Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D3.4

Report on Logic for Resources and Information Flow

Due date of deliverable: 2007-09-01 (T0+24)

Actual submission date: 2007-10-09

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **UEDIN**

Creation

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Acknowledgement

Site	**Contributed to**
UEDIN	Chapter 1, Chapter 6
CTH	Chapter 3
WU	Chapter 2
LMU	Chapter 4, Chapter 5

Executive Summary:
Task 3.2: Logic for Resources and Information Flow

This document summarises deliverable D3.4 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr`.

A significant basis for formal reasoning about Java bytecode is developed in MOBIUS deliverable D3.1 [34]. The job of the current task is to show how this basis can be used (with some extension) to reason about information flow properties and resource properties. The prototypical example of an information flow property is non-interference: high security data cannot be observed through low security fields. Resource properties include, e.g. maximum heap space used, total number of bytecode instructions executed or number of calls to a particular method. As *ghost variables* are heavily used in the Java Modelling Language (JML, the high-level specification language adopted by MOBIUS) for specifying resource usage as well as functional properties, this task also addresses semantics, modelling, and reasoning about ghost variables.

For purposes of this task, the two most important parts of the MOBIUS reasoning basis are *Bicolano*, a model of the Java Virtual Machine (JVM), and the MOBIUS *base logic*, a VDM-style program logic for JVM bytecode. Bicolano supplies an abstract model of the JVM state (heap, stack, loaded program, program counter) as certain Coq types, and the base logic reasons about Coq formulas making assertions about this abstract state. It is clear that state information is not enough to reason about all resource properties that might be interesting. In particular properties that require accumulation of information over a program execution, or more generally a trace of the history of execution, will require some extension to Bicolano.

Chapter 2 presents two kinds of extension mechanism for Bicolano. *Horizontal* extension allows information about program execution to be accumulated, up to a full program trace. For a particular horizontal extension Bicolano must be supplied with bookkeeping functions that accumulate the necessary information in a special state field. Importantly, the design of this extension mechanism guarantees that it is safe; any horizontal extension does not affect the transition behaviour of the Bicolano model of JVM execution. This is accomplished through the Coq module system by Coq typing of the bookkeeping functions. Conversely, *vertical* extension is designed to allow modular and cumulative extensions to Bicolano transition behaviour. This mechanism is more general than the horizontal framework. For example, while the horizontal framework can keep track of the maximum heap space used in a program execution, if one wanted to model raising the `OutOfMemoryError` exception when a certain limit is exceeded, the vertical framework must be used (Bicolano assumes infinite heap and stack). Since vertical extension can change the step relation of Bicolano, there is no automatic guarantee that the model behaves as expected when using a vertical extension; a person making such an extension must check its correctness.

The prototypical information flow property is non-interference. This says that some parts of the machine state (thought of as publicly visible information) are never observed to depend on some other parts of the machine state (thought of as secret information). Chapter 3 discusses how to express non-interference in MOBIUS base logic. The basic idea is to use quantifier alternation in the Coq meta-language to express data dependencies in a Coq formula containing judgement(s) of the base logic. Chapter 3 also outlines a refinement of this basic approach producing goals that are more convenient to prove than the basic approach. This refinement uses the horizontal extension framework of Bicolano described in Chapter 2 to track field dependencies.

The original rationale behind the horizontal extension framework of Bicolano is to support extension of the MOBIUS base logic for reasoning about resource usage; this is discussed in Chapter 4. In order to reason about the new state fields of extended Bicolano, the assertions of base logic judgements are given access to the full extended state information of Bicolano. Assertions used in the base logic to reason about a resource property must be tailored to access the structure of the extended Bicolano state created by the particular horizontal extension. Since horizontal extension cannot affect transition bahaviour of the model, the extended base logic is proved sound w.r.t Bicolano once and for all, extending the soundness proof from deliverable D3.1 [34].

Ghost variables are convenient for specification, and are heavily used in the Java Modelling Language. Chapter 5 uses a simplified language, operational semantics and program logic to give formal meaning to ghost variables annotating functional properties. Using a formal notion of *event*, the use of ghost variables to monitor intentional behaviour (e.g. heap allocation or calls to particular library methods) is also explained. It is then shown how ghost variables not occurring in external (method) specifications can be mechanically eliminated. Ghost variables are thus seen as a conservative extension of the basic program logic.

Contents

Chapter 1

Introduction

MOBIUS Deliverable 7.2, *Next 18 Months' Work Program* [?], outlines the job of the current task:

> The aim of this task is to further develop the MOBIUS base logic for bytecode and bytecode semantics of Task 3.1 towards meeting the security requirements reported in Deliverable 1.1.
>
> The central objective supporting this is, in response to the results of Tasks 1.1 and 1.2, to investigate applications of the bytecode logic and semantics to reasoning about resource usage and information flow. [...]
>
> A contributing objective for the task is to package the Coq definitions, proofs, and working documentation for the bytecode semantics in a formal release that makes it readily available to other partners across the MOBIUS project.

The prototypical example of an information flow property mentioned in Deliverable 1.1 is non-interference (high security data cannot be observed through low security fields). Resource properties mentioned in Deliverable 1.1 include maximum heap space used and number of calls to a particular method. In this document we show how to extend the MOBIUS base logic and bytecode semantics so as to be able to express and reason about non-interference and resource properties. This development is carried out in the Coq formalisation of the operational semantics and base logic as described in MOBIUS Deliverable D3.1 [34].

As *ghost variables* are heavily used in the Java Modelling Language (JML, the high-level specification language adopted by MOBIUS) for specifying resource usage as well as functional properties, this report also addresses semantics, modelling, and reasoning about ghost variables.

1.1 The MOBIUS reasoning basis

A significant basis for formal reasoning about Java bytecode is developed in MOBIUS Deliverable D3.1 [34]. For purposes of this task, the two most important parts of the MOBIUS reasoning basis are *Bicolano*, a model of the Java Virtual Machine (JVM), and the MOBIUS *base logic*, a VDM-style program logic for JVM bytecode. These are described in detail in MOBIUS Deliverable D3.1 [34], including references to the Coq formalization of Bicolano, and the Coq reference implementation of the base logic. In this section we only point out some features relevant to the present task.

Bicolano is an abstract (as opposed to executable) model of the JVM. It provides (abstract) representations of the machine state: heap, stack, loaded program and program counter. It defines various relations expressing "small step" and "big step" transitions of the JVM. For example, the small step semantics, (implicitly paramaterised by a program considered to be already loaded in the machine) is a subset of $\mathsf{state} \times \mathsf{state}$. This is encoded in Coq as a relation of type (roughly) $\mathsf{state} \to \mathsf{state} \to \mathsf{Prop}$.

The MOBIUS base logic is *shallowly* embedded in Coq; this means that the judgements are Coq relations over the various Coq data structures that make up the Bicolano state. The base logic has judgements of shape

$$G, Q \vdash \{A\}pc\{B\}(I) \tag{1.1}$$

where *pc* is a location in the program, G a context of recursive proof assumptions as needed for verification of loops, and Q, A, B and I are *assertions* of various forms. Assertions have direct access to Bicolano state. E.g. A has Coq type

$$\mathit{InitState} \times \mathit{LocalState} \rightarrow \mathsf{Prop}$$

(where *InitState* and *LocalState* are Coq structures constructed, used and maintained by Bicolano, whose exact contents is unimportant for this introduction). Similarly Q, B and I have direct access to various components of the Bicolano state. Furthermore, the judgement form (1.1) itself is a Coq formula.

1.2 Extensions of Bicolano

Both reasoning about information flow and reasoning about resource usage suggest the need to extend Bicolano. Two kinds of extension mechanism are detailed in Chapter 2.

- *Horizontal* extension allows maintaining resource information, and other information up to full execution traces. Importantly, the design of this extension mechanism precludes the stored information from influencing the transition behaviour of the machine.

- *Vertical* extension, conversely, is designed to allow modular and cumulative incremental extensions to Bicolano transition behaviour.

We outline the ideas here.

1.2.1 Horizontal extension

Having access to the Bicolano state, program specifications can already mention some resource properties, such as the size of the operand stack. However, Bicolano is abstract, and its state can only be examined via the supplied accessor functions. Further, one may want to specify resource properties that are not captured by the state at all, such as the number of calls to a particular method. Thus it was necessary to extend the Bicolano state with a new auxiliary (or "ghost field") called ACT (explained in section 2.3) for recording arbitrary information as required to enforce particular resource policies. By combining information into a single structure (e.g. a sum of products), any number of resource fields can be accomodated. Also, the data is not restricted to resource annotations, but can be used for general semantic ghost variables, including full execution traces.

To update the information in this auxiliary field, some bookeeping functions must be provided for Bicolano tailored to maintain the particular resource information needed for a desired resource policy. These bookeeping functions must fit an abstract interface that the extended Bicolano presents (supported by the Coq module system), which prevents them from influencing any part of the Bicolano state which the step relations of Bicolano may observe. Thus, by design, the accumulation of resource information, and other ghost variables, cannot influence the stepping behaviour (abstract execution) of the Bicolano model.[1] It follows from this that the base logic can be extended and proved sound for horizontal extensions once and for all (this is discussed below).

Horizontal extensions may be combined (e.g. by cartesian product), but they are not incremental.

1.2.2 Vertical extension

The vertical extension framework is more general than the horizontal framework. By design it allows extension of the state that the abstract step relation may examine, and change of the step relation itself. For example, while the horizontal framework can keep track of the maximum heap space used in a program execution, if one wanted to model raising the `OutOfMemoryError` exception when a certain limit is exceeded,

[1]This is a meta-statement, whose truth depends on correctness of the Coq module system; there is no formal statement of this claim.

the vertical framework must be used (Bicolano assumes infinite heap and stack). Since Bicolano models a complicated structure (the JVM), and is part of the *trusted computing base*,[2] it is desirable to accumulate relatively small incremental extensions, and the vertical framework supports this.

Since vertical extension can change the step relation of Bicolano, there is no automatic guarantee that the model behaves as expected when using a vertical extension; a person making such an extension must check its correctness. Furthermore, to reason about a vertical extension, one must extend the base logic to match the Bicolano extension, and should prove that the extended base logic is still sound w.r.t. the extended Bicolano.

1.3 Logic for information flow

The prototypical information flow property is non-interference. This says that some parts of the machine state (thought of as the publicly visible information) are never observed to depend on some other parts of the machine state (thought of as the secret information). There are many issues hidden in this statement, and the idea can be generalized in many ways (see MOBIUS Deliverable D1.1 [33] for discussion). Chapter 3 discusses how to express the basic non-interference property in MOBIUS base logic.

In outline, the idea is to use quantifier alternation in the Coq meta-language to express data dependencies in a Coq formula. Assume there are publicly visible heap locations, l (low security), and private heap locations, h (high security). One can express that the values of l in the return state S_r are independent of the values of h in the initial state S_0 by the formula

$$\forall\, v.\ \exists\, r.\ \forall\, u.\ (S_0(l) = v \wedge S_0(h) = u) \quad \Rightarrow \quad S_r(l) = r$$

saying that r (the returned values of public locations) is a function of v (the initial values of public locations) by Skolemization, but is independent of u (the initial values of private locations). This formula cannot be stated as a judgement in the base logic because of the initial quantifiers, but is a formula about base logic judgement(s) in the Coq language, which servs as a meta-language.

Section 3.3 outlines a refinement of this basic approach producing goals that are more convenient to prove than the basic approach. This refinement uses the horizontal extension framework of Bicolano described in Chapter 2 to track field dependencies.

1.4 Logic for resources

The original rationale behind the horizontal extension framework of Bicolano is to support extension of the MOBIUS base logic for reasoning about resource usage; this is discussed in Chapter 4. In order to reason about the ACT field of extended Bicolano (section 1.2), the assertions of base logic judgements, A, B, I and Q, are given access to the full extended state information of Bicolano, including the ACT field. To reason about any particular resources, it is necessary to supply appropriate bookeeping functions for Bicolano to accumulate the necessary information. Assertions used in the base logic to reason about a program must be tailored to access the particular ACT structure created by these bookeeping functions. (An assertion will be rejected by Coq as ill-typed if it doesn't respect the correct structure of the ACT field.) There are also some purely technical changes to the base logic, since it must know how to correctly examine the Bicolano state to compute the weakest precondition of the various assertions. Finally, since well-typed bookeeping functions cannot change any of the Bicolano state that the step relations examine, the extended base logic is proved sound w.r.t Bicolano once and for all, extending the soundness proof from Deliverable D3.1 [34].

[2]Since the official specification of the JVM is *informal*, it is unavoidable that there be some foundation to MOBIUS that is trusted (without formal proof) to bridge the informal to the formal.

1.5 Elimination of ghost variables

Ghost variables are convenient for specification, and are heavily used in the Java Modelling Language. In Chapters 2 and 4 we have shown how to reason about “ghost variables” which can occur in method specifications and in local assertions, and whose values cannot influence the behaviour of the machine. But the meaning of assertions containing ghost variables for verification of program behaviour is not completely satisfactory. For example, if a ghost variable is used to count the number of times a particular method is called, one must read the entire program to check that every call to that method actually increments the ghost variable; there is no way to formally state or verify this property. Chapter 5 addresses this problem. Using a simplified language, operational semantics and program logic it gives formal meaning to ghost variables annotating functional properties. Using a formal notion of *event*, the use of ghost variables to monitor intentional behaviour (e.g. heap allocation or calls to particular methods) is also explained. It is then shown how ghost variables not occurring in external (method) specifications can be mechanically eliminated. Ghost variables are thus seen as a conservative extension of the basic program logic.

Chapter 2

Extensions frameworks for Bicolano

2.1 Introduction

The existence of widely deployable global computing frameworks requires mechanisms to guarantee safety of the user's assets, even though the executed software is obtained through an insecure channel. Proof-carrying code (PCC) [37] is a powerful concept that allows to verify the properties of programs in executable form (machine code, bytecode) and in this way ensure their safety. A PCC system which is able to guarantee complicated features of programs should be based on the foundational PCC introduced by Appel and Felty [2]. In this case, verification is possible as the semantics of the programs can be expressed in a logical formalism. The formulae in such a formalism express the desired properties of programs.

One of the possibilities to provide a framework of foundational proof-carrying code for the Java platform is to formalise the semantics of the Java Virtual Machine (JVM) and the Java bytecode language in a proof checker. The logic of the proof checker can then be used as a formalism to express program properties. One existing JVM formalisation is Bicolano [40]. It is a formalisation of JVM in the Coq proof assistant [17, 14] which covers a considerable subset of the instructions (72 out of 142) and handles aspects such as existence of multiple classes, inheritance, method invocation and exceptions [34]. Bicolano was designed to serve as an element of a PCC architecture for mobile devices in which the program properties are expressed in Coq logic and the certificates are either Coq proof scripts or Coq proof terms (or both). In this light, the semantics should explicitly formalise JVM instructions, rather than to provide their abstraction like in the work by Klein and Nipkow [27]. This approach allows to directly translate the bytecode programs into Coq values while the program properties and proofs are expressed directly within the Coq format. This provides a strong trusted computing base which relies on the Coq typechecker (which is relatively small), Bicolano, and a small set of simple utilities to translate bytecode program representation to a Coq representation.

The current version of Bicolano covers a set of JVM features which enables certification of a wide range of programs. Still, it is not easy to reason about some properties which are of interest, even for restricted versions of the JVM such as the CLDC/MIDP mobile platform [43]. The existing formalisation is idealised in certain ways, for instance the available heap memory and method stack are unlimited. Moreover, it is not easy to express properties in terms of the actual program runs, e.g. to trace memory usage throughout program runs. In this light, it is very likely that Bicolano semantics will have to be adjusted in certain ways to accommodate better the particular verification needs.

Adding additional features to the semantics is not straightforward. The reliability of Bicolano semantics has been checked in two ways. First, the semantics was developed in two flavours: small-step and big-step, and the big-step semantics was proved to be sound with regard to the small-step one. Second, a separate group of people checked that the code of Bicolano indeed obeys the descriptions in the JVM specification [31]. Moreover, the formalisation and soundness proofs are large for a trusted code base (over 11 thousand lines). Therefore, changes to the Bicolano formalisation must be done with caution. In order to guarantee that additions do not destroy the already achieved guarantees we developed two different extension frameworks for Bicolano semantics: *the horizontal extension framework* and *vertical extension framework*. They address different needs and can be used for different purposes.

The horizontal extension framework has a primary goal to provide a natural basis for reasoning about the traces of bytecode program runs. The idea of the framework is based on the concept of types with effects (see e.g. the overview by Nielson and Nielson [38]). In this scheme, a typing discipline can be augmented with an additional property called an effect. Each function signature, in addition to the usual typing information, comes equipped with information about the abstract effect execution of the function has on the executing environment (e.g. which variables are assigned [44], what is the memory consumption [45] etc.). In an extreme case the effect can contain the full history of a computation [42]. Consequently, this framework can naturally serve to express and specify the properties of program runs. At the same time this framework is not suitable for extending the behaviour of the semantics since the collecting of the effects is defined so that it is impossible to affect the state manipulated by the program instructions. The construction of this extension required the modification of the existing Bicolano semantics so it was essential to constrain the modifications (using the Coq module system) so that they do not impact the trust already gained.

The vertical extension framework has a primary goal to introduce additional behaviours that were omitted from the original Bicolano semantical description. We assume that the additions are incremental in their nature — they usually concern a limited number of instructions (for instance in order to add the memory counting we have to modify the `new` instructions only). Consequently, we assume also that in for most instructions the original Bicolano semantics is sufficient. These assumptions have led to a design in which we retain the original state and admit additional state components which allow to formalise the supplementary behaviours that cannot be expressed in terms of the original heap and stack. Furthermore, we refer to the original semantics to obtain the state transformation as in the original Bicolano (together with the previous extensions). This arrangement is different from other approaches to the modular semantics such as the modular monadic semantics [30] or action semantics [36] since the other frameworks assume that the original semantics is right from the beginning designed to be extended in one of these frameworks. These modular semantics express the programming language concepts in terms of atomic operations that are inherent to their construct. In case of Bicolano, we do not want to rewrite the existing formalisation as this is a very costly operation.

Both extensions are formalised in the small step semantics; the horizontal extension framework is formalised in big step semantics as well. We choose to formalise the small step semantics because certain extensions are very difficult to formalise by means of big step semantics; in particular extensions with multithreaded execution or non-deterministic errors of the virtual machine. We formalised the horizontal framework in the big step flavour as well because this kind of semantics is required to provide the basis for extension of the MOBIUS base logic

Examples In order to demonstrate the usability of the extensions, we provided a few examples. In both frameworks, we develop an extension that counts the instructions executed in a program run and an extension that tracks the memory usage in a program run. Throughout this paper, we use the memory usage tracking extension as an example which illustrates the concepts and the design of the extension frameworks. Together with the memory tracking extension, we sketch here a more involved example of the extension that allows to model so called ghost variables [32, 28]. Ghost variables are variables that can be modified like ordinary variables, but only specifications can refer to them — it is impossible to examine their value in the running code of a program. Usually, ghost variables serve as a way to describe certain aspects of the computation history. (See Chapter 5 for discussion of ghost varoables.)

Overview

The chapter is organised as follows. Section 2.2 presents Bicolano architecture and the details of the semantics which are used in the rest of the chapter. Section 2.3 presents the horizontal extension framework and Section 2.4 presents the vertical extension framework. In Section 2.5 we compare the frameworks, and we conclude in Section 2.6.

2.2 Preliminaries

In this section we present the details of Bicolano necessary to understand the formalisation of the proposed extension frameworks. Bicolano is a description of the Java Virtual Machine (JVM) formalised in the Coq proof assistant. It is situated at the bottom of the trusted code base of the MOBIUS project and serves as a base for the soundness proof for the MOBIUS base logic. Bicolano itself has no formal soundness proof as the official JVM specification is in natural language only. Trustworthiness is supported by the fact that the semantics is defined in the big step and small step fashion and the two formalisations are proved to be equivalent.

2.2.1 Bicolano architecture

Bicolano consists of three main parts, two of which are used directly by the extensions frameworks.

Axiomatic base

The axiomatic base of Bicolano specifies the basic terms used in the formalisation, in particular terms specific for the JVM and bytecode programs. The extensions frameworks directly refer to module signatures `PROGRAM` and `SEMANTIC_DOMAIN`. Beside these two, the axiomatic base contains a signature specifying JVM arithmetic.

The `PROGRAM` signature defines the syntax of Java bytecode programs in Coq. Most terms are declared as abstract types and are equipped with operations and axioms specifying their properties. Such terms include class and method names, the program counter (instruction addressing), local variable indices, methods, classes and programs. The declared operations allow to decompose a given program into classes, a class into methods and a method into instructions. It is also possible to retrieve a class or a method by its name.

The JVM instructions are defined as an inductive type. Many JVM instructions have been parametrised and in some cases one constructor in Bicolano represents many JVM instructions (e.g. `If_icmp`, parametrised with a comparison operator, represents many conditional jump JVM instructions).

The `SEMANTIC_DOMAIN` signature defines the domain of the semantics, that is the values needed to describe a state of a running Java Virtual Machine. As in `PROGRAM`, many values are declared as abstract types supplied with operations and axioms. Among these values are types that model local variables, arrays of local variables, the heap and locations. More complex structures are defined here as products or via inductive type definitions. These structures include:

- values of different kinds,
- operand stack (a list of values),
- frames (normal or exceptional),
- call stack (a list of frames),
- the whole state (normal or exceptional).

A normal frame consists of:

- a method,
- a program counter,
- an operand stack,

- a local variables array.

An exceptional frame consists of:

- a method,
- a program counter,
- a location (of the exception object),
- an array of local variables.

It is worth to underline that a state contains the current instruction address and even the current method (as well as methods and addresses of all the methods called throughout the call stack).

A (normal or exceptional) state consists of:

- the heap,
- the current frame (normal or exceptional, respectively),
- the call stack (list of normal frames).

Exceptional states appear when an exception is thrown, but not yet handled by the exception handler mechanism.

Operational semantics

The operational semantics is provided in two flavours: small step semantics and big step semantics. The main objective for the small step semantics is to be as close to the original JVM specification as possible. One step corresponds to execution of a single JVM instruction, as described in the JVM specification. A special case is a step when an exception is raised but not caught within a method and the method returns with an exception.

The small `step` relation has the following type:

```
Program -> State.t -> State.t -> Prop
```

It holds for a program `p` and states `st1` and `st2` if and only if `st1` and `st2` are consecutive states when the program `p` is run. It is worth mentioning that for a given state `st1`, there may be no state `st2` such that `step` holds. In such cases, which should not occur for proper programs, we say that the program *gets stuck*. Note that the current version of Bicolano (without multithreading) is deterministic.

The big step semantics has been provided for two main reasons:

- as an internal verification of the semantics formalisation,
- for use in MOBIUS base logic, which has assertions that hold at the end of a big step, if and when a method terminates (see [34] for details on this point).

The big step semantics defines several kinds of states and steps:

- `IntraNormalState` and `NormalStep`,
- `IntraExceptionState` and `ExceptionStep`,
- InitCallState and `CallStep`,
- `ReturnState` and `ReturnStep`.

This formalisation of the semantics is more complicated and does not allow to formalise certain behaviours, such as the interleaving of multiple threads.

2.3 The horizontal extension framework

The primary motivation for this extension framework was to enable tracing resource usage in runs of programs. This is achieved by adding a ghost field to the state of Bicolano, and a modular interface to *bookkeeping functions* which are called by Bicolano transitions to update this ghost field. Particular horizontal extensions must give appropriate bookkeeping functions fitting this modular interface. The addition of this effect information to the Bicolano semantics enables to extend the MOBIUS base logic [13] with primitives to reason about the resource consumption of programs, since the framework has been developed with big step semantics. This Bicolano ghost field also supports ghost variables in bytecode.

2.3.1 How to use the framework?

Small step semantics In order to exploit the extension framework one has to define a Coq module for the following module type:

```
Module Type SS_EFFECT.
  Declare Module Dom: SEMANTIC_DOMAIN.
  Import Dom Prog.

  Parameter ACT: Set.

  Parameter bookkeepSmall: Program ->
                           State.t -> State.t -> ACT -> ACT.
End SS_EFFECT.
```

The parameter `ACT` (short for ACtion Trace) determines which information is collected along a run of a bytecode program. It keeps track of the actions we are interested in that were executed in the program in question. An arbitrary type can be assigned to it, depending on the actual need. In case of the extension which traces the memory usage, the parameter may be instantiated to natural numbers `nat`. This additional component represents the current size of the heap memory. In case of the extension that models the ghost variables the parameter may be instantiated to the second state component augmented with the information on where the ghost variables are updated. In this way, all the ghost variables are kept in this supplementary state and the running program cannot refer to them.

The parameter `bookkeepSmall` is a function whose arguments are:

- an abstract program identifier,
- a state before the current instruction of the program is executed,
- a state after the current instruction of the program is executed,
- the value of `ACT` before the current instruction of the program is executed.

`bookkeepSmall` returns the resulting value of `ACT`.

In case of the extension which traces the memory usage the `bookkeepSmall` should determine if the instruction to be executed is either `new` or `newarray` and then add the size of the object or the array being created to the current size of the heap memory contained in the `ACT` parameter and the result of the addition should be returned as the updated `ACT`.

In case of the extension with ghost variables, we extract from the `ACT` information which ghost variable should be updated for the current step and do the update accordingly to return the new `ACT`.

Big step semantics In order to exploit the extension framework in case of the big step semantics, one has to provide definitions for a module type BS_EFFECT, similar in concept to SS_EFFECT. Instead of a single bookkeepSmall one has to define bookkeeping functions for each of the basic kinds of big steps:

- bookkeepNormal — for normal internal steps,
- bookkeepException — for exception steps,
- bookkeepCall — for call steps,
- bookkeepReturn — for return steps.

The bookkeeping functions should be used to specify the changes in the ACT part of the state that happens in a semantics step. Their types are similar to the type of bookkeepSmall. For example the type of bookkeepException is:

```
Parameter bookkeepException:
        Program -> Method -> _IntraNormalState ->
        _IntraExceptionState -> ACT -> ACT.
```

Note that the returned value of ACT depends not only on the program, but also on the method being executed, the state before the execution of an instruction (_IntraNormalState) and the state after the execution of the instruction (_IntraExceptionState; the resulting state is exceptional).

In order to define the extension which traces the memory usage, one has to again instantiate ACT with nat and define all the bookkeeping functions above. However, only bookkeepNormal should do non-trivial work. It should check if the current program counter points to a new or newarray bytecode instruction and in that case it should increase the size of the used memory traced in the ACT type.

In order to define the extension with ghost variables, one has to instantiate ACT with the disjoint union of the big step states augmented with the information on where the ghost variables are updated. Each of the bookkeeping functions must update the value of the program counter in the ACT component and the functions must also update the heap for the ghost variables according to the ghost update instructions.

We emphasise that this extension framework is designed such that the bookkeeping functions cannot influence the original Bicolano state, since ACT is orthogonal to this state. Thus horizontal extensions cannot influence Bicolano transitions.

2.3.2 The realisation of the framework

State

The small step semantics in the new version of Bicolano defines the EState module which encapsulates the old State combined with the action traces in ACT.

In case of the big step semantics the types of state have been extended with the ACT component while the original state types have been renamed by adding the underscore _ as a prefix. For example the old IntraNormalState is now _IntraNormalState while the current IntraNormalState is defined to be _IntraNormalState*ACT.

Semantics

In case of the small step semantics, the old relation step is renamed to _step. The name step is used to denote the new version of the semantical step which takes into account the bookkeeping. The definition of the step looks as follows:

```
Inductive step (p:Program): EState.t -> EState.t -> Prop :=
| ENS_ : forall s t S T,
      _step p s t -> T = bookkeepSmall p s t S -> step p (s,S) (t,T).
```

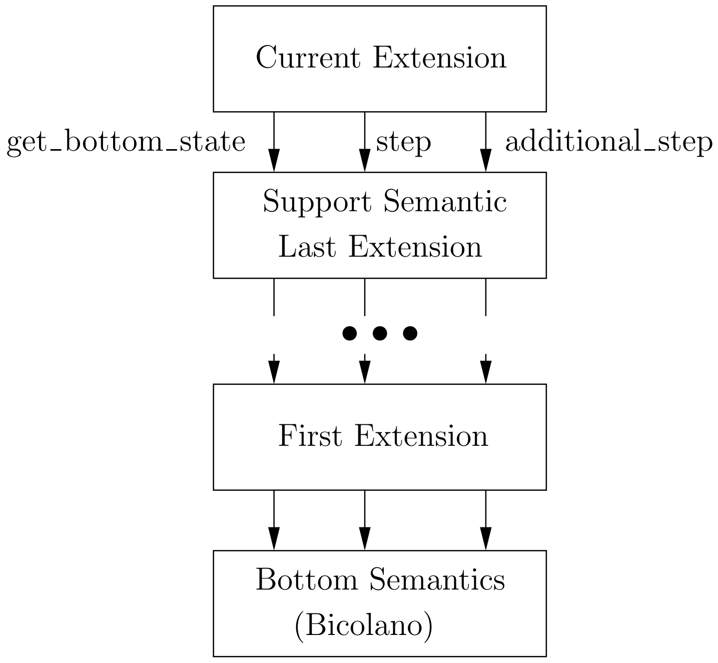

Figure 2.1: The schema of the vertical extension framework

In order to check if the new `step` relation holds we must check if the original relation holds (`_step`) and compute the value of the bookkeeping function (`bookkeepSmall`). Then the original states (`s` and `t`) are combined with the `ACT` effects (`S` and `T`, respectively) to form the values of the new state type `EState.t`.

In case of the big step semantics, the reformulation of the semantics is similar. The original semantics steps definitions are renamed by adding the underscore _ as a prefix. For instance the old `NormalStep` is renamed now to `_NormalStep`. The old names are used for new step definitions, which take into account the bookkeeping in the `ACT` component. Here is an example of the redefined step:

```
Inductive ExceptionStep(p:Program):
    Method -> IntraNormalState -> IntraExceptionState -> Prop :=
| EES_ : forall m s t S T,
       _ExceptionStep p m s t ->
       T = bookkeepException p m s t S ->
       ExceptionStep p m (s,S) (t,T).
```

This new `ExceptionStep` first checks if the original exceptional step relation holds (`_ExceptionStep`) then it generates the result of the bookkeeping (using `bookkeepException`) to establish at last the required `ExceptionStep` relation.

2.4 The vertical extension framework

This framework enables to cumulatively stack new behaviour on top of the previous semantics (Figure 2.1). The bottom of the extension stack (*bottom semantics*) is Bicolano semantics. Also of special interest is the *support semantics*, just below the current extension, which aggregates all the behaviours defined in previous kevels. Each semantics between the bottom semantics and support semantics, excluding the former and including the latter, is called *intermediate semantics*.

We assume that each extension may define behaviour which cannot be modelled with the use of the original Bicolano state definition. For instance the memory usage extension requires an additional field: the current size of the allocated memory (the maximum size is a parameter of the module). Thus, along with semantics extensions we define state extensions. The state on which the program operates in the bottom semantics is called the *bottom state*. The *support state* is the state that corresponds to the support semantics.

As Figure 2.1 suggests, the subsequent extensions are composed with help of three relations: `step`, `get_bottom_state`, and `additional_step`. The most important relation is the `step` relation which determines the behaviour of small semantical steps. This relation takes into account the behaviours described in all the extensions below the current one and the one defined by the current extension. It is worth mentioning that the extensions may influence the bottom state and the semantical steps in the extensions below may influence the step in the current extension (e.g. if the current extension traces memory usage, and the intermediate extensions allocate objects in the bottom state, this must be taken into account by the memory tracing extension).

We assume that the extension developers understand Bicolano semantics, but have limited understanding of the intermediate extensions. Thus we allow to directly refer to Bicolano state with the help of `get_bottom_state`. This relation describes how to strip off all the state information which is defined by extensions and obtain the component of the state which is directly operated by the bottom semantics. Certain extensions may have the bottom semantics undefined (e.g. an extension which supplements the semantics with a kind of error that is absent from the original formalisation) or have many possible bottom states (e.g. an extension which formalises garbage collection). Therefore, we decided to formalise `get_bottom_state` as relation rather than a function.

Finally, the relation `additional_step` complements `get_bottom_state` and allows to recognise which behaviour is added by the intermediate extensions below the current one provided that we know how the state is modified by a step in the bottom semantics.

2.4.1 How to use the framework

In order to use the extension framework in a particular situation, one has to provide two Coq modules that realise two module types. The first one (`STATE`) describes additional state components that are necessary to describe the additional behaviour modelled by the currently defined extension. The second one (`SS_SEM`) defines the actual small step semantics by providing the descriptions of `step` and `additional_step`.

The module type `STATE` is defined as:

```
Module Type STATE.
  Declare Module Dom: SEMANTIC_DOMAIN.
  Import Dom.
  Parameter state_t : Type.
  Definition bottom_state_t := Dom.State.t.
  Parameter get_bottom_state: state_t -> bottom_state_t -> Prop.
End STATE.
```

It declares the module `Dom` which contains the semantic domain, over which the state is built. The extended state is defined as `state_t` while the state component which is defined and manipulated with the use of the original Bicolano semantics is `get_bottom_state`. The signature also defines a shorthand for the type of the bottom state as `bottom_state_t`.

The `state_t` type in case of the memory extension is just product of the support state and an integer component which keeps track of the memory usage. In order to define `get_bottom_state`, we just project the state to the support state component and then use `get_bottom_state` in the support semantics to obtain the required relation.

The `state_t` type in case of the memory extension is the product of the support state and a component which contains the bottom state augmented with information on where and which instructions to set the ghost variables are used in the program. The component keeps track of the use of the ghost variables and the contains the state in which the instructions are executed. In order to define `get_bottom_state`, we have to again project the state to the support state component and then use `get_bottom_state` in the support semantics to obtain the required relation.

In order to define the semantical components of the extension one has to provide definition of a module with the following module type.

```
Module Type SS_SEM.
  Declare Module State: STATE.
  Parameter step: State.Dom.Prog.Program ->
                  State.state_t -> State.state_t -> Prop.
  Parameter additional_step: State.Dom.Prog.Program ->
                  State.state_t -> State.bottom_state_t ->
                  State.state_t -> Prop.
End SS_SEM.
```

`State` is a module which instantiates the aforementioned `STATE` module type for the current extension. The parameter `step` relates the subsequent steps of the semantics in such a way that `(step p st1 st2)` holds when there is a possible step in the extended semantics of program `p` from state `st1` to state `st2`. `additional_step` relates the states in the current semantics with the state in the bottom one. The relation `(additional_step p st1 bost2 st2)` holds when all the intermediate extensions (including the current one) transform the bottom part of `st1` to `bost2` in a step which transforms `st1` to `st2`. (Note that the complexity of this definition results from the fact that, by choice, `get_bottom_state` is a relation rather than a function.)

In case of the memory extension, the relation `state` uses in its definition the relation `additional_step` from the support semantics to check if the step there allocates memory. In this case, it adds the memory used to the component which keeps track the memory consumption. Additionally, it checks with the help of `get_bottom_state` if the current instructions allocate memory (`new` and `newarray`). In this case the memory is also added to the memory component. In case the memory limit is exceeded the extension throws the `OutOfMemoryError`. In case this error is thrown, the `additional_step` of the current extension should include the information that the error is encountered in the bottom semantics. Otherwise, `additional_step` just calls the `additional_step` in the support state.

The extension with ghost variables can use the `state` relation of the support semantics to perform the real computation and can use its own definitions to perform the computation in the state of the ghost variables. In this extension, the relation `additional_step` should be just a redirection to `additional_step` in the support semantics.

2.4.2 The realisation of the framework

The extensions framework is based on Coq modules. The semantics modules build a hierarchy. At the bottom, there is always the bottom semantics and state. The actual extensions must be provided as functors that take the support semantics as their parameters and return the resulting semantics of the extension. Therefore, the usual extension module header looks as follows:

```
Module SemExtension (SupportSem: SS_SEM) <: SS_SEM.
```

and the accompanying state extension module has a header of the following form:

```
Module StateExtension (SupportState: STATE) <: STATE.
```

The module `BottomSem` together with `BottomState` are the bottom extension modules. They have the mentioned above module types and provide their instantiation with the primary Bicolano semantics. The module `BottomState` realises the type `STATE` and `BottomSem` realises the type `SS_SEM`. The type `state_t` is defined here to be the state in the module type `SEMANTIC_DOMAIN` used in Bicolano. The relation `get_bottom_state` is defined here as the identity on the state from `SEMANTIC_DOMAIN`. The `state` relation is defined to be the small step relation of the original Bicolano as this is the basic semantical step. The relation `additional_step` is just the identity on the state type as this extension does not add any additional behaviour to the original semantics.

2.4.3 Examples of extensions

In this section we present the particular extensions we provided as an illustration of the usability of the framework. For each of the extensions we present its general overview and a description of the state type, the step relation and the additional step relation. Both step and additional step are expressed in Coq as inductive definitions. These definitions are formulated so that each inductive case has its own label and the actual definition for such a case is an implication. The descriptions of steps below present the labels of cases followed by the presentation of premises of the implication and its result.

Instruction counter extension

The instruction counter extension counts the number of executed JVM instructions. Each semantical step is supplemented with the Coq code that increments the counter which is the part of the extended state.

State The state consists of the following fields:

- the support state,
- the current number of instructions that have been executed so far.

Step Step in the semantics holds if and only if it holds in the support semantics. An instruction is executed only when the transition is from a normal state to a normal state.

1. `step_ic_count`
   Assumptions:
   (a) there exists an appropriate step in the support semantics,
   (b) this is a step from a normal state to a normal state.

   Result:
   (a) the support state part is taken from the support semantics,
   (b) the instructions counter is increased by one.
2. `step_ic_noncount`
   Assumptions:
   (a) there exists an appropriate step in the support semantics,
   (b) this is a step from an exceptional state or to an exceptional state.

   Result:
   (a) the support state part is taken from the support semantics,
   (b) the instructions counter is not changed.

Additional step We assume that an additional step from a normal state to a normal state is an execution of a JVM instruction, and other additional steps are not.

1. `adst_ic_count`
   Assumptions:
   (a) this is a step from a normal state to a normal state.

   Result:

   (a) the support state part is constructed using `additional_step` from the support semantics,
   (b) the instructions counter is increased by one.

2. `adst_ic_notcount`
   Assumptions:

   (a) this is a step from an exceptional state or to an exceptional state.

   Result:

   (a) the support state part is constructed using `additional_step` from the support semantics,
   (b) the instructions counter is not changed.

Memory extension

The memory extension keeps track of the memory used. It is done by supplementing the memory allocation instructions (i.e. `new` and `newarray`) semantics with the checking on the memory consumption. In case a memory limit is exceeded the error `OutOfMemoryError` is thrown. The maximum allowed size of memory is a parameter of the extension:

```
Parameter max_heap_size: nat.
```

In our approach the memory is never released. However, it is easy to add additional Coq code that formalises the garbage collecting.

The sizes of objects and arrays are computed using the `SIZE` module type (which is an additional parameter of the memory extension functor). It allows to give size counting functions depending on the underlying JVM implementation. We provided an implementation which assumes that integer values and references have 32 bits and which simply adds the sizes of object fields. This corresponds to a typical implementation of JVM on the Intel platform where no padding or alignment is done on the 32-bit values.

State The state consists of the following fields:

- the support state,
- the current size of the heap.

Step Step in the memory semantics holds in the following cases, as defined in the inductive definition branches:

1. `step_inherit_nn`, `step_inherit_en`, `step_inherit_ee`
   Assumptions:

   (a) there exists an appropriate step in the support semantics,
   (b) the step does *not* throw any exception,
   (c) the step is *not* the execution of a `New` nor `Newarray` instruction.

   Result:

   (a) the support state part is taken from the support semantics,
   (b) the memory related fields are not modified.

2. `step_inherit_ne`
   Assumptions:

   (a) there exists an appropriate step in the support semantics,
   (b) the step throws an exception.

   Result:

   (a) the support state part is taken from the support semantics,
   (b) the current heap size is increased by the size of the exception.

3. `step_new_ok`, `step_newarray_ok`
   Assumptions:

   (a) there exists an appropriate step in the support semantics,
   (b) the step is the execution of a `New` or `Newarray` instruction,
   (c) no exception is thrown in this step (on the support semantics level),
   (d) there is sufficient memory available.

   Result:

   (a) the support state part is taken from the support semantics,
   (b) the current heap size is increased by the size of the constructed object or array.

4. `step_new_OutOfMemory`, `step_newarray_OutOfMemory`
   Assumptions:

   (a) there exists an appropriate step in the support semantics,
   (b) the step is the execution of a `New` or `Newarray` instruction,
   (c) no exception is thrown in this step (on the support semantics level),
   (d) there is *not* sufficient memory available.

   Result:

   (a) the support state part is constructed using the `additional_step` relation from the support semantics, with the bottom state expressing that the `OutOfMemoryError` has been thrown.
   (b) the current heap size is increased by the size of the exception.

Additional step When an additional step holds in a upper level of the semantics hierarchy, it changes the memory state in the way specified here. There are several possible cases as defined in the inductive definition branches:

1. `adst_mem_nn`, `adst_mem_ee`, `adst_mem_en`
   Assumptions:

   (a) the step does *not* throw any exception.

   Result:

   (a) the support state part is constructed using `additional_step` from the support semantics,
   (b) the memory related fields are not modified.

2. `adst_mem_ne_ok`
   Assumptions:

   (a) the step throws an exception,

(b) there is sufficient memory (for the exception object).

Result:

(a) the support state part is constructed using `additional_step` from the support semantics,

(b) the current heap size is increased by the size of the exception.

3. `adst_mem_ne_OutOfMemory`
   Assumptions:

   (a) the step throws an exception,

   (b) there is sufficient memory (for the exception object).

   Result:

   (a) the support state part is constructed using `additional_step` from the support semantics, but in the bottom part the `OutOfMemoryError` error is thrown instead of the exception from the upper semantics level.

   (b) the current heap size is increased by the size of the exception.

Stack overflow extension

The stack overflow extension controls the size of the call stack. When the size of the call stack would exceed the maximal allowed size, then the error `StackOverflowError` is thrown. The maximum allowed size of the stack is a parameter:

```
Parameter max_stack_size: nat.
```

State The state type is equal to the support state type.

Step Step in the stack overflow semantics holds in the following cases, as defined in the inductive definition branches:

1. `step_inherit_ok`
   Assumptions:

   (a) there exists an appropriate step in the support semantics,

   (b) the stack size after the step is less that or equal to the maximum allowed size.

   Result:

   (a) the support state part is taken from the support semantics.

2. `step_inherit_n_overflow`, `step_inherit_e_overflow`
   Assumptions:

   (a) there exists an appropriate step in the support semantics,

   (b) the stack size after the step is greater that the maximum allowed size.

   Result:

   (a) the support state part is constructed using the `additional_step` relation from the support semantics, with the bottom state supplemented so that the `StackOverflowError` is thrown.

Additional step When an additional step holds in a upper level of the semantics hierarchy, it changes the stack overflow state in the way specified here. There are the following possible cases as defined in the inductive definition branches:

1. `adst_so_ok`
   Assumptions:
   (a) the stack size after the step is less that or equal to the maximum allowed size.

   Result:
   (a) the support state part is constructed using `additional_step` from the support semantics.
2. `adst_so_n_overflow`, `adst_so_e_overflow`
   Assumptions:
   (a) the stack size after the step is greater that the maximum allowed size.

   Result:
   (a) the support state part is constructed using `additional_step` from the support semantics, with the bottom state supplemented so that the error `StackOverflowError` is thrown.

2.5 The frameworks compared

Simulation of the horizontal framework The vertical framework is strictly more expressive than the horizontal one. In particular, it allows to define a memory tracking extension which is able to throw the `OutOfMemoryError` in case the memory limit is exceeded. This kind of behaviour is impossible to model with the help of the horizontal framework as the bookkeeping functions do not modify the state. However, an important property of the horizontal extension is that it does not change the semantics — every step of the extended semantics has a corresponding step in the support semantics and vice-versa. This property can be expressed in the vertical framework, but at an additional cost of defining and proving the following lemmas:

```
Lemma support_step_implies_step:
  forall p t s',
    SupportSem.step p (get_support_state t) s' ->
    exists t': state_t, step p t t' /\ (get_support_state t') = s'.

Lemma step_implies_support_step:
  forall p t t', step p t t' ->
    SupportSem.step p (get_support_state t) (get_support_state t').
```

The properties indeed hold and are proved for the vertical framework. However, the proofs require additional assumption on the intermediate states hierarchy:

```
Parameter fget_bottom_state: SupportState.state_t -> bottom_state_t.

Axiom get_bottom_state_ok: forall sust bost,
  SupportState.get_bottom_state sust bost <-> fget_bottom_state sust = bost.
```

This assumption expresses that the projection to the bottom state is deterministic. The assumption holds for the bottom (not extended) state and is preserved by many extensions, in particular by all the extensions described in Section 2.4.3.

Except from these properties, we also provided an extension that realises the horizontal extension framework by means of our vertical one. We additionally proved in Coq a theorem which expresses the original horizontal framework is equivalent to the one defined in terms of the vertical one.

The cost of the vertical framework The use of the vertical framework is more expensive. The files with the Coq terms that represent the definitions are bigger in the case of the vertical extension framework. For instance the memory extension in the vertical framework has 268 KB while in the horizontal extension framework only 160 KB. One cannot argue that the memory extension describes richer behaviour and this is the reason of the size increase as the extension that counts instructions has 254 KB in case of the vertical framework while only 135 KB in case of the horizontal one.

In this light, the vertical framework should be rather really used to supplement the original semantics with an additional behaviour which is necessary to be modelled while all the extensions which only trace the history of program runs, in particular the extension with the ghost variables should rather be realised with the use of the horizontal framework.

2.6 Conclusions

Proof-carrying code infrastructure can ensure safety of global computers. Such an infrastructure requires sound and complete semantics of the global computing platform. Bicolano is an operational semantics of the major part of the Java bytecode language. We present here two extension frameworks for the semantics and discuss their different features. Both frameworks are made in a modular fashion. The first one, so called horizontal, allows to extend states with additional information that traces a running program behaviour (e.g. memory consumption). The second one, so called vertical, additionally allows an extension to supplement the behaviour specified in the original semantics and supports extensions composition. A comparison of these frameworks is presented. In particular, we prove that the horizontal framework can be simulated by the vertical one and show an example of an extension which cannot be realised in the horizontal one, but can be realised in the vertical one. Extensions in the horizontal framework automatically preserve the proven soundness of the MOBIUS base logic. In the following chapters it will be seen that the horizontal framework is adequate to support reasoning about non-interference, resource usage and ghost variables.

Chapter 3

Formalising secure information flow in the MOBIUS Base Logic

3.1 Secure information flow

3.1.1 General

Personal and classified data are stored and processed in a large number of electronic computing devices. The processing programs must ensure that secrets are kept and that leakage of classified information is prevented. Printing secret information on the screen is the most obvious kind of leakage. But there are many more subtle ways how a program can leak information so that an observer can gain knowledge about the private data by observing public accessible values during one or more program runs. MOBIUS deliverable D1.1 [33] gives an extensive overview.

Example 1 *The following source code excerpts shall give an intuition which kind of programs are secure and which are classified as insecure due to direct (explicit) or indirect (implicit) flow of information. Assume a security lattice with two security levels* high *and* low *and variable* `h` *of* high *resp. variable* `l` *of* low *security level.*

1. *the program* `h = l;` *is secure as the assignment does not change a public variable and thus no information about the initial value of* `h` *is leaked.*
2. *the program* `l=h; l=0;` *is secure despite the obvious leakage of information in the first statement as afterwards the low level variable* `l` *is assigned unconditionally to* `0`*.*
3. *the program* `if (h>10) l=1; else l=2;` *is insecure due to an indirect information flow caused by the conditional's guard* `h`*. Depending on whether* `h` *is greater than* `10` *the public variable* `l` *is set to different values.*

3.1.2 Type-based and logic-based approaches

Type-based and logic-based approaches are the two main methods used in order to ensure that a program does not reveal classified information. Both methods define formal systems, i.e., a type system resp. a logic calculus, which are applied to programs and analyse them with respect to secure information flow.

Both approaches are sound in the sense that only programs that do not leak information are considered as secure and pass the analysis. The approaches are complementary in aspects of completeness, automation and expressiveness. The degree of completeness of a system indicates how many secure programs are rejected as insecure.

While type-based approaches usually work fully automatic and require no user interaction, their efficiency stems from their specialisation towards the property to be analysed and a high degree of abstraction. This specialisation and abstraction comes at the price that many secure programs are rejected as insecure.

Example 2 *Consider the following program fragment which contains only high program locations* `h1` *and* `h2`.

```
if (h1!=0)
   h2 = h2/h1;
```

The assignment in line 2 is, in principle, secure as no value is assigned to a location with a low security level. The potentially problematic point is the raise of an `ArithmeticException` *caused by a division by zero. This situation, however, is excluded by the preceding* `if`.

As current type-based systems are not *value sensitive* they reject the program in Example 2 as insecure. In contrast, logic-based approaches are based on a program logic modelling the program language semantics with no (or less) abstraction and are able to determine that the program in Example 2 is secure. This higher degree of completeness comes with a significant loss in the degree of automation. How to improve the logic-based approaches by combining them, in particular, how to exploit methods from type-based approaches is examined and done in Task 3.5 and described in MOBIUS deliverable D3.2 [35]. Here we describe how to formalise a basic notion of secure information flow, namely non-interference, in terms of the MOBIUS base logic. This formalisation will then be further used in Task 3.5 by CTH. A formalisation of security policies in the MOBIUS base logic allows also the use of uniform certificates with less code for the certificate checkers.

3.2 Formalising secure information flow

In this section we explain how to formalise non-interference in the MOBIUS base logic. In Section 3.2.1 the security types are defined in terms of location (variable, field, etc.) dependencies. Section 3.2.2 provides the necessary notions and definitions required to understand the non-interference formalisation. The formalisation itself is given in Section 3.2.3 before wrapping up in an outlook how to make use of the horizontal extension framework presented in Chapter 2 to enable efficient reasoning about non-interference properties.

3.2.1 Specification

In order to specify secure information flow one has to assign security levels to the different locations. For the used formalisation it will not be necessary to extend the MOBIUS base logic by new security types. Instead we use results presented the first time in [24] which shows that any security lattice can be simulated when using the universal lattice, which is the power set of program variables (locations) as security lattice.

The security type of a program variable is then identical to a set of program variables. The order is induced by the subset relation on the set of program variables. Intuitively the security type of a program variable can be interpreted as the set of locations on which the value of this variable is allowed to depend. Let Loc denote the set of all program variables. In order to model the elementary security lattice containing only the security types low and high one identifies the security type high with the set of all program variables and uses the set Low containing all variables of low security level to represent the low security level type. Thus the value of private variables may depend on the value of any location, whereas the value of a public variable may only depend on other low level variables.

A program assigning a high level variable to a low level variable would make the value of the low level variable dependent on the value of a high level variable which is not element of the set Low. Thus this assignment would not be according to the chosen security lattice.

3.2.2 The MOBIUS base logic

In order to formalise non-interference in the MOBIUS base logic we recall some definitions. The definitions are aligned to the description of the base logic in MOBIUS deliverable D3.1 [34]. The MOBIUS base logic is a VDM style logic and allows to reason about Java bytecode programs.

The method specification table M provides for any method m a specification entry $\mathsf{M}(m)$ of the form $((Pre, Post, Inv), \mathsf{G}, \mathsf{Q})$ where

- $(Pre, Post, \phi)$ denotes the (partial) method specification with
  - precondition Pre, which can access the initial and current state;
  - postcondition $Post$, which can access the initial, current and final state;
  - invariant ϕ to be preserved during method execution.
- G representing the current proof context and the annotation label Q relating labels to (local) assertions.

In order to provide meaningful specifications one has to know more details about the (kind of) states accessible in the pre- and postconditions as well as in the method invariants:

Initial states are used to represent program execution states before the first instruction of a method has been executed. These states provide access to local registers and the heap.

Local states are used to represent the intermediate states while a method is executed. In addition to the initial states they provide (access to) operand stacks.

Terminal states represent states after a method has been executed and provide a return value.

Verification of a method body itself is a judgment of the following form:

$$\mathsf{G}, \mathsf{Q} \vdash \{A(s_0, s)\}pc\{B(s_0, s, t)\}(I(s_1, s_2))$$

where

- $pc = (m, l)$ is the program counter referring to the instruction at position l of method m to be executed next.
- A is a precondition, which can access the initial state of the method and the current local state s.
- B is a postcondition of the method, which can access the current local state s and the final state t of the current method invocation (if the method terminates).
- I denotes an invariant which has to hold throughout the method body execution and can access the state s_1 at the current pc position and another intermediate state.

3.2.3 Translation into the base logic

Formalising non-interference in a program logic has been described in [18, 7, 19]. We will use the approach of Darvas et al. [18, 19], but instead of a dynamic logic for Java Card source code, the MOBIUS base logic for Java bytecode is used. As stated in [24] non-interference can be expressed as a relation that keeps track of which locations the value of a variable, field, register, etc., depends on after executing a sequence of program statements.

For the formalisation of non-interference in the MOBIUS base logic we fix a program `TP` as well as two sets of locations FS_{low} and FS_{high}. The non-interference property to be formalised is:

> After the execution of method m of `TP` no value stored at a location contained in FS_{low} depends on the value of a location addressed in FS_{high}.

In order to streamline the presentation we assume in the following that both FS_{high} and FS_{low} each contain exactly one unique field signature referred to as `TP.fsHigh` and `TP.fsLow`. The desired non-interference property for verification of a method body can now be expressed as follows in terms of the MOBIUS base logic:

$$\forall lloc\ \forall lv\ \exists r\ \forall hloc\ \forall hv\ \forall s_0\ \forall s\ \forall t.$$
$$\mathsf{G}, \mathsf{Q} \vdash \{Pre(s_0, s)\}pc\{NonInterferencePO(s_0, s, lloc, lv, r, hloc, hv, t)\}(true)$$

where

- $Pre(s_0, s)$ is the precondition of the method required to hold before invocation, in the simplest case, *true*. The precondition has to be provided by the specifier and is independent of the non-interference proof obligation. It may, for example, be obtained by the translation of a Bytecode Modelling Language (BML) specification into a formula of the base logic.

- $NonInterferencePO(s_0, s, \ldots, t)$ is a predicate defined as:

$$\lambda\,\overbrace{(localVarStack_0, heap_0)}^{s_0}\,\overbrace{(op, localVarStack, heap)}^{s}\,lloc\ lv\ r\ hloc\ hv\,\overbrace{(heap_f, returnVal)}^{t}\,.$$
$$get\ heap_0\ (lloc\ \texttt{TP.fsLow})\ =\ lv\ \wedge$$
$$get\ heap_0\ (hloc\ \texttt{TP.fsHigh})\ =\ hv\ \rightarrow\ get\ heap_f\ (lloc\ \texttt{TP.fsLow})\ =\ r$$

The actual formalisation in Coq code is shown in Fig. 3.1. In addition to technicalities to decompose the states for heap access, some additional work is necessary to specify the correct addressing mode. In the following we explain the non-interference predicate and its Coq formalisation in more detail.

```
Definition NonInterferencePO (s0: InitState) (s: LocalState)
              lloc (lv: value) (r: value) hloc (hv: value) (t: ReturnState) : Prop :=
 let (mem0, act0) := s0 in
  let (localVarStack0, heap0) := mem0 in
   let (memRS, actRS) := t in
    let (heapf,returnVal) := memRS in
      Heap.get heap0 (Heap.DynamicField lloc TP.fsLow) = lv
      /\
      Heap.get heap0 (Heap.DynamicField hloc TP.fsHigh) = hv
      ->
      Heap.get heapf (Heap.DynamicField lloc TP.fsLow) = r.
```

Figure 3.1: Excerpt of the Coq formalisation of the non-interference property.

In order to access the heap in the initial and final state, the states handed over to the (local) postcondition need to be decomposed. Therefore, pattern matching is performed on the parenthesised expressions matching against the initial and final state of method m. Thereby, the variables $localVarStack_0$, $localVarStack$ are matched against the collection of local registers available to the method under consideration, while the variables $heap_0, heap, heap_f$ match against the heap of the corresponding state. Further the variable $returnVal$ matches against the return value of the method and the variable op against the actual operand stack.

The crucial part of the formalisation of the non-interference property remains the quantifier shift between the universal and existential quantifiers in the formalisation above. The quantifier shift is the key to express dependence among variables.

As the quantification over the variable values occurs before the quantification over the states, it is ensured that the result value r does not indirectly depend on any high security location contained in s_0, but only on those public locations explicitly enumerated to the left of the existential quantifier. More precisely, it is stated that

> for all values lv a field `TP.fsLow` with a low security level of an object $lloc$ may have in the initial state, there exists one value r independent of the initial value stored in the high security level heap location `TP.fsHigh` of any object $hloc$ such that in the final state the value of the low level field `TP.fsLow` of the specified object $lloc$.

There is an interesting difference between the non-interference formalisation in logics with and without an explicit state representation like the dynamic logic used in KeY [9]. The difference is that in the latter case the quantification over initial states occurs implicitly in front of the quantifier shift. This is sound as a logic with shallow state embedding does not allow to quantify over locations.

The premise of the implication inside the non-interference formula accesses the initial value of the low level, respectively, the high level locations and formalises that the non-interference property must hold for any possible initial value of the involved locations.

The consequence of the implication formalises that the value of the low level heap location in the final state must be equal to the value of the existentially quantified variable r. Two things are important at this point:

- it is necessary to access the *final* state after method invocation to obtain the final value of the low security heap location;
- the previously existentially quantified variable r is independent of the values of the high security level variables.

Because of the independence of variable r from the high level variables, the formula states that there is *no* information flow from a private variable to a public variable and thus if the proof can be closed, the program is proven to have secure information flow.

Non-interference for intermediate states The given formalisation uses the postcondition of a method specification to ensure non-interference and, therefore, considers programs such as `l=h;l=0` as secure. Usually, this is the desired interpretation, but if a more strict security policy is in place then a similar judgement formula can be used as the method invariant and then ensures the non-interference property even for intermediate states. The invariant has no access to the initial and final state, therefore, one replaces these states in the above formalisation by the states directly before and after the execution of the current instruction. As these states may coincide with the initial and final state at the beginning or end of a method invocation one has to perform some renaming to allow proper template matching.

Other kinds of locations As previously mentioned the presented formula makes only use of two dynamic fields. The extension to different kinds of location (field, array component, registers, etc.) is easy. The syntactic construction of the non-interference formula starts the quantified subformula with (a list of) location(s) considered to be of a low security level. Followed by an existential quantification(s) stating the existence of value(s) r which is (are) independent of the value(s) hv of the high security level variable(s) universally quantified over afterwards. The definition of the universally quantified lists of locations that preceed and succeed the existentially quantified value r are directly obtained from the dependence sets used to represent the security types as explained in Sect. 3.2.1. Supporting other kinds of locations is straight forward and involves mostly to alternate the heap addressing mode or to access the local variable stack instead. For example, in order to provide the verification of method security signatures stating a security level for the method's return value the following version of the non-interference predicate can be used:

$$\begin{aligned}
&\lambda\ (localVarStack_0, heap_0)(op, localVarStack, heap) lloc\ lv\ r\ hloc\ hv(heap_f, returnVal).\\
&\quad get\ heap_0\ (lloc\ \texttt{TP.fsLow}) \ =\ lv \wedge\\
&\qquad get\ heap_0\ (hloc\ \texttt{TP.fsHigh}) \ =\ hv\ \to\ returnVal\ =\ r
\end{aligned}$$

It accesses the return value instead of a heap location in the postcondition. If proven valid, it is guaranteed that the return value of the method depends only on the public variable `TP.fsLow`.

3.3 Abstraction based rules—an outlook

A problem with the presented formalisation of non-interference is that one has to guess the value of the existentially quantified variable r. This can be hard as the resulting expression may be arbitrarily complex. Therefore, the approach of Hähnle et al. [23] uses an abstraction-based calculus allowing efficient reasoning by incorporating rules from type-based security systems.

The main idea is to avoid computation of all details of intermediate states, but to track only information required to reason about non-interference properties. For instance, when executing an assignment such as `x=E;` the relevant information for information flow is on which other locations variable `x` depends on rather than its concrete value.

This is an interesting use case for the modular extension framework explained in Chapter 2 of this deliverable. The idea is to use the notion of an extended state in order to record the dependencies of a location at each execution time.

In a first step one has to decide which one of the extension frameworks presented in Chapter 2 shall be used.

As the immediate goal is to collect variable dependencies along a program trace, the horizontal framework using big step semantics is the natural choice. Big step semantics, because we want to use the extension in the MOBIUS base logic. An advantage of the horizontal framework is that we do not have to adapt the soundness proof of the base logic as the machine behaviour itself is not touched. In future work it might nevertheless be necessary to use the vertical extension framework in order to provide appropriate abstraction for (blocks) of program instructions in order to simplify reasoning even further.

In the following paragraphs the presentation of the idea refers to the horizontal framework. But the description is general enough to see easily how the concepts can be transferred to the vertical framework.

As explained in Section 2.3 adding additional effects using the big step semantics of the horizontal framework requires to extend the Coq module `BS_EFFECT`. To model non-interference, this extension has to provide a table from locations to sets of locations mapping a single location to the set of locations its value depends on.

Initially, the table is assumed to map each location to the set of locations containing only itself as an element. When defining the effect of actions on such a table one ensures that any assignment of a value stored at a location r to a location l also updates the corresponding table entry of l to the set of locations stored in the table for location r.

Collecting the dependencies as additional effects allows us to abstract from the value of the existentially quantified variable shown in Sect. 3.2.3 and to consider only its dependencies, which is a much simpler task than to guess its actual value.

In order to realise the sketched tracking of location dependencies using the horizontal extension framework in its big step semantics variant, the following definitions are required:

Instantiation of parameter `ACT`. The parameter `ACT` of module `BS_EFFECT` has to be instantiated with the type representing the location dependency table `LocationDependencyMap` used to map locations to their current dependency sets. The entries in the dependency table are relative to the current state in the program trace.

Definition of the bookkeeping functions. Following the manual given in Section 2.3 one has to define additional bookkeeping functions to describe which bytecode instructions have an effect on the location dependency table and what the effect looks like. For the big step semantics there are four such bookkeeping functions including the function `bookkeepNormal` that describes the effect for a normal execution of a bytecode instruction causing no exception, method invocation or method return.

The bookkeeping functions have to describe the additional effect for all instructions storing a value in any kind of variable, field, stack or register or creating new stacks or substacks. These additional effects have to perform the following operations:

- Creation and deletion of new entries in connection with the creation of new stacks. For example, local variable stacks are affected when invoking (captured by function `bookkeepCall`) or returning from a method (captured by function `bookkeepReturn`).
- Table entry updates each time when a storage of a value occurs (captured by function `bookkeepNormal`). This implies to look up the dependencies of the location where the value to be assigned is stored and

merging[1] the corresponding entries.

With these definitions it is relatively straightforward to equip the non-interference formalisation with the capability to make use of the defined support state.

In the following we give a small example of how the suggested extension works. Assume a program contains a class `A` declaring an integer instance field `a` and a method `m`. The non-interference proof obligation for method `m` states that the initial state contains the identity map as location dependency table. Identity map means that the location dependency table maps each location l to its identity dependency set $\mathsf{D}_l := \{l\}$ containing the location itself as only entry.

Let the implementation of method `m` contain an assignment of a field to a local variable, e.g., `v = o.a`, where `v` and `o` are local variables.

The new dependency set $\mathsf{D}^{\mathrm{new}}_{\texttt{o.a}}$ for variable `v` is then computed as follows:

1. The dependency set $\mathsf{D}_{\texttt{o.a}}$ of the location `o.a` is looked up in the table.

2. If $\mathsf{D}_{\texttt{o.a}}$ contains v then $\mathsf{D}^{\mathrm{new}}_{\texttt{o.a}} := \mathsf{D}_{\texttt{o.a}} \cup \mathsf{D}_v$ otherwise $\mathsf{D}^{\mathrm{new}}_{\texttt{o.a}} := \mathsf{D}_{\texttt{o.a}}$.

3. The old dependency set $\mathsf{D}_{\texttt{o.a}}$ is replaced by $\mathsf{D}^{\mathrm{new}}_{\texttt{o.a}}$.

To provide even more details, assume the above source code line is translated into the following lines of bytecode:

```
1:      aload_1
2:      getfield   #2; //Field a:I
3:      istore_2
```

Line 1 loads the value of the local variable `o` onto the operand stack. Consequently, the dependency entry for the addressed operand stack element has to be modified (the maximal depth of the operand stack is determined at compile time). The corresponding dependency set of the table entry has to be replaced by the dependency set belonging to the table entry of local variable `o`.

The `getField` instruction in Line 2 reads the value of a field of an object and puts the read value on top of the operand stack. The reference to the object must be on top of the operand stack, hence `getField` performs first a pop operation on the operand stack accessing and removing its top element (which has to be of reference type) and performs then a heap lookup to determine the value of the specified field `#2` (here:`a`). The value is then pushed onto the operand stack.

In order to support the `getField` instruction the following additional updates on the location dependency table have to be performed each time a `getField` instruction is executed:

1. The dependencies of the operand stacks top element are saved temporarily.

2. The corresponding table entry is cleared.

3. When the value of the given field is pushed on top of the operand stack, the table entry related to the top operand stack element is assigned the merge of the dependency set belonging to specified field and the dependency set stored temporarily in step 1.

Line 3 puts the value of `o.a` onto the the local variable stack at position 2 (i.e., the position of local variable `v`). Once again one has to perform an update of the table entry belonging to local variable `v`.

Please note, that in case that a constant is assigned to a location l the corresponding dependency entry in the table is replaced by the empty set. The effect of a bytecode operation storing a constant on the operand stack on the table entries of the operand stack has to be formalised in the obvious way.

[1]Merging is here more complex than simply taking the union of both sets.

3.4 Conclusion

In this chapter we provided a formalisation of secure information flow in the MOBIUS base logic and outlined how to add abstraction-based reasoning [23] into the base logic. The results will be used and extended as part of Task 3.5 in order to specify and reason about elaborate secure information flow policies. Elaborate means to permit fine-grained specification of what information may released at a certain position in the source code under which conditions following the classification given in [41].

Chapter 4

Resource-extended **MOBIUS** Base Logic

In this chapter we give a brief overview of an extension of the MOBIUS base logic [34] for dealing with resources in a generic way. The need to develop such a logic was identified in the initial MOBIUS work plan [**?**], and the following properties were identified in Section 6.2 of [34] as being beyond the capabilities of the MOBIUS base logic:

Resource consumption Specific resources that we would like to reason about include instruction counters, heap allocation, and frame stack height. A well-known technique for modeling these resources is *code instrumentation*, i.e. the introduction of (real or ghost) variables and instructions manipulating these. However, code instrumentation appears less well suited in a PCC environment, as it does not provide an end-to-end guarantee that can be understood without reference to the program at hand. In particular, the overall satisfaction of a resource property using code instrumentation requires an analysis of the annotated program, i.e. a proof that the instrumentation variables are introduced and manipulated correctly. Furthermore, the interaction between auxiliary variables of different domains, and between auxiliary variables and proper program variables is difficult to reason about.

Execution traces Here, the goal is to reason about properties concerning a full terminating or non-terminating execution of a program, for example by imposing that an execution satisfies a formula expressed in temporal logics or a policy given in terms of a security automaton. Such specifications may concern the entire execution history, i,e. be defined over a sequence of (intermediate) Bicolano states, and are thus not expressible in the MOBIUS base logic.

Ghost variables are heavily used in JML, both for resource-accounting purposes as well as functional specifications, but are not directly expressible in the base logic.

Our work towards satisfying these requirements consists of two items of work. The first item consists of an extension of the MOBIUS base logic by a generic resource-accounting mechanism that may be instantiated to the above tasks and is described in the present chapter. Second, we have performed an analysis of the usage made of ghost variables in JML, and have developed interpretations of ghost variables either in the original MOBIUS base logic or the generic logic extension described in this section. This second item of work is reported in the following chapter.

4.1 Semantic modeling of generic resources

In order to avoid the pitfalls of code instrumentation discussed above, a semantic modeling of resource consumption was chosen. The logic is defined over an extended operational semantics, the judgements of which are formulated over the same components as the standard Bicolano operational semantics, plus a further resource-accounting component. The additional component is of the a priori unspecified type ACT, and occurs as a further component in initial, final, and intermediate states, as described in Section 2.3.

Structurally, the extended program logic follows the same design principles as the MOBIUS base logic. The global specification structure of a program P consists of a method specification table M, that contains for each method identifier m in P

- a method specification $\mathsf{S} = (R, T, \Phi)$, comprising a precondition R, a postcondition T, and a method invariant Φ,
- a local specification table G, i.e. a context of local proof assumptions, and
- a local annotation table Q that collects the (optional) assertions associated with labels in m.

An entry $\mathsf{M}(m) = ((R, T, \Phi), \mathsf{G}, \mathsf{Q})$ is to be understood as follows. The tuple (R, T) represents a partial-correctness specification, i.e. the postcondition $T(s_0, t)$ is expected to hold whenever an execution of m with initial state s_0 that satisfies $R(s_0)$ terminates, where t is the final state. The tuple (R, Φ) represents a method invariant, i.e. the assertion $\Phi(s_0, s)$ is expected to hold for any state s that arises during the (terminating or non-terminating) execution of m with initial state s_0 satisfying $R(s_0)$. The annotation table Q is a finite partial map from labels occurring in m to assertions $Q(s_0, s)$. If label l is annotated by Q, then $Q(s_0, s)$ will be expected to hold for any state s encountered at program point (m, l) during a terminating or non-terminating execution of m with initial state s_0 satisfying $R(s_0)$. Finally, the proof context G collects proof assumptions that may be used during the verification of the method. It consists of a finite partial map from labels in m to the components (A, B, I) of local proof judgements $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$.

The verification of bytecode phrases uses local judgements of the form $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$. Here,

1. A is a (local) precondition, i.e. a predicate $A(s_0, s)$ that relates the state s at program point pc (i.e. the state prior to executing the instruction at that program point) to the initial state s_0 of the current method invocation,
2. B is a (local) postcondition, i.e. a predicate $B(s_0, s, t)$ that relates the state s at pc to the initial state s_0 and the final state t of the current method invocation, provided the execution of the current method invocation terminates,
3. I is a (local) invariant, i.e. a predicate $I(s, s')$ that relates the state s at pc to any future state encountered during the continued execution of the current method, including those arising in sub-frames.
4. G is the proof context which may be used to store recursive proof assumptions, as needed e.g. for the verification of loops.

Additionally, if $pc = (m, l)$ and $\mathsf{Q}(l) = Q$, then the judgement $\mathsf{G}, \mathsf{Q} \vdash \{A\}\, pc\, \{B\}\, (I)$ implicitly also mandates that $Q(s_0, s)$ holds for all states s encountered at l, where s_0 is as before.

The verification task for full programs consists of showing that M is justified. For each entry $\mathsf{M}(m) = (\mathsf{S}, \mathsf{G}, \mathsf{Q})$, we need to show that:

1. The body b_m of m satisfies the method specification. This amounts to deriving the judgement $\mathsf{G}, \mathsf{Q} \vdash \{A_0\}\, m, init_m\, \{B_0\}\, (I_0)$ where the assertion A_0, B_0, and I_0 are obtained by converting the method specification S into the format suitable for the local proof judgement.
2. All entries in the proof context G are justified.
3. The specification table M satisfies the behavioural subtyping condition, i.e. the specification of an overriding method implies the specification of the overridden method.

Together, these conditions form the verified-program property, which we denote by $\mathsf{M} \vdash P$.

In contrast to the MOBIUS base logic, however, formulae of the extended logic refer to extended states, i.e. states that contain a component holding a value of the generic resource type ACT. As a consequence, the

```
Inductive SP_pre (P:Program) (M:Method) (l l':PC) (A:Assertion)
                 (s0:InitState) (t:LocalState) : Prop :=
   | SP_pre_def: forall s S T tt,
        A s0 (s,S) -> NormalStep P M ((l,s),S) ((l',tt),T) ->
        t = (tt,T) -> SP_pre P M l l' A s0 t

Definition SP_post (P:Program) (M:Method) (l l':PC) (B:LocalPost)
                   (s0:InitState) (r:LocalState) (t:ReturnState) : Prop :=
     forall s ss S rr R,
        s = (ss,S) -> r = (rr,R) ->
        NormalStep P M ((l,ss),S) ((l',rr),R) -> B s0 s t.

Definition SP_inv (P:Program) (M:Method) (l l' :PC) (I:StrongInv)
                 (r:LocalState) (t:LocalState) : Prop:=
     forall s ss S rr R,
        s = (ss,S) -> r = (rr,R) ->
        NormalStep P M ((l,ss),S) ((l',rr),R) -> I s t.
```

Figure 4.1: Definition of non-exceptional operators for basic instructions

proof rules are modified to take the consumption of resources into account. In particular, modified operators relating formulae of adjacent instructions are employed.

Treating the rule basic instructions as a representative example, Figure 4.1 presents the definition of the modified operators for non-exceptional behaviour, where S, T, ... range over values of type ACT. The code is directly lifted from our implementation in Coq.

The operators for exceptional behaviour of basic instructions are shown in Figure 4.2. The operational judgement `ExceptionStep` models the raising of the exception, which results in a heap `h2` with an exception object at location `loc2`, and a value `T` of the resource-accounting type ACT that arises from `S` in the generic way described in Section 2.3. Next, the operators ensure that the dynamic class of the exception object is that stipulated by the argument `e`. Finally, they model the fact that the handler is invoked on a singleton operand stack containing only the reference to the exception object.

The rules then employ the operators in a similar way as the base logic. Figure 4.3 presents the rule for basic instructions. The first two side conditions ensuring that the local annotation and the strong invariant are satisfied. Next, a hypothesis concerning normal behaviour requires the successor instruction to satisfy the judgement arising from the main judgement by applying the operators for normal behaviour.

The remaining two conditions concern exceptional behaviour. The first one applies in the case where the instruction in focus, `l`, raises an exception that is caught by the current method, `M`. In this case, any instruction label `ll` associated with `l` and the class `e` of the exception that has been raised is required to satisfy a judgement that is related to the main judgement by the operators for exceptional behaviour. As discussed above (Figure 4.2), these operators take the effect of raising an handling the exception into account in the a way that is compatible with the operational semantics. By including the type of the raised exception, `e`, in the list of arguments of the operators, it is ensured that the correct handler is chosen.

The final condition applies in the case where the instruction in focus raises an exception that is not handled locally, i.e. the current method dose not associate a handler with the class of the raised exception. At runtime, the execution of the current method would be terminated in such a case. Consequently, the condition for this case requires the (partial correctness) specification `B` to be satisfied.

Similar to this rule, modified rules are given for non-basic instructions: method invocations, switch, conditional and unconditional jumps, return instructions.

The soundness proof of the extended logic follows the same pattern as the soundness proof from Deliverable D3.1 [34], and ensures that all method specifications of verified programs are valid. Since well-typed bookeeping functions cannot change any of the Bicolano state that the step relations examine, the extended soundness proof holds for all well-typed horizontal extensions.

```
Inductive SP_preEC (P:Program) (M:Method) (l:PC) (e:ClassName)
                    (A:Assertion) (s0:InitState) (t:LocalState) : Prop :=
   | SP_preEC_def:  forall  (ops:OperandStack.t)(s:LocalVar.t)(h:Heap.t) S T,
                  A s0 ((h,ops,s),S) ->
                  forall (h2:Heap.t) loc2,
                  ExceptionStep P M ((l,(h,ops,s)),S) ((h2,loc2),T) ->
                  Heap.typeof h2 loc2 = Some (Heap.LocationObject e) ->
                  t = ((h2,Ref loc2::OperandStack.empty, s),T) ->
                  SP_preEC P M l e A s0 t.

Definition SP_postEC (P:Program) (M:Method) (l:PC) (e:ClassName)
                     (B:LocalPost) (s0:InitState) (r:LocalState)
                     (t:ReturnState) : Prop :=
   forall s h ops ss S R h2 loc2,
      s = ((h,ops,ss),S) ->
      ExceptionStep P M ((l,(h,ops,ss)),S)  ((h2,loc2),R) ->
      Heap.typeof h2 loc2 = Some (Heap.LocationObject e) ->
      r = ((h2,Ref loc2::OperandStack.empty, ss),R) ->
      B s0 s t.

Definition SP_invEC (P:Program) (M:Method) (l:PC) (e:ClassName)
                    (I:StrongInv) (r:LocalState) (t:LocalState) : Prop:=
   forall (h2:Heap.t) loc2  (s:LocalVar.t) ops h S R,
      r = (((h2,Ref loc2::OperandStack.empty), s),R) ->
      ExceptionStep P M ((l,((h,ops),s)),S) ((h2,loc2),R) ->
      Heap.typeof h2 loc2 = Some (Heap.LocationObject e) ->
      I ((h,ops,s),S) t .
```

Figure 4.2: Definition of exceptional operators for basic instructions

```
... | SP_INSTR : forall  (l:PC) (A:Assertion) (B:LocalPost) (I:StrongInv),

     isBasicInstr M l ->
     (* Local Annotation is satisfied *)
     (forall Q , Anno_LOOKUP l = Some Q ->
                (forall s0 s , A s0 s -> Q s0 s)) ->

     (* Strong Invariant is satisfied *)
     (forall s0 s, A s0 s -> I s s) ->

     (* Next instruction is valid, 3 cases : *)
        (* 1 - Normal execution, SP_pre, SP_post, SP_inv use
           NormalStep which does not contain the case of
           exception *)
        (forall ll, next M l = Some ll ->
             SP_Judgement true ll (SP_pre P M l ll A)
                                  (SP_post P M l ll B)
                                  (SP_inv P M l ll I)) ->

        (* 2 - Exception caught by the handler *)
        (forall ll e bm,
             METHOD.body M = Some bm ->
             lookup_handlers P (BYTECODEMETHOD.exceptionHandlers bm)
                                         l e ll ->
             SP_Judgement true ll (SP_preEC P M l e A)
                                   (SP_postEC P M l e B)
                                   (SP_invEC P M l e I)) ->

        (* - Uncaught exception (ie return) *)
        (forall s0 s S T, A s0 (s,S) ->
           forall h2 loc2,
              ExceptionStep P M ((l,s),S) ((h2,loc2),T) ->
              UnCaughtException  P M (l,h2,loc2) ->
              B s0 (s,S) ((h2,Exception loc2),T)) ->
    (*****************************************************************)
     SP_Judgement false l A B I
```

Figure 4.3: Proof rule for basic instructions (excerpt from the defintion of the predicate `SP_Judgement`)

Chapter 5

Elimination of ghost variables in program logics

Ghost variables are assignable variables that appear in program annotations but do not correspond to physical entities. They are used to facilitate specification and verification, e.g., by using a ghost variable to count the number of iterations of a loop, and also to express extra-functional behaviours. In this chapter we give a formal model of ghost variables and show how they can be eliminated from specifications and proofs in a compositional and automatic way. Thus, with the results of this chapter ghost variables can be seen as a specification pattern rather than a primitive notion.

5.1 Introduction

With the rapid development of programming systems, requirements for software quality become more complex. In reply to this, the techniques for program verification also evolve. Modern specification languages must support a variety of features in order to be expressive enough to deal with complex program properties. A typical example is JML (Java Modeling Language), a design by contract specification language tailored to Java programs. JML has proved its utility in several industrial case studies [16, 25]. Other examples are ESC/Java [21], the Larch methodology [22] and Spec# [4]. JML syntax is very close to the syntax of Java, supporting all Java expressions, and other specification constructs which do not have a counterpart in the Java language. While program logics and specification languages help in the development of correct code they have also been proposed as a vehicle for *proof-carrying code* (PCC) [37, 1, 3, 8] where clients are willing to run code supplied by untrusted and possibly malicious code producers provided the code comes equipped with a certificate in the form of a logical proof that certain security policies are respected. For PCC to be trusted, the underlying logical formalism must have a solid semantic basis so as to prevent inadvertent or malicious exploitation. On one hand, the logic must be shown sound with respect to some well-defined semantics; on the other hand, the meaning of specifications must be as clear as possible so as to minimise the risk of formally correct proofs which do not establish the intuitively intended property. This calls for a rigorous assessment of all the features employed in a specification language; in this chapter we do this for JML's *ghost variables.*

5.1.1 Ghost variables and their use

A ghost variable is an assignable variable that does not appear in the executable code but only in assertions and specifications. Accordingly, annotated code is allowed to contain ghost statements that assign to ghost variables; these ghost statements are not actually executed, but specifications and assertions involving ghost variables are understood "as if" the ghost statements were executed whenever reached.

Ghost variables in internal assertions Ghost variables can be used for an internal method annotation in order to facilitate the program verification process. For instance, in JML we can use a ghost variable to express that a program variable is not changed by a loop execution (by assigning to it prior to the loop), or to count the number of loop iterations. Such use of ghost variables usually makes them appear in intra-method assertions like loop invariants, but does not introduce them in the contract of a method (the pre- and postcondition). For illustration, consider an example which calculates the double of the variable x and stores it in the variable y:

```
//@ensures 2*\old(x) = y
y=0;
//@ghost int z;
//@set z = 0;
//@loop_invariant 2*z = y && z = \old(x) - x
while (x >= 0) {
  x = x - 1;
  y = y + 2;
  //@set z = z + 1;}
```

The desired property of this code fragment is introduced by the keyword ensures and states that y has accumulated the double of the initial value of x, i.e. \old(x). In the specification, we have used the ghost variable z. (z is declared in Java comments as is the case for any kind of JML specification.) Its value at the beginning of every iteration corresponds to the number of loop iterations done so far. Thus, before the loop, z is initialised to be 0 and at the end of the loop body, it is incremented. Note that z does not appear in the postcondition, i.e., the end-to-end specification of the program fragment. Its purely ancillary role is to facilitate the verification process by allowing the loop invariant to refer to the number of iterations even though no physical variable counts them.

Expressing extra-functional code properties. Secondly, ghost variables may be used to express extra-functional properties about program behavior. In such cases, ghost variables may become part of the method contract. For example, they may serve to model the memory consumption of a program. To illustrate this, consider a fragment of a Java class with two ghost variables - MEM which counts the number of allocated heap cells and MAX which models the maximal amount of heap space that can be allocated by the program:

```
//@ public static ghost int MEM;
//@ public static final ghost int MAX;

//@ requires MEM + size(A) <= MAX
//@ ensures  MEM - \old(MEM) <= size(A)
public void m () {
  A a = new A ()
 //@ set MEM = MEM + size (A)}
 ...
```

The precondition (introduced by the keyword requires) of method m states that the memory used so far (the value of the ghost variable MEM) plus the memory allocated in the method (size(A)) does not exceed the allowed limit (the value of the variable MAX). The postcondition states that the method allocates no more than the size of an object of type A. Finally, in the method body, the ghost variable MEM is set to its new value. We notice that the relationship between the value of the ghost variable MEM and the actual memory consumption of the method is implicit in the annotation policy, i.e., lies in the fact that MEM is incremented precisely when memory is being allocated and nowhere else and not modified in any other way.

Thus, ghost variables are particularly suitable when the code annotation is completely transparent, for example, for software auditing performed interactively over the source code, where a code producer verifies the written code respects their intentions. In such situations the good intuitions that ghost variables provide as opposed, to more functional or abstract ways of specification are fully brought to bear.

Ghost variables have also been used to indicate when class invariants are required to hold and may be relied upon [29] and as a means to enforce a particular order in which API methods should be invoked [39]. While our method also applies to those usages of ghost variables we limit ourselves to the former two in this chapter for lack of space.

5.1.2 Problems with ghost variables

Ghost variables lack a clear formal meaning. Usually, program semantics is expressed as a transition between states where states represent the values related to program variables. For the case of JML, verification tools like ESC/Java [21] and Jack [15] treat ghost variables as ordinary program variables. While this works in order to generate verification conditions and justify some proof rules, it is unsatisfactory if we treat program semantics as primary and program verification as a means to an end. To appreciate this point notice that the formal operational semantics of a language, e.g. Java Bytecode, can in principle not be proven adequate. One can compare it to other formalisations of the semantics, e.g. as a virtual machine, but ultimately it is an unprovable axiom that the formal semantics does indeed adequately reflect the physical effect of a program. For this reason, we feel that program semantics should be as simple as possible and certainly not be modelled to suit a particular verification methodology. Its primary aim should be to make the correspondence with the real world as evident as possible. Thus we feel one should give meaning to ghost variables without altering the operational semantics of the language, even by adding non-existent variables to its memory model.

There is a second problem with ghost variables that shows up when they are used to track extra-functional program properties like memory consumption above; the intended as opposed to the formal meaning of a contract is then contingent on respecting a particular code annotation policy. For a concrete example, suppose that we have a mobile code scenario in which a client system with constrained memory resources must receive a new component which is the class `C` implementing the interface `I` published by the client. Moreover, we suppose that the policy of the client system requires that the implementation of the interface `I` must not allocate memory in the heap. As in the example above, the client policy is expressed via the the ghost variable `MEM` which keeps track of the allocated memory in every program state, thus the component code and its specification can be:

```
public class C implements I {
  private count;
  //@requires  MEM = \old(MEM);
  //@ensures   MEM = \old(MEM);
  public void m() {
    count ++ ;}}
```

In this case the only method in class `C` does not allocate memory, hence respects the client requirements. However, the implementor might provide code (maliciously or not) that creates an object without incrementing the variable `MEM`:

```
public class C implements I {
  private count;
  //@requires  MEM = 0;
  //@ensures   MEM = 0;
  public void m() {
    count ++ ;
    A a = new A();}}
```

In the presence of a standard verification calculus which treats ghost variables as ordinary program variables, the verification of this last example will succeed although the code violates the client policy.

One could argue that this can be fixed by decreeing that a “certificate” of the resource property in question comprises not only the formal proof of the contract but also the code itself which should be manually or automatically inspected so as to ensure that ghost assignments are inserted next to all memory allocating instructions and only there. Note, however, that arguing that such a policy captures the intended resource property is again part of the *semantic gap* outside the realm of formal verification and must be left to human inspection and ultimately belief. Especially in situations where we assume the existence of malicious code producers who try to fake certificates we prefer minimise such non-rigorous methods. Of course, if we are interested in extra-functional code properties we must at some point formally define what the observable extra-functional effects of a program are, such as memory usage, time consumptions, consumption of other resources, etc. However this formalisation should be done openly by a trusted body of experts, and carefully argued by means of examples, test cases, etc; it should not be done over and over again for each verification tool or method.

We therefore argue that once we have a program semantics and program logic that can speak about extra-functional prperties it will no longer be necessary to make reference to ghost variables in contracts so that we need only the first usage of ghost variables, namely as an auxiliary device employed to facilitate a verification.

5.1.3 Contributions of this chapter

In this chapter we demonstrate that ghost variables can be eliminated from formal proofs in a program logic in such a way that on the one hand the same outside contracts will be proved and on the other hand the intuitive ease that ghost variables afford is retained. We do this by showing that proofs in a program logic with ghost variables can be translated automatically and compositionally into proofs of the same specifications in a logic that does not use ghost variables. In other words, ghost variables become a conservative extension of ordinary program logics.

In order to focus on salient aspects we study the problem of ghost variables using a simple, unstructured while language specified by a big-step operational semantics and reasoned about in a VDM-style program logic using I/O-relations as assertions. The proof rules of the program logic are such that whenever $C : P$ is provable then $P(S, T)$ holds where S, T are respectively initial and final states of a terminating run of program C.

Elimination of ghost variables We consider programs C_g annotated with assignments to ghost variables and introduce ad-hoc proof rules for deducing statements of the form $C_g : P_g$ where, now, P_g is a binary relation between pairs of states: (initial state, initial ghost state) and (final state, final ghost state). The proof rules are motivated by the intuitive meaning of ghost variables but are not formally validated against any kind of operational semantics of ghost variables. Instead, our first result shows that if we have a derivation of $C_g : P_g$ then we can *effectively* find a derivation of $C : P$ where C is the program C_g with all ghost instructions removed and where $P(S, T) \iff \forall S_g.\exists T_g.P_g((S, S_g), (T, T_g))$. In particular, whenever $P_g((S, S_g), (T, T_g)) \iff Q(S, T)$ for some I/O-relation Q then $P \iff Q$. This models the case where ghost variables do not appear in the outside contract, but possibly appear in internal assertions, e.g. as invariants in invocations of the proof rule for while-loops. The qualification “effective” of the claimed proof transformation means that the transformation is by induction on proofs and does not require inventing of new invariants, assertions, mathematical proofs of side condition or similar, and is thus fully automatic. Without this extra qualification a result like the one we announced could be trivially true by appealing to a completeness result for the program logic.

Extension to extra-functional properties We then extend our approach to encompass extra-functional properties. In order to model these we extend our language by *external procedures* that have no effect on the store but do cause an event to occur that is visible from the outside. Formally, we assume a set *Extern* of external functions and decree that for $f \in \mathit{Extern}$ and e an integer expression we can form the command

$f(e)$ which has the same effect as *Skip* but causes the *event* (f, n) to occur where n is the current value of expression e. Thus, an event is an element of $\mathit{Event} := \mathit{Extern} \times \mathbf{Z}$.

Now that programs can cause observable effects during their execution we can no longer semantically identify all nonterminating programs as is typically done by big-step operational semantics. Instead we define for each program C a relation $\overset{C}{\to}$ where $S \overset{C,\vec{ev}}{\to} S'$ means that when we start program C in initial state S then during its execution there is a point at which we have reached state S' and up to that point the events $\vec{ev} \in \mathit{Event}^*$ have occurred.

We then consider a program logic that in addition to VDM-style assertions (which now may also mention the trace of events occurred) also has a judgement $C : I$ with the intention that whenever $S \overset{C,\vec{ev}}{\to} S'$ then $I(S, ev)$ will hold. The rules for the definition of this judgement have premises referring to the usual assertions. In this extension of the program logic we can thus assert extra-functional properties without using ghost variables. We show that, again, ghost variables can be eliminated from proofs of specifications that do not themselves mention ghost variables.

Suppose now that we have a proof that program C satisifies the invariant "`MEM` = 0" where `MEM` is a ghost variable purportedly counting the number of memory allocations made. As argued above such a proof ought to be accompanied by a formal argument explaining that the ghost variable `MEM` really does reflect the number allocations made. In our resource-enhanced logic this could be formalised as a proof of the invariant `MEM` $= \mathit{mem}(\vec{ev})$ where $\mathit{mem}(\vec{ev})$ is the number of allocation events in execution trace *tr*. Combining the two proofs then yields a proof of the invariant $\mathit{mem}(\vec{ev}) = 0$ to which elimination of ghost variables applies.

Coq development All definitions, theorems, proofs have been carried out within the Coq theorem prover and are available for download at `www.tcs.ifi.lmu.de/~mhofmann/fsttcscoq.tgz`.

5.2 Preliminaries

5.2.1 Simple programming language

We consider a simple programming language with assignment, conditional, loop, sequence, and skip statements:

Inductive *stmt* : *Type* :=
  | *Assign* : *var* → *expr* → *stmt*
  | *If* : *expr* → *stmt* → *stmt* → *stmt*
  | *While* : *expr* → *stmt* → *stmt*
  | *Sseq* : *stmt* → *stmt* → *stmt*
  | *Skip* : *stmt*.

Here and in the rest of the chapter, we use a Coq syntax for introducing definitions. The Coq code above is an inductive type with several constructors corresponding to the different statements of the language. This definition thus corresponds to the following more common notation:

stmt :=
  | *Assign* (*var expr*)
  | *If* (*expr stmt stmt*)
  ...

We elide the syntax of arithmetic expressions. Values in our language are integers. Our formal Coq development includes recursive methods, which we omit here for simplicity.

In the Appendix we give a standard big-step operational semantics which characterises the terminating executions of program statements. It is defined as a relation between initial and final states of the execution of a statement, where states are mappings from variables to values:

$\mathit{exec\_t} : \mathit{state} \to \mathit{stmt} \to \mathit{state} \to \mathit{Prop}$

5.2.2 Logic for partial correctness for a simple language

The partial correctness logic is formulated in a VDM style [26]. This differs from the perhaps more common Hoare style rules, where assertions are predicates on the current state; in VDM, program assertions are functions of the initial and final state of a program statement:

Definition *assertion* := *state* → *state* → *Prop.*

This choice avoids the use of auxiliary variables, which is necessary in Hoare logic for relating the values of variables in different states [**?**]. The logic is encoded in Coq as an inductive predicate with one constructor for each proof rule:

Inductive *RULET*: *stmt* → *assertion* → *Prop.*

See Appendix for details.

The soundness theorem shows correctness of the logic w.r.t. the operational semantics and is formulated as follows:

Lemma *correct*: ∀ (*st*: *stmt*) (*s1 s2* : *state*),
exec_t s1 st s2 →∀ (*post* : *assertion*), *RULET st post* → *post s1 s2.*

5.3 Logic for partial correctness for a language with ghost variables

We now consider an extension of the simple language with ghost variables. To that end, we assume a set of ghost variables *gVar* disjoint from the set of program variables *var*. The language, formalised as an inductive type *Gstmt* (see Appendix) has the same constructs as the original language (*Stmt*) plus a new construct, *GAssign*, allowing one to assign to ghost variables. Ghost variables are not allowed to appear in guards of loops or case distinctions nor may they be written into ordinary variables so as not to influence the flow of control in any way.

Properties of programs with ghost variables should certainly talk about the values of ghost variables. Thus, ghost assertions *Gassertion* are mappings from the initial and final program states and also from the initial and final ghost states to a truth value:

Definition *Gassertion* := *state* → *gState* → *state* → *gState* → *Prop.*

The logic of the ghost language then takes the form of an inductive definition:

Inductive *GRULE*: *Gstmt* → *Gassertion* → *Prop*

The rules for this logic are similar to the rules for the standard simple language, but are defined for assertions with ghost variables. Consider e.g. the assignment rule, which differes from the non-ghost assignment rule only in that ghost states are threaded through and required not to change.

GAssignRule: ∀ *x e* (*post* : *Gassertion*),
(∀ (*s1 s2* : *state*) (*g1 g2*: *gState*),
g1 = *g2* → *s2* = *update s1 x* (*eval_expr s1 e*) → *post s1 g1 s2 g2*) →
GRULE (*GAssign x e*) *post*

The only substantial difference between the logic for standard simple language and its ghost extension is the rule for ghost assignment, which does not have an analogue in the standard logic:

GSetRule : ∀ *x* (*e* : *gExpr*) (*post* : *Gassertion*),
(∀ (*s1 s2* : *state*) (*g1 g2*: *gState*),
g2 = *gUpdate g1 x* (*gEval_expr s1 g1 e*) → *s1* = *s2* →
post s1 g1 s2 g2) → *GRULE* (*GSet x e*) *post.*

5.3.1 Relation between ghost and standard partial logic

In the following we use a function *transform* : *Gstmt* → *Stmt* that returns the underlying standard program by replacing all ghost assignments with skips. With each ghost assertion ψ (of type *Gassertion*) we associate a standard assertion *transform*(ψ) (of type *assertion*) by

$$\mathit{transform}(\psi) := \lambda\sigma_0, \sigma_1.\forall\sigma_0^g, \exists\sigma_1^g,\ \psi\ \sigma_0\ \sigma_0^g\ \sigma_1\ \sigma_1^g$$

Notice that if ψ does not mention ghost variables then *transform*(ψ) is equivalent to ψ itself. The formal statement about the relation of the two logical systems then says that a proof in the ghost logic (*GRULET*) that a statement *stmt* of the ghost language meets the ghost assertion ψ can be transformed into a proof in the standard logic (*RULET*) that the statement *transform*(*stmt*) meets the assertion *transform*(ψ):

Lemma *Relation between standard and ghost partial logics*:
 ∀ (*gst*: *Gstmt*) (*Gpost*: *Gassertion*),
 let st := transform gst in
 let post:= (*fun s1 s2* ⇒ ∀ (*sg1*: *gState*),∃ *sg2*:*gState*, *Gpost s1 sg1 s2 sg2*) *in*
 GRULE gst Gpost → *RULE st* (*fun s1 s2* ⇒ *post s1 s2*).

The proof of this statement is done by induction over the the ghost logical rules (*GRULET*). The curious part of this result is that the respective proof in the standard logic uses the same loop invariants with the respective quantifications (universal for the values in the initial state and existential for the values in the final state) over the ghost variables. Moreover, the established relation between the ghost and standard logic proposes an algorithm for transformation of "ghost" specifications into standard specification constructs without ghost variables. Since the proof is conducted by induction over proof rules it contains an algorithm that effectively performs the transformation on the level of proofs.

Returning to our example which is actually provable with the program logic *GRULET*, the respective program and annotation provable in the logic *RULET* are the following:

```
//@ensures y = 2*\old(x)

y=0;

//@loop_invariant \exists z, y = 2 * z && x = \old(x) - z
while (x >= 0) {
  x = x - 1;
  y = y + 2;
}
```

The new specification does not only quantify the loop invariant over the ghost variable, but the ghost variable has been completely removed from it. Of course, it would have been possible to use such existentially quantified invariant in the first place or even cleverly guess the equivalent invariant `y=2*(x-\old(x))`. Many people find this confusing and cumbersome and prefer to use ghost variables. Our result shows that this is perfectly rigorous and can be understood as a shorthand comparable, e.g., to the use of named variables as opposed to combinators.

We remark that if a specification does not contain ghost variables but its proof does then that same specfication is provable in the ordinary program logic using the above correspondence followed by an instance of the consequence rule thus establishing conservativity of ghost variables.

Lemma *conservative*: ∀ (*s*: *Gstmt*) (*post* : *assertion*),
 GRULE stmt (*fun* (*s1*:*state*)(*g1*:*gState*)(*s2*:*state*)(*g2*:*gState*) ⇒ *post s1 s2*) →
 RULE (*transform stmt*) *post*.

5.4 Ghost variables and trace properties

So far, we have seen the meaning of ghost variables w.r.t. a standard partial correctness. Such formulation basically describes the functional relation between input/output. In the following sections we show how to extend our results to reasoning about extra-functional properties such as "a program should not allocate more than X memory cells", "a program should not open nested transactions", "a program should not open more than X number of files" etc. Indeed, the practical interest of being able to reason over such extra-functional properties is evident, especially for critical applications tailored to PDAs or smart cards [10, 39] or in mobile code scenarios.

An important new feature brought about here is that one can no longer semantically identify all non-terminating programs which we address by axiomatising reachable states and adding invariants to specifications as explained in the Introduction. Formally, we specify the semantics of reachable states of the thus extended language with the following inductive predicate:

Inductive *reach*: *state* → *stmt* → *list event* → *state* → *Prop*

where $reach(\sigma_0, stmt, \vec{ev}, \sigma_1)$ means that the execution of *stmt* started in state σ_0 reaches the state σ_1 and produces the list of events $\vec{ev}$. The definition of the predicate *reach* relies on the notion of terminating executions which is defined with the following predicate:

Inductive *t_exec*:*state* → *stmt* → *list event* → *state* → *Prop*

The predicate *t_exec* is defined standardly in a big step style but this time it keeps track not only of the initial and final state but also of the list of events produced during the execution. The defining clauses for both predicates are given in the Appendix.

The properties which are managed by the trace logic will speak about the initial state of the statement execution and the trace generated so far. Thus, trace invariants have the form:

Definition *invariant* := *state* → *list event* → *Prop.*

The logic which allows to reason over trace properties is defined in Coq as the inductive type $RULER$(see Appendix). The definition of $RULER$ uses the logic for partial correctness for our language. The latter is defined via the inductive definition $RULET$ which has the following signature:

Inductive $RULET$: *stmt* → *assertion* → *Prop*

The inductive predicate $RULET$ defines a standard partial program logic only that the assertions it manipulates now are defined to depend not only on the initial and final state but also on the trace of events produced during execution:

Definition *assertion* := *state* → *list event* → *state* → *Prop.*

The soundness statement of the logic requires that if a statement has a proof in $RULER$ w.r.t. an invariant then every reachable state of the execution of the statement satisfies the invariant:

Lemma *soundReach*: ∀ *stmt* (*s1 s2* : *state*) *events*,
(*reach s1 stmt events s2*) →∀ *inv*, *RULER stmt inv* → *inv s1 events* .

The proof of the upper lemma is done by induction over $RULER$. Note that because $RULER$ uses the logic $RULET$ for partial correctness its soundness proof exploits the soundness of $RULET$.

5.4.1 Program logic for trace properties and ghost variables

The logic for trace properties tailored to a language with ghost variables is actually quite the same as the logic for trace properties for a standard language presented in the previous section. The only difference is that the ghost trace logic manipulates assertions with ghost variables and the assertion for trace properties talk about the initial and current values of ghost variables. Thus, the signature of ghost trace invariants is as follows:

Definition *Ginvariant* := *state* → *gState* → *list event* → *gState* → *Prop.*

Similarly for the case of the trace logic without ghost variables, we also need to define assertions that are manipulated by the partial ghost logic. Those now have the signature:

Definition *Gassertion* := *state* → *gState* → *list event* → *state* → *gState* → *Prop*

5.4.2 Relation between standard and ghost logics for trace properties

In the presence of event traces, there exists again an effective way for transforming ghost programs, ghost specifications and their proofs into standard programs, specifications and proofs.

Lemma *Relation between standard and ghost trace logics*:
∀ (*gstmt*:*Gstmt*) (*ginv*: *Ginvariant*),
let stmt := *transform gstmt in*
let inv := (*fun s1 event* ⇒ ∀ *g1*, ∃ *g2*, *ginv s1 g1 event g2*) *in*
RULERG gstmt ginv → *RULER stmt inv.*

5.5 Related work

There is some research work that shows the use of ghost variables. The paper [39] presents an algorithm for automatic annotation of Java Card applets against security policies concerning the transaction mechanism in the Java Card virtual machine. The algorithm models the transaction mechanism via a ghost variable whose value is changed when the application opens, commits or aborts a transaction. The authors of [6] discuss how memory consumption policies may be encoded with the help of ghost variables (in the fashion that we show in our introduction). Although these papers show the practical interest of ghost variables, to our knowledge their formal meaning has not been so far investigated.

Regarding formalisms for reasoning over trace properties (that hold throughout a program execution), B.Beckert and S. Schlager [11] describe two sound extensions of dynamic logic - one to deal with properties that must hold for every state in the execution and another one which validates properties that hold in at least one state of the program execution. This extension has been implemented in the Key verification framework for JavaCard programs and applied for reasoning over the transaction mechanism of the JavaCard programs [39]. The MOBIUS base logic [34] supports both partial correctness w.r.t. a postcondition and invariant correctness w.r.t a property that must hold throughout the execution.

5.6 Conclusion

We have given a rigorous semantics of ghost variables in terms of a VDM-style program logic without altering in any way the operational semantics of the language which, as we have argued, is a source of vulnerability for proof-carrying code architectures since it escapes formal validation. We have also argued that ghost variables can be avoided in end-to-end specifications of extra-functional properties provided the program logic is given the ability to speak about traces of observable events. Dynamic logic also offers some of the features that we propose: asserting that some extra-functional property holds throughout the execution [11] and the use of existential quantification in situations where ghost variables might appear [9]. The fact that proofs involving ghost variables (of terminating and non-terminating programs) can always and automatically be translated into proofs without ghost variables appears here for the first time and is the main technical contribution of this chapter. We found our approach to be very robust and did not experience obstacles with the inclusion of recursive methods as well as with the other uses of ghost variables [29, 39] mentioned in the introduction. A full version will contain a more detailed account. We also find that translation into a standard program logic is in general a useful method for giving meaning to the fancier features of specification languages.

Chapter 6

Conclusion

6.1 Summary

In this task we have shown how to extend the MOBIUS reasoning base (described in MOBIUS deliverable D3.1 [34]) to be able to specify and reason about information flow and resource properties of Java bytecode programs. Ghost variables, often used for expressing program trace properties, including resource properties, are also supported in bytecode by these extensions, and shown to be eliminable in certain cases.

Bicolano extensions In Chapter 2 we describe Coq code of two frameworks for extending the underlying Bicolano model of the JVM. One of these, the horizontal extension framework, allows a new "ghost" field to be added to the Bicolano state, and bookkeeping functions to maintain information up to a full program trace in this new field. An essential point is that horizontal extensions are safe, by design they cannot change the transition behaviour of Bicolano. This horizontal framework also supports ghost variables for bytecode, by representing them in the ghost field added to Bicolano state.

We also present a general iterative extension framework, the vertical extension framework. This allows to modify Bicolano state and transitions, but safety of such an extension is not automatic. It is up to the person defining a vertical extension to assure that it behaves as intended.

Information flow In Chapter 3 we show how to express non-interference, a prototypical information flow property, in the existing MOBIUS base logic. This simplest representation is seen by experience to be inconvenient in practice, so a more convenient refinement of the representation technique is also shown. This refinement uses the horizontal extension framework of Bicolano.

Base logic extension and soundness In Chapter 4 we give an extension of the MOBIUS base logic that is compatible with the horizontal extension framework of Bicolano. This extended base logic supports assertions that may refer to the ghost field added by horizontal extensions of Bicolano. This extension is carried out in the Coq code of the base logic. Since horizontal extensions of Bicolano cannot change transition behaviour, there is also a once-and-for-all Coq verified proof of soundness of the extended base logic.

Ghost variables: semantics and eliminability In Chapters 2 and 4 we have shown how to reason about "ghost variables" which can occur in method specifications and in local assertions. In this chapter we clarify the semantics of ghost variables, and show that they can be eliminated from assertions in some cases by compiling into non-ghost assertions.

6.2 Further work

Here are areas for further work, and their places in the MOBIUS work program for the next 18 months [?].

Information flow In this task we showed how to represent non-interference in the MOBIUS reasoning base. It is desired to be able to represent and reason about more flexible information flow properties, e.g. involving declassification. Work on this area is planned by CTH for MOBIUS Task 3.5.

In addition, LMU is working on an alternative for formalising secure information flow in the MOBIUS base logic. Preliminary work has appeared in [12], and will be detailed in Task MOBIUS 3.5.

Verification condition generation The MOBIUS base logic is a low-level program logic for Java bytecode; it is not expected that users will do proofs by hand in this logic. In MOBIUS Deliverable 3.1 [34] two verification condition generators were presented; one directly in Coq, proven sound w.r.t. the base logic in Coq, but producing crude verification conditions; one more sophisticated, using BoogiePL, producing smaller verification conditions. Since both Bicolano and the base logic are extended in the present task, there is need to update these verification condition generators. Work on efficient verification condition generation appears in Tasks 4.3 and 3.6.

Bibliography

[1] A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.

[2] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Principles of Programming Languages*. ACM Press, 2000.

[3] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile resource guarantees for smart devices. In Barthe et al. [5], pages 1–26.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Barthe et al. [5], pages 151–171.

[5] G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors. *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[6] G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In *Software Engineering and Formal Methods*, pages 86–95. IEEE Press, 2005.

[7] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Types in Language Design and Implementation*, pages 103–112. ACM Press, 2005.

[8] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. MOBIUS: Mobility, ubiquity, security — objectives and progress report. In *TGC 2006* [20].

[9] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* Number 4334 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[10] B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering*, volume 2621 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, April 2003.

[11] B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 626–641. Springer-Verlag, 2001.

[12] L. Beringer and M. Hofmann. Secure information flow and program logics. In *IEEE Computer Security Foundations Symposium*, pages 233–248. IEEE Press, 2007.

[13] L. Beringer and Martin Hofmann. A bytecode logic for JML and types. In N. Kobayashi, editor, *Programming Languages and Systems: Proceedings of the 4th Asian Symposium, APLAS 2006*, volume 4279 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 2006.

[14] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[15] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In D. Mandrioli K. Araki, S. Gnesi, editor, *Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.

[16] N. Cataño and M. Huisman. Chase: A static checker for JML's assignable clause. In *Verification, Model Checking and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40. Springer-Verlag, 2003.

[17] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, March 2004. `http://coq.inria.fr/doc/main.html`.

[18] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security.* IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[19] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.

[20] *Trustworthy Global Computing: Revised Selected Papers from the Second Symposium TGC 2006, Lucca, Italy, November 7–9, 2006*, number 4661 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Languages Design and Implementation*, volume 37, pages 234–245, June 2002.

[22] J. V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[23] Reiner Hähnle, Jing Pan, Philipp Rümmer, and Dennis Walter. Integration of a security type system into a program logic. In *TGC 2006* [20], pages 166–131.

[24] S. Hunt and D. Sands. On flow-sensitive security types. In *Principles of Programming Languages*, Charleston, South Carolina, USA, January 2006. ACM Press.

[25] B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, Stirling, UK, July 2004. Springer-Verlag.

[26] C. B. Jones. *Systematic software development using VDM.* Prentice Hall, 1990.

[27] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[28] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[29] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004. Available from `www.sct.inf.ethz.ch/publications/index.html`.

[30] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234. Springer-Verlag, 1996.

[31] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification. Second Edition.* Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

[32] M. Marcus and A. Pnueli. Using ghost variables to prove refinement. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 226–240. Springer-Verlag, 1996.

[33] MOBIUS Consortium. Deliverable 1.1: Resource and information flow security requirements, 2006. Available online from `http://mobius.inria.fr`.

[34] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from `http://mobius.inria.fr`.

[35] MOBIUS Consortium. Deliverable 3.2: Intermediate report on embedding type-based analyses into program logics, 2007. Available online from `http://mobius.inria.fr`.

[36] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[37] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[38] F. Nielson and H. Riis Nielson. Type and effect systems. In *Correct System Design*, number 1710 in Lecture Notes in Computer Science, pages 114–136. Springer-Verlag, 1999.

[39] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Smart Card Research and Advanced Application.* Kluwer Academic Publishing, 2004.

[40] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`. Summary appears in [34], 2006.

[41] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations Workshop*, pages 255–269. IEEE Press, 2005.

[42] Christian Skalka and Scott F. Smith. History effects and verification. In Wei-Ngan Chin, editor, *Asian Programming Languages and Systems Symposium*, volume 3302 of *Lecture Notes in Computer Science*, pages 107–128. Springer-Verlag, 2004.

[43] Sun Microsystems Inc., 4150 Network Circle, Santa Clara, California 95054. *Connected Limited Device Configuration.Specification Version 1.1. Java^{TM} 2 Platform, Micro Edition (J2ME^{TM})*, March 2003.

[44] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[45] M. Tofte and J-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

Appendix A

Formal Coq-definitions for Chapter 5

A.1 Functional Behaviours

A.1.1 Syntax of the Language

```
Inductive exec_t : state → stmt → state → Prop :=
 | ExecAssign : ∀ s x e,
     exec_t s (Assign x e) (update s x (eval_expr s e))
 | ExecIf_true : ∀ s1 s2 e stmtT stmtF,
     eval_expr s1 e ≠ 0 →
     exec_t s1 stmtT s2 →
     exec_t s1 (If e stmtT stmtF) s2
 | ExecIf_false : ∀ s1 s2 e stmtT stmtF,
     eval_expr s1 e = 0 →
     exec_t s1 stmtF s2 →
     exec_t s1 (If e stmtT stmtF) s2
 | ExecWhile_true : ∀ s1 s2 s3 e stmt,
     eval_expr s1 e ≠ 0 →
     exec_t s1 stmt s2 → exec_t s2 (While e stmt) s3 →
     exec_t s1 (While e stmt) s3
 | ExecWhile_false : ∀ s1 e stmt,
     eval_expr s1 e = 0 →
     exec_t s1 (While e stmt) s1
 | ExecSseq : ∀ s1 s2 s3 i stmt1 stmt2,
      exec_t s1 stmt1 s2 →
      exec_t s2 stmt2 s3 →
      exec_t s1 (Sseq stmt1 stmt2) s3
 | ExecSkip : ∀ s, exec_t s Skip s.
```

A.1.2 Logic for partial correctness

```
Inductive RULET: stmt → assertion → Prop :=
 | AssignRule:∀ x e (post: assertion),
   (∀ (s1 s2: state), s2 = update s1 x (eval_expr s1 e) → post s1 s2) →
   RULET (Assign x e) post
 | IfRule:∀ e (stmtT stmtF: stmt ) (post1 post2 post : assertion),
   (∀ ( s1 s2: state),
     (eval_expr s1 e ≠ 0 → post1 s1 s2) →
     (eval_expr s1 e = 0 → post2 s1 s2) →
```

```
        post s1 s2) →
    RULET stmtT post1 →
    RULET stmtF post2 →
    RULET (If e stmtT stmtF) post
 | WhileRule:∀ (st: stmt)(post b post1 : assertion) e,
    (∀ s1 s2, eval_expr s2 e = 0 → post1 s1 s2 → post s1 s2) →
    (∀ s p t, eval_expr s e ≠ 0 → b s p → post1 p t → post1 s t) →
    (∀ s, eval_expr s e = 0 → post1 s s) →
    RULET st b→
    RULET (While e st) post
 | SeqRule:∀ (stmt1 stmt2: stmt ) (post1 post2 post: assertion),
    (∀ s1 s2,(∃ p, post1 s1 p ∧ post2 p s2) → post s1 s2) →
    RULET stmt1 post1 →
    RULET stmt2 post2 →
    RULET (Sseq stmt1 stmt2) post
 | SkipRule:∀ (post: assertion),
    (∀ (s1 s2: state), s1 = s2 → post s1 s2) →
    RULET Skip post.
```

A.1.3 Syntax of language with ghost variables

```
Inductive Gstmt : Type :=
 | GAssign: var → expr → Gstmt
 | GIf : expr → Gstmt → Gstmt → Gstmt
 | GWhile : expr → Gstmt → Gstmt
 | GSseq : Gstmt → Gstmt → Gstmt
 | GSkip : Gstmt
 | GSet : gVar → gExpr → Gstmt.
```

A.2 Extra-functional behaviours with traces

A.2.1 Semantics of terminating executions in the presence of traces

```
Inductive t_exec : state → stmt → list event → state → Prop :=
 | ExecAffect : ∀ s x e,
      t_exec P B s (Affect x e) nil (update s x (eval_expr s e))
 | ExecIf_true : ∀ s1 s2 e stmtT stmtF eventsT,
      eval_expr s1 e ≠ 0 →
      t_exec P B s1 stmtT eventsT s2 →
      t_exec P B s1 (If e stmtT stmtF) eventsT s2
 | ExecIf_false : ∀ s1 s2 e stmtT stmtF eventsF,
      eval_expr s1 e = 0 →
      t_exec P B s1 stmtF eventsF s2 →
      t_exec P B s1 (If e stmtT stmtF) eventsF s2
 | ExecWhile_true : ∀ s1 s2 s3 e stmt eventsI eventsC ,
      eval_expr s1 e ≠ 0 →
      t_exec P B s1 stmt eventsI s2 →
      t_exec P B s2 (While e stmt) eventsC s3 →
      t_exec P B s1 (While e stmt) (app eventsI eventsC) s3
 | ExecWhile_false : ∀ s1 e stmt ,
      eval_expr s1 e = 0 →
      t_exec P B s1 (While e stmt) nil s1
 | ExecSseq : ∀ s1 s2 s3 stmt1 stmt2 events1 events2,
      t_exec P B s1 stmt1 events1 s2 →
      t_exec P B s2 stmt2 events2 s3 →
      t_exec P B s1 (Sseq stmt1 stmt2) (app events1 events2) s3
 | ExecSkip : ∀ s, t_exec P B s Skip nil s
 | ExecSignal : ∀ s event, t_exec P B s ( Signal event ) ( event :: nil ) s .
```

A.2.2 Semantics of reachable states in the presence of traces

```
Inductive reach: state → stmt → list event → state → Prop :=
| ReachAssign: ∀ s x e,
   reach s (Assign x e) nil (update s x (eval_expr s e))
| ReachIf_true: ∀ s1 s2 e stmtT stmtF eventsT ,
   eval_expr s1 e ≠ 0 →
   reach s1 stmtT eventsT s2 →
   reach s1 (If e stmtT stmtF) eventsT s2
| ReachIf_false: ∀ s1 s2 e stmtT stmtF eventsF,
   eval_expr s1 e = 0 →
   reach s1 stmtF eventsF s2 →
   reach s1 (If e stmtT stmtF) eventsF s2
| ReachWhile_false: ∀ s1 e stmt,
   eval_expr s1 e = 0 →
   reach s1 (While e stmt) nil s1
| ReachWhile_true1 : ∀ s1 s2 e stmt eventsB,
   eval_expr s1 e ≠ 0 →
   reach s1 stmt eventsB s2 →
   reach s1 (While e stmt) eventsB s2
| ReachWhile_true2: ∀ s1 s2 s3 e stmt eventsB eventsW,
   eval_expr s1 e ≠ 0 →
   t_exec s1 stmt eventsB s2 →
   reach s2 (While e stmt) eventsW s3 →
   reach s1 (While e stmt)(eventsB::eventsW) s3
| ReachSseq1: ∀ s1 s2 stmt1 stmt2 events1,
   reach s1 stmt1 events1 s2 →
   reach s1 (Sseq stmt1 stmt2) events1 s2
| ReachSseq2: ∀ s1 s2 s3 stmt1 stmt2 events1 events2,
   t_exec s1 stmt1 events1 s2 →
   reach s2 stmt2 events2 s3 →
   reach s1 (Sseq stmt1 stmt2) (events1::events2) s3
| ReachSkip: ∀ s, reach s Skip nil s
| ReachRefl : ∀ s stmt, reach P B s stmt nil s
| ReachSignal: ∀ s event ,
 reach s (Signal event) (event::nil) s.
```

A.2.3 Logic for partial correctness in the presence of traces for the extended language

```
Inductive RULET : stmt → assertion → Prop :=
| AffectRule : ∀ x e (post : assertion) ,
  (∀ (s1 s2: state), s2 = update s1 x (eval_expr s1 e) → post s1 nil s2) →
  RULET (Affect x e) post
| IfRule : ∀ e (stmtT stmtF: stmt)(post1 post2 post : assertion) ,
  (∀ ( s1 s2: state) event,
     ((eval_expr s1 e ≠ 0)) → post1 s1 event s2) ∧
     (eval_expr s1 e = 0 → post2 s1 event s2) → post s1 event s2) →
  RULET stmtT post1 →
  RULET stmtF post2 →
  RULET (If e stmtT stmtF) post
| WhileRule : ∀ (st : stmt ) ( post post1 posti : assertion) e,
  (∀ s1 s2 event, post1 s1 event s2 ∧ eval_expr s2 e = 0→
    post s1 event s2) →
  (∀ s p t event1 event2, eval_expr s e ≠ 0 → posti s event1 p →
    post1 p event2 t → post1 s (app event1 event2) t) →
  (∀ s, eval_expr s e = 0 → post1 s nil s ) →
  RULET st posti →
  RULET (While e st) post
| SeqRule: ∀ (stmt1 stmt2: stmt ) ( post1 post2 post: assertion),
  (∀ s1 s2 event1 event2, (∃ p , post1 s1 event1 p ∧ post2 p event2 s2) →
    post s1 (app event1 event2) s2) →
  RULET stmt1 post1 →
  RULET stmt2 post2 →
  RULET (Sseq stmt1 stmt2) post
| SkipRule: ∀ (post: assertion),
  (∀ (s1 s2: state), s1 = s2 → post s1 nil s2) →
  RULET Skip post
| SignalRule : ∀ (post: assertion) event,
  (∀ (s1 s2: state)event , s1 = s2 → post s1 (event :: nil) s2) →
  RULET (Signal event) post.
```

A.2.4 Logic for trace properties for the extended language

```
Inductive RULER ( : methPost) ( : methInv ) : stmt → invariant → Prop :=
| AffectRuleR : ∀ x e (post : invariant) ,
  (∀ (s1 : state) l, l = nil → post s1 l) →
  RULER (Affect x e) post
| IfRuleR : ∀ e (stmtT stmtF: stmt ) ( post1 post2 post : invariant) ,
  (∀ ( s1 : state) event,
    ((not (eval_expr s1 e = 0)) → post1 s1 event) ∧
    (eval_expr s1 e = 0 → post2 s1 event) → post s1 event) →
   (∀ (s1 : state) event, event = nil → post s1 event) →
   RULER stmtT post1 →
   RULER stmtF post2 →
   RULER (If e stmtT stmtF) post
| WhileRuleR : ∀ (st : stmt)(post post1: invariant) e (inv : assertion),
    (∀ s1 event, post1 s1 event → post s1 event) →
    (∀ (s1 : state) l, l = nil → post1 s1 l) →
    (∀ s , eval_expr s e = 0 → post1 s nil) →
    RULER st post1 →
    RULET st inv →
    (∀ s1 s2 e1 e2, ( inv s1 e1 s2 → eval_expr s1 e ≠ 0 →
      post1 s2 e2 → post1 s1 (app e1 e2) ) )-¿
    RULER (While e st) post
| SeqRuleR : ∀ (stmt1 stmt2: stmt)(post post1 postRst2 : invariant)
   (postT :assertion ),
   (∀ s1 e , ( post1 s1 e → post s1 e )) →
   (∀ s1 s2 e1 e2 , postT s1 e1 s2 → postRst2 s2 e2 →
      post1 s1 (app e1 e2 ) ) →
   RULER stmt1 post1 →
   RULET stmt1 postT →
   RULER stmt2 postRst2 →
   (∀ (s1 : state) l , l=nil → post s1 l ) →
   RULER (Sseq stmt1 stmt2) post
| SkipRuleR : ∀ (post: invariant),
  (∀ (s1 : state ) l, l = nil → post s1 l) →
  RULER Skip post
| CallRuleR : ∀ ( mName : methodNames ) ( post : invariant) ,
  (∀ (s1 : state ) event, ( mName ) s1 event → post s1 event) →
  (∀ (s1 : state) event , event= nil → post s1 event ) →
  RULER (Call mName ) post
| SignalRuleR: ∀ (post: invariant) event,
   ( ∀ (s1 : state ) l , l = nil → post s1 (event :: l) ) →
   ( ∀ (s1 : state ) l, l = nil → post s1 l ) →
   RULER (Signal event) post.
```