Elimination of ghost variables in program logics

Martin Hofmann and Mariela Pavlova

Institut für Informatik
LMU München

Abstract. 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 paper 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 paper ghost variables can be seen as a specification pattern rather than a primitive notion.

1 Introduction

With the fast development of programming systems, the requirements for software quality also become more complex. In reply to this, the techniques for program verification also evolve. This is the case also for modern specification languages which must support a variety of features in order to be expressive enough to deal with such complex program properties. A typical example is JML (short for Java Modeling Language), a design by contract specification language tailored to Java programs. JML has proved its utility in several industrial case studies [10, 7]. JML syntax is very close to the syntax of Java and it actually supports all Java expressions. JML has also 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* [9, 1–3] 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. In this case, the underlying logical formalism must have a very solid semantic basis so as to prevent inadvertent or malicious exploitation. On the 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 nevertheless establish not quite the intuitively intended property. This calls for a rigorous assessment of all the features employed in a specification language; in this paper we do this for JML's *ghost variables*.

In brief, 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 into 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.

Ghosts in internal assertions First, they can be used for an internal method annotation in order to facilitate the program verification process. For instance, in JML ghost variables can be used in an assertion to refer to the value of a program variable at some particular program point different from the point where the assertion is declared and must hold. Concretely, we can use for instance, 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 in order to count the number of loop iterations. Such use of ghost variables usually makes them appear in intra method assertions like loop invariants or assertions at a particular program point but does not introduce them in the contract of a method (i.e. the pre- and postcondition). For illustration, we consider an example which calculates the double of the variable x which is stored in the variable y:

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

y=0;
//@ghost int z;
//@set z = 0;
//@loop_invariant 2*(z - \old(z))= y-\old(y) && z+x = \old(z)+\old(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. We may notice that 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 is used to describe the invariant relation between x and y which holds at the borders of every loop iteration. On the contrary, the postcondition states the desired relation between the initial value of x and the final value of y without the use of the ghost z. Actually, the ghost variable z helps only to proving the program correct but not to express the program externally visible specification, i.e. its postcondition.

Expressing extra-functional code properties with ghosts. 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, we shall consider a fragment of a Java class with two ghost variables - MEM which counts the number of allocated heap space 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 (i.e. the value of the ghost variable MEM) with the allocated memory in the method (size(A) is the memory allocated by an object of class 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 either.

Therefore, ghost variables are particularly suitable when the code annotation is completely transparent, for example, for software auditing performed interactively over the source code, i.e. in the process where a code producer verifies if the written code respects their initial intentions. In such situations the good intuitions that ghost variables provide as opposed, perhaps, 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 [12] and as a means to enforce a particular order in which API methods should be invoked [11]. While our method also applies to those usages of ghost variables we limit ourselves to the former two in this paper for lack of space.

A critique of ghost variables As we said earlier, 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 and Jack 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 find that one should give meaning to ghost variables without altering the operational semantics of the language not even by adding non-existent variables to its memory model.

There is a second problem with ghost variables that shows up only when they are used to track extra-functional program properties like memory consumption

above, which is to do with the fact that the intended as opposed to the formal meaning of a contract then is contingent on respecting a particular code annotation policy. For the sake of 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 upper example, the client policy is expressed via the the ghost variable MEM which keeps track of the size of the allocated memory in every program state and thus, the component code and its specification can be :

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

In this case, of course nothing bad happens as the only method in class C does not allocate memory and it definitely respects the client requirements for not allocating memory. However, for some reason (deliberately or not), the implementor could have written also a malicious code (creates an object) with a wrong specification (does not increment 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 allocates new memory cells and thus, violates the client policy.

One could argue that this could 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 does capture 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 ultimate belief. Especially in situations where we assume the existence of malicious code producers who try to fool the code consumer with faked certificates we would prefer to reduce resorting to such non-rigorous methods to a bare minimum. Of course, if we are interested in extra-functional code properties we have to 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, we argue that this formalisation should be done openly by a trusted body of experts, and carefully argued by means of examples, test cases, etc. In brief, it is a procedure that 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 are thus brought back to essentially the first usage of ghost variables, namely as an auxiliary device employed to facilitate a verification.

1.1 Contributions of this paper

In this paper 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 definitional extension of ordinary program logics.

In order to focus on salient aspects we study the problem of ghost variables using first 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 whenever S, T are initial, respectively final states of a terminating run of program C then $P(S, T)$ holds.

Elimination of ghost variables We then 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 relations 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, when $P_g((S, S_g), (T, T_g)) \iff Q(S, T)$ 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 possible in internal assertions, e.g., as invariants in invocations of the proof rule for while-loops. The qualification "*effective*" of the announced 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 $f \in Extern$ 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 $Event := Extern \times \mathbf{Z}$.

Now that programs can cause observable effects already 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 as relation $\xrightarrow{C}$ where $S \xrightarrow{C,\boldsymbol{ev}} S'$ 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 $\boldsymbol{ev} \in Event^*$ have occurred.

We then consider a program logic that derives assertions of the form $C : P, I$ with the intention that whenever $S \xrightarrow{C,\boldsymbol{ev}} S'$ then $I(S, ev)$ will hold (the "system invariant") and if, moreover, it happens that S' is a terminal state, then $P(S, \boldsymbol{ev}, S')$ (the "contract") will hold, too. Both invariant and contract may thus speak about observable events occurred; functional properties are specified in the contract alone.

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 assertion "$\texttt{MEM} = 0$" 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 assertion $\texttt{MEM} = mem(\boldsymbol{ev})$ where $mem(\boldsymbol{ev})$ is the number of allocation events in execution trace *tr*. Combining the two proofs then yields a proof of the assertion $mem(\boldsymbol{ev}) = 0$ 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.

Acknowledgement We acknowledge support by the EU integrated project MOBIUS IST 15905.

2 Preliminaries

2.1 Simple programming language

We shall consider a simple programming language with the traditional constructs 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 paper, we shall use a Coq syntax for introducing definitions. The upper Coq code is an inductive type with several constructors and every one of them correspond to the different statements of the language. This definition corresponds to the following more common notation:

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

Thus, an expression in the language are program variables, integer constants and arithmetic expressions but here we do not give explicitly their syntax. Values in our language are integers. Our formal Coq development comprises recursive methods; we elide them here for the sake of simplicity. We give a standard big step operational semantics which characterises the terminating executions of program statements. It is defined as an relation between initial and final state of the execution of statement where states are mappings from variables to values:

exec_t : *state* → *stmt* → *state* → *Prop*

The inductive definition is of *exec_t* is given in the appendix.

2.2 Logic for partial correctness for a simple language

The partial correctness logic is formulated in a VDM style. Differently from the Hoare style rules, where assertions are functions of the current state to a boolean value, 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 used for relating the values of variables in different states, see [8]. The logic is encoded in Coq as an inductive predicate

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

where we find one constructor for each proof rule, see Appendix.

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

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

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), then 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 $Gassertion := state \to gState \to state \to gState \to Prop.$

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

Inductive $GRULE$: $Gstmt \to Gassertion \to Prop$

The rules for the ghost language are quite the same as the rules for the standard simple language except that those are defined for assertions with ghost variables. For instance, the rule for program assignment in the type *GRULET* is basically the same as its counterpart *RULET* modulo the presence of the ghost states:

$GAssignRule$: $\forall\ x\ e\ (post : Gassertion)$,
$(\forall\ (s1\ s2 : state)\ (g1\ g2 : gState)$,
$g1 = g2 \to s2 = update\ s1\ x\ (eval\_expr\ s1\ e) \to post\ s1\ g1\ s2\ g2) \to$
$GRULE\ (GAssign\ x\ e)\ post$

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):

$GSetRule : \forall\ x\ (e : gExpr)\ (post : Gassertion)$,
$(\forall\ (s1\ s2 : state\ )\ (g1\ g2 : gState)$,
$g2 = gUpdate\ g1\ x\ (gEval\_expr\ s1\ g1\ e) \to s1 = s2 \to$
$post\ s1\ g1\ s2\ g2) \to GRULE\ (GSet\ x\ e)\ post.$

3.1 Example

Let us return back to our first example in the introductory part which calculates the double of the input variable x.

If we imagine that that program is a code to be sent over the web to another computer system along with its certificate, the code producer may tend to use ghosts and a ghost logical system (as shown in the example) as those provide more intuition in the annotation and in the proof process eventhough ghost variables do not have a clear meaning in the operational semantics presented in Section 2.2. However, the client system would be most probably equipped with a verification algorithm sound w.r.t. to a rigorous operational semantics (e.g.

the operational semantics presented in Section 2.1) as for instance, the standard logic from Section 2.2. A Proof Carrying Code architecture will hardly work if the code producer uses one proof technology for the certificate generation and the client uses another one for checking the certificate. However, there exists a relation between the two logical systems which allows for this. The relation shows that if a certificate in the ghost logic exists then a certificate into the standard logic also exists.

3.2 Relation between ghost and standard logic

In the sequel we use a function $\mathit{transform} : \mathit{Gstmt} \rightarrow \mathit{Stmt}$ that returns the underlying standard program by replacing all ghost assignments with skips.

Next, with each ghost assertion ψ (of type *Gassertion*) we associate a standard assertion $\mathit{transform}(\psi)$ (of type *assertion*) by

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

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 $\mathit{transform}(\mathit{stmt})$ meets the assertion $\mathit{transform}(\psi)$.

Notice that if ψ does not mention ghost variables then $\mathit{transform}(\psi)$ is equivalent to ψ itself.

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 back 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. Probably, turning the specification developer mind to use such universal and existential quantification is difficult. Actually, the above lemma shows that we can use

ghosts and a ghost logic to construct the program proof of correctness even for critical applications where the soundness of the logic w.r.t. to a rigorous semantics is important. This is because from such a program proof we can construct another one in a standard sound logic.

Note that the ghost logic may be used to reason about specifications which do not necessarily refer to ghost variables. In particular, if a ghost program and ghost specification but which do not speak about ghosts are provable in the ghost logic defined with *GRULET* then their respective counterparts in the standard programming and specification languages are provable in *RULET*. This is actually a conservativity property for the ghost logic w.r.t. the standard logic. This is a direct consequence from the previous lemma and its formal statement follows:

Lemma *conservative*: $\forall$ (*s*: *Gstmt*) (*post* : *assertion*),
GRULE s (*fun* (*s1*:*state*)(*g1*:*gState*)(*s2*:*state*)(*g2*:*gState*) $\Rightarrow$ *post s1 s2*) $\rightarrow$
RULE (*transform s*) *post*.

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 sohw 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 [4, 11] 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* $\rightarrow$ *stmt* $\rightarrow$ *list event* $\rightarrow$ *state* $\rightarrow$ *Prop*

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

Inductive *t_exec*:*state* $\rightarrow$ *stmt* $\rightarrow$ *list event* $\rightarrow$ *state* $\rightarrow$ *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 correct ness 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*.

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*

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 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 [6] and the use of existential quantification in situations where ghost variables might appear [5]. 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 paper. We found our approach very robust and did not experience obstacles with the inclusion of methods, even recursive, as well as with the other uses of ghost variables [12, 11] 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.

References

1. Andrew W. Appel. Foundational proof-carrying code. In *Logic in Computer Science*, 2001.
2. David Aspinall, Stephen Gilmore, Martin Hofmann, Donald Sannella, and Ian Stark. Mobile resource guarantees for smart devices. In *CASSIS*, pages 1–26, 2004.
3. 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: Proceedings of the second symposium on Trustworthy Global Computing*, LNCS. Springer-Verlag, 2006.
4. B. Beckert and W. Mostowski. A program logic for handling java card's transaction mechansim, 2002.
5. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
6. Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. *Lecture Notes in Computer Science*, 2083:626+, 2001.

7. Néstor Cataño and Marieke Huisman. Formal specification and static checking of gemplus' electronic purse using ESC/java.
8. Bart Jacobs, Claude Marché, and Nicole 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.
9. Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.
10. George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.
11. 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, *Proceedings of CARDIS'04*, Toulouse, France, August 2004. Kluwer Academic Publishers.
12. K. Rustan, M. Leino, and P. uller. Object invariants in dynamic contexts, 2004.

A Functional Behaviours

A.1 Syntax of the Language

Inductive $exec\_t : state \to stmt \to state \to Prop :=$
| $ExecAssign : \forall\ s\ x\ e,$
  $exec\_t\ s\ (Assign\ x\ e)\ (update\ s\ x\ (eval\_expr\ s\ e))$
| $ExecIf\_true : \forall\ s1\ s2\ e\ stmtT\ stmtF,$
  $eval\_expr\ s1\ e \neq 0 \to$
  $exec\_t\ s1\ stmtT\ s2 \to$
  $exec\_t\ s1\ (If\ e\ stmtT\ stmtF)\ s2$
| $ExecIf\_false : \forall\ s1\ s2\ e\ stmtT\ stmtF,$
  $eval\_expr\ s1\ e = 0 \to$
  $exec\_t\ s1\ stmtF\ s2 \to$
  $exec\_t\ s1\ (If\ e\ stmtT\ stmtF)\ s2$
| $ExecWhile\_true : \forall\ s1\ s2\ s3\ e\ stmt,$
  $eval\_expr\ s1\ e \neq 0 \to$
  $exec\_t\ s1\ stmt\ s2 \to exec\_t\ s2\ (While\ e\ stmt)\ s3 \to$
  $exec\_t\ s1\ (While\ e\ stmt)\ s3$
| $ExecWhile\_false : \forall\ s1\ e\ stmt,$
  $eval\_expr\ s1\ e = 0 \to$
  $exec\_t\ s1\ (While\ e\ stmt)\ s1$
| $ExecSseq : \forall\ s1\ s2\ s3\ i\ stmt1\ stmt2,$
  $exec\_t\ s1\ stmt1\ s2 \to$
  $exec\_t\ s2\ stmt2\ s3 \to$
  $exec\_t\ s1\ (Sseq\ stmt1\ stmt2)\ s3$
| $ExecSkip : \forall\ s,\ exec\_t\ s\ Skip\ s.$

A.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.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.
```

B Extra-functional behaviours with traces

B.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 .
```

B.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.
```

B.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.
```

B.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 ) →
```

$RULER\ (Call\ mName\ )\ post$
$|\ SignalRuleR$: $\forall\ (post$: $invariant)\ event$,
$(\ \forall\ (s1\ :\ state\ )\ l\ ,\ l = nil \rightarrow post\ s1\ (event :: l)\ ) \rightarrow$
$(\ \forall\ (s1\ :\ state\ )\ l,\ l = nil \rightarrow post\ s1\ \ l\ ) \rightarrow$
$RULER\ (Signal\ event)\ post.$