. Then from 42 and 43, and applying the rescue rule we get:

$$\mathcal{A}_0 \rhd \{\ P\ \}\ \ s_1\ \texttt{rescue}\ s_2\ \ \{\ Q_n\ ,\ Q_e\ \}$$

□

A.3.10 Routine Implementation Rule

We have to prove:

$$\models \{\ P\ \}\ \ T@m\ \ \{\ Q_n\ ,\ Q_e\ \} \Rightarrow \mathcal{A}_0 \rhd \{\ P\ \}\ \ T@m\ \ \{\ Q_n\ ,\ Q_e\ \}$$

Assume $\models \{\ P\ \}\ \ T@m\ \ \{\ Q_n\ ,\ Q_e\ \}$.

$$\dfrac{\dfrac{\dfrac{\mathcal{A}_0 \rhd \{\ \$ = \$'\ \}\ \ T@m\ \ \{\ Q_n^{T@m}\ ,\ Q_e^{T@m}\ \}}{\mathcal{A}_0 \rhd \{\ \$ = \$' \wedge\ P[\$'/\$]\ \}\ \ T@m\ \ \{\ Q_n^{T@m}\ \wedge\ P[\$'/\$]\ ,\ Q_e^{T@m}\ \wedge\ P[\$'/\$]\ \}}}{\mathcal{A}_0 \rhd \{\ \$ = \$' \wedge\ P[\$'/\$]\ \}\ \ T@m\ \ \{\ Q_n\ ,\ Q_e\ \}}\ (44),\ (45)}{\mathcal{A}_0 \rhd \{\ P\ \}\ \ T@m\ \ \{\ Q_n\ ,\ Q_e\ \}}\ \exists\$'$$

where (44), (45) are the following implications:

$$Q_n^{T@m}\ \wedge\ P[\$'/\$]\ \ \Rightarrow\ \ Q_n \tag{44}$$
$$Q_e^{T@m}\ \wedge\ P[\$'/\$]\ \ \Rightarrow\ \ Q_e \tag{45}$$

A.3.11 Routine Invocation Rule

We have to prove:

$$\models \{\ P\ \}\ \ x := y.T{:}m(e)\ \ \{\ Q_n\ ,\ Q_e\ \} \Rightarrow \mathcal{A}_0 \rhd \{\ P\ \}\ \ x := y.T{:}m(e)\ \ \{\ Q_n\ ,\ Q_e\ \}$$

Assume $\models \{\ P\ \}\ \ x := y.T{:}m(e)\ \ \{\ Q_n\ ,\ Q_e\ \}$. Then, by definition of $\models$ we obtain:

$$\models \{\ P[\mathit{Result}'/\mathit{Result}]\ \}\ \ \mathit{Result} := y.T{:}m(e)\ \ \{\ Q_n[\mathit{Result}'/\mathit{Result}, \mathit{Result}/x]\ ,\ Q_e[\mathit{Result}'/\mathit{Result}]\ \}$$

Let P', Q_n' and Q_e' be the following pre and postconditions:

$$P' \triangleq P \begin{bmatrix} \mathit{Result}'/\mathit{Result}, & \mathit{Current}'/\mathit{Current}, \\ \mathit{Current}/y & \end{bmatrix} \wedge\ p = e$$

$$Q_n' \triangleq Q_n \begin{bmatrix} \mathit{Result}'/\mathit{Result}, & \mathit{Result}/x \\ \mathit{Current}'/\mathit{Current}, & \mathit{Current}/y \end{bmatrix}$$

$$Q_e' \triangleq Q_e \begin{bmatrix} \mathit{Result}'/\mathit{Result}, & \mathit{Current}'/\mathit{Current}, \\ \mathit{Current}/y & \end{bmatrix}$$

By definition of $\models$, we get:

$$\models \{ \ P' \ \} \quad Result := Current.T{:}m(p) \quad \{ \ Q'_n \ , \ Q'_e \ \}$$

Then, by definition of $\models$, we obtain:

$$\models \{ \ P' \ \} \quad T{:}m(p) \quad \{ \ Q'_n \ , \ Q'_e \ \}$$

Then, we obtain the following derivation:

$$\frac{\mathcal{A}_0 \rhd \{ \ P' \ \} \quad T{:}m(p) \quad \{ \ Q'_n \ , \ Q'_e \ \}}{\mathcal{A}_0 \rhd \{ \ P'' \ \} \quad x := y.T{:}m(e) \quad \{ \ Q''_n \ , \ Q''_e \ \}} \text{ invocation rule}$$

where P'', Q''_n, and Q''_e are defined as follows:

$$P'' \triangleq \begin{array}{l} y \neq Void \ \wedge \ P'[y/Current, e/p] \\ y = Void \wedge \ Q'_e[y/Current] \end{array}$$

$$Q''_n \triangleq Q'_n[y/Current, x/Result]$$

$$Q''_e \triangleq Q'_e[y/Current]$$

Unfolding the definition of P', and Q'_e we obtain

$$P'' \triangleq \left(\begin{array}{l} y \neq Void \ \wedge \ P \left[\begin{array}{l} Result'/Result, Current'/Current, \\ Current/y, \ y/Current, \end{array} \right] \wedge \ e = e \ \wedge \\ \\ y = Void \wedge \ Q'_e \left[\begin{array}{l} Current/y, y/Current, \\ Result'/Result, \ Current'/Current \end{array} \right] \end{array} \right)$$

$$\equiv \left(\begin{array}{l} y \neq Void \ \wedge \ P \left[\begin{array}{l} Result'/Result \\ Current'/Current \end{array} \right] \\ \\ y = Void \wedge \ Q'_e \left[\begin{array}{l} Result'/Result \\ Current'/Current \end{array} \right] \end{array} \right)$$

Also, unfolding the definition of Q'_n and Q'_e we know:

$$Q''_n \triangleq Q_n \left[\begin{array}{l} Result'/Result, Current'/Current, \\ Current/y, \ y/Current, \\ Result/x, x/Result \end{array} \right]$$

$$\equiv Q_n \left[\ Result'/Result, Current'/Current \ \right]$$

$$Q''_e \triangleq Q_e \left[\begin{array}{l} Result'/Result, Current'/Current, \\ Current/y, \ y/Current \end{array} \right]$$

$$\equiv Q_e \left[\ Result'/Result, Current'/Current \ \right]$$

Thus, the only replacement used in P'', Q''_n, and Q''_e is $Result'/Result$ and $Current'/Current$. Now applying the *invoc_var_rule* with $Result'$ and $Current'$ we obtain the following derivation:

$$\frac{\mathcal{A}_0 \rhd \{ \ P'' \ \} \quad x := y.T{:}m(p) \quad \{ \ Q''_n \ , \ Q''_e \ \}}{\mathcal{A}_0 \rhd \{ \ (y \neq Void \ \wedge \ P) \vee \ y = Void \wedge \ Q_e \ \} \quad x := y.T{:}m(p) \quad \{ \ Q_n \ , \ Q_e \ \}} \text{ invoc\_var\_rule}$$

Finally, since we know $(P \wedge \; y = \mathit{Void}) \Rightarrow Q_e$ from the hypothesis, applying the rule of consequence we prove:

$$\mathcal{A}_0 \rhd \{ \; P \; \} \quad x := y.T{:}m(e) \quad \{ \; Q_n \; , \; Q_e \; \}$$

A.3.12 Virtual Routines

To prove this case, $T{:}m$, we use the following lemma:

Lemma 9 (Subtypes)

$$\forall T' \preceq T : \; \models \{ \; P \; \wedge \; \tau(\mathit{Current}) = T' \; \} \quad T{:}m \quad \{ \; Q_n \; , \; Q_e \; \} \quad \mathit{then}$$

$$\rhd \{ \; P \; \wedge \; \tau(\mathit{Current}) = T' \; \} \quad T'{:}m \quad \{ \; Q_n \; , \; Q_e \; \}$$

Now, we prove the case $T{:}m$ as follows. We know:

$$\models \{ \; P \; \} \quad T{:}m \quad \{ \; Q_n \; , \; Q_e \; \}$$

Applying Lemma 9 to all descendants of T, and applying the subtype rule and the consequence rule get:

$$\rhd \{ \; P \; \} \quad T{:}m \quad \{ \; Q_n \; , \; Q_e \; \}$$

□

B Appendix: Auxiliary Functions to Support Multiple Inheritance

This section presents the definition of the function *imp*. While $impl(T, m)$ 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.

Given a class declaration list *env* (the list of classes that defines the program), a type t, and a routine r, impl returns the routine implementation where the routine r is defined. To do it, it takes the class t and looks for the routine declaration in t. If r is defined in t then it returns the routine implementation $t@r$; otherwise it searches in all the ancestors of t. The *impl* function is defined as follows:

$$impl :: \ ClassDeclaration\ list \times\ Type \times RoutineId \rightarrow RoutineImp$$

$$\begin{array}{ll} impl\ env\ t\ rId & = \textbf{if}\ (defined\ t\ rId)\ \textbf{then}\ t@rId \\ & \quad \textbf{else}\ (implementation\ env\ (list\_inherits\ env\ t\ rId)) \end{array}$$

The *imp* function is generalized using the function *implementation*. The *implementation* function takes a list of types and routines because a routine could be renamed, and one needs to search for the routine implementation using another routine name. This function is defined as follows:

$$implementation :: \ ClassDeclaration\ list \times ((Type \times RoutineId)\ list)\ \rightarrow RoutineImp$$

$$\begin{array}{ll} implementation\ env\ (t,\ rId)\#xs & = \textbf{if}\ (deep\_defined\ env\ t\ rId)\ \textbf{then} \\ & \qquad (impl\ env\ t\ rId) \\ & \quad \textbf{else}\ (implementation\ env\ xs) \end{array}$$

The function *deep_defined* yields true if only if given a class declaration list *env*, a type t, and a routine r, r is defined in t or in any of its ancestors classes. This function uses the auxiliary function *deep_defined_list* which takes a list of types and routines to handle redefinition. The definitions are as follows:

$$deep\_defined :: \ ClassDeclaration\ list \times ((Type \times RoutineId)\ list)\ \rightarrow Bool$$

$$\begin{array}{ll} deep\_defined\ env\ cDecl\ rId & = undefined \\ deep\_defined\ env\ cDecl\ rId & = \textbf{if}\ (defined\ cDecl\ rId)\ \textbf{then}\ True \\ & \quad \textbf{else}\ (deep\_defined\_list\ env\ (list\_inherits\ env\ cDecl\ rId)) \end{array}$$

$$deep\_defined\_list :: \ ClassDeclaration\ list \times ((Type \times RoutineId)\ list)\ \rightarrow Bool$$

$$\begin{array}{ll} deep\_defined\_list\ env\ [\,] & = False \\ deep\_defined\_list\ env\ (t,\ rId)\#xs & = (deep\_defined\ env\ t\ rId) \vee \\ & \quad (deep\_defined\_listenv\ xs) \end{array}$$

Given a type t, and a routine r, the function *list_inherits* yields a list of the parents classes and routines where the routine r might be defined. This functions considers renaming and undefining of routines. Its definition is the following:

$$list\_inherits :: \ ClassDeclaration\ list \times Type \times RoutineId\ \rightarrow\ (Type\ \times\ RoutineId)\ list$$

$$\begin{array}{ll} list\_inherits\ [\,]\ t\ rId & = [\,] \\ list\_inherits\ env\ t\ rId & = (list\_inh\ env\ (parents\ t)\ rId) \end{array}$$

Given a list of class declarations *env*, an inheritance clause *inh*, and a routine r, the function *listInh* yields a list of types and routines where the routine r might be defined. If the routine is undefined in the parent class, the function does not search its implementation. If the routine is renamed, it searches for the new routine name. This function is defines as follows:

$$
\begin{array}{l}
\mathit{list\_inh} :: \mathit{ClassDeclaration\ list} \rightarrow \mathit{Inheritance} \rightarrow \mathit{RoutineId} \rightarrow \\
\quad ((\mathit{ClassDeclaration}\ \times \mathit{RoutineId})\ \mathit{list})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{list\_inh}\ \mathit{env}\ [\,]\ \mathit{rId} & = [\,] \\
\mathit{list\_inh}\ \mathit{env}\ ((t1,\ (\mathit{undef}, \mathit{redef}, \mathit{rename}))\#\mathit{xs})\ \mathit{rId} & = \\
\qquad \textbf{if}\ (\mathit{isUndefined}\ \mathit{undef}\ \mathit{rId})\ \textbf{then} & \\
\qquad\quad (\mathit{listInh}\ \mathit{env}\ \mathit{xs}\ \mathit{rId}) & \\
\qquad \textbf{else}\ (\mathit{renamed}_t\mathit{ypeenv}\ t1\ \mathit{rename}\ \mathit{rId})\#(\mathit{list\_inh}\ \mathit{env}\ \mathit{xs}\ \mathit{rId}) &
\end{array}
$$

where the function *is_undefined* yields true if the routine is undefined in the inheritance clause, and the function *renamed_type* yields the name of the routine considering renaming (if the routine r is not renamed, it yields the same routine r).