Elimination of Ghost Variables in Program Logics

Martin Hofmann[1] and Mariela Pavlova[2]

[1] Institut für Informatik LMU München, Germany
[2] Trusted Labs, Sophia-Antipolis, France
hofmann@ifi.lmu.de, Mariela.Pavlova@trusted-labs.fr

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 [1,2]. Other examples are ESC/Java [3], the Larch methodology [4] and Spec# [5]. JML syntax is very close to the syntax of Java. 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* [6,7,8,9] 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*.

G. Barthe and C. Fournet (Eds.): TGC 2007, LNCS 4912, pp. 1–20, 2008.
© Springer-Verlag Berlin Heidelberg 2008

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/ghostcoq.tgz`.

2 Ghost Variables and Their Use

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. Note that ghost variables should not interfere with the values of normal program variables or the control flow of a program.

Ghost variables appear to ease proof automation in automatic theorem provers like Simplify as they instantiate existential quantification by pointing the object which satisfies the otherwise existentially quantified proposition. On the other hand, they are intuitive and thus, helpful in the specification process.

2.1 Ghost Variables 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. Thus, 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 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 doubles the value 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. 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 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.

2.2 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, let us 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 <= MAX
public void m () {
  A a = new A ()
 //@ set MEM = MEM + size (A)}
```

The postcondition asserts that MEM is bounded by MAX which ensures bounded memory allocation provided that MEM accurately tracks the number of allocations made. In the example this is ensured by the assignment to MEM.

We notice that this 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 [10] and as a means to enforce a particular order in which API methods should be invoked [11].

3 Problems with Ghost Variables

In this section we describe why we feel that ghost variables as currently used and modelled might be harmful in a proof carrying code scenario where formal proofs are provided as certificates by untrusted and possibly even malicious code producers.

3.1 Semantics of Ghost Variables

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 [3] and Jack [12] treat ghost variables as ordinary program variables. While this works in order to generate verification conditions and justify proof rules, it is not entirely satisfactory 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 adequacy of the last formalised semantics in such a chain of translations always remains an unprovable axiom. 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.

One may argue that one could prove a semantics adequate by formally relating it to assembly language level formalizations of hardware architectures. As argued above this only shifts the "semantic gap" somewhere else. What remains unproven in that case is that the assembly language level formalization does indeed adequately reflect what's going on in the hardware and also that the assumed translation of bytecode to assembly code is what is really done by the compiler and the JVM implementation.

3.2 Modelling Extra-Functional Properties

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 someone advertises a Java card game to be run on a mobile phone and claims that it definitely runs within 10M of memory by providing a formal proof that its main-method satisfies the postcondition

```
ensures MEM <= 10485760
```

Unfortunately, such a formal proof only guarantees that the ghost variable MEM has a value $\leq 10M$ after the execution of the program; the fact that it relates to memory allocation remains unproved. Thus, n order to be really sure that the program really does not use more than 10M the recipient would have to carefully study the code to make sure that indeed the ghost variable MEM has been appropriately incremented at every allocation site and not been tempered with anywhere else. Not only does this place an awkward burden on code recipients; it also opens to door to all kinds of exploits by malicious code producers based on somehow hiding assignments to MEM in obcure library functions or similar.

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 a successful run of some automatic analysis which checks 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 ultimately belief. 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.

Before continuing, we emphasize again that there is nothing wrong with ghost variables in a verification tool or formalism. It is only in the scenario of proof-carrying code where we intend to use proofs in formalised program logic as unforgeable certificates that our discussion applies and our results are of value.

4 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 conservative 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.

4.1 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.

4.2 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 $\overset{C}{\to}$ where $S \overset{C,\boldsymbol{ev}}{\to} S'$ 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 $\boldsymbol{ev} \in Event^*$ have occurred.

We then consider a program logic that in addition to VDM-style assertions (which now, of course, may also mention the trace of events occurred) also has a judgement $C : I$ with the intention that whenever $S \overset{C,\boldsymbol{ev}}{\to} S'$ 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` = `\old(MEM)`" 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` $= \mathit{mem}(\boldsymbol{ev})$ where $\mathit{mem}(\boldsymbol{ev})$ is the number of allocation events in execution trace tr. Combining the two proofs then yields a proof of the invariant $\mathit{mem}(\boldsymbol{ev}) = 0$ to which elimination of ghost variables applies.

5 Language and Program Logic

In this section we define a simple programming language and a VDM style program logic as a vehicle for a formalisation of our results. The language has neither local variables nor objects, yet the salient features of our modelling of ghost variables can be sufficiently well illustrated therein without introducing unnecessary clutter.

5.1 Simple Programming Language

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

Inductive $\mathit{stmt} : \mathit{Type} :=$
$\mid \mathit{Assign} : \mathit{var} \rightarrow \mathit{expr} \rightarrow \mathit{stmt}$
$\mid \mathit{If} : \mathit{expr} \rightarrow \mathit{stmt} \rightarrow \mathit{stmt} \rightarrow \mathit{stmt}$
$\mid \mathit{While} : \mathit{expr} \rightarrow \mathit{stmt} \rightarrow \mathit{stmt}$
$\mid \mathit{Sseq} : \mathit{stmt} \rightarrow \mathit{stmt} \rightarrow \mathit{stmt}$
$\mid \mathit{Skip} : \mathit{stmt}.$

Here and in the rest of the paper, 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:

$\mathit{stmt} :=$
$\mid \mathit{Assign}\ (\mathit{var}\ \ \mathit{expr})$
$\mid \mathit{If}\ \ (\mathit{expr}\ \mathit{stmt}\ \mathit{stmt})$
$\ldots$

We elide the syntax of arithmetic expressions. Values in our language are integers. Our formal Coq development comprises recursive methods; we omit 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 a relation between initial and final states of statement execution where states are mappings from variables to values:

$\mathit{exec} : \mathit{state} \rightarrow \mathit{stmt} \rightarrow \mathit{state} \rightarrow \mathit{Prop}$

The inductive definition of *exec* is given in the Appendix.

5.2 Logic for a Simple Language

The partial correctness logic is formulated in a VDM style [13]. 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 $assertion := state \rightarrow state \rightarrow Prop.$

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 [14]. The logic is encoded in Coq as an inductive predicate with one constructor for each proof rule, see Appendix:

Inductive $RULET$: $stmt \rightarrow assertion \rightarrow Prop$

The soundness theorem is standard and establishes that a derivation over program and judgement implies that every execution of the program satisfies the judgement:

Proposition 1 (Soundness of partial logic)
∀ (st: stmt) (s1 s2 : state),
exec s1 st s2 →∀ (post : assertion), RULET st post → post s1 s2.

6 Introducing 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, assertions with ghost variables ("ghost assertions" for short) *Gassertion* are mappings from the initial and final program states and also from the initial and final ghost states to a truth value:

Definition $Gassertion := state \rightarrow gState \rightarrow state \rightarrow gState \rightarrow Prop.$

Now we define inductively a logic for ghost assertions:

Inductive $GRULET$: $Gstmt \rightarrow Gassertion \rightarrow Prop$

The rules for this logic are quite the same as the rules for the standard simple language except that those are defined for assertions with ghost variables. Consider e.g. the assignment rule which differs from the ordinary assignment rule only in that ghost states are threaded through and required not to change.

$GAssignRule$: $\forall\ x\ e\ (post : Gassertion)$,
$(\forall\ (s1\ s2 : state)\ (g1\ g2 \text{: } gState)$,
$g1 = g2 \rightarrow s2 = update\ s1\ x\ (eval\_expr\ s1\ e) \rightarrow post\ s1\ g1\ s2\ g2) \rightarrow$
$GRULET\ (GAssign\ x\ e)\ post$

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):

$GSetRule : \forall\ x\ (e : gExpr)\ (post : Gassertion)$,
$(\forall\ (s1\ s2 : state\ )\ (g1\ g2 \text{: } gState)$,
$g2 = gUpdate\ g1\ x\ (gEval\_expr\ s1\ g1\ e) \rightarrow s1 = s2 \rightarrow$
$post\ s1\ g1\ s2\ g2) \rightarrow GRULET\ (GSet\ x\ e)\ post.$

We do not prove the soundness of this logic w.r.t. an operational semantics instrumented with ghost variables. As we pointed out in the introductory part ghost variables lack a physical meaning and thus, should not be present in the program semantics. Actually, the relation between the ghost and standard logic presented in the following justifies the ghost logic w.r.t. standard operational semantics *exec* presented in the Appendix.

6.1 Relation between Ghost and Standard Logic

In the sequel we use a function $transform : Gstmt \rightarrow Stmt$ 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 $transform(\psi)$ (of type *assertion*) by

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

Notice that if ψ does not mention ghost variables then $transform(\psi)$ 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 $transform(\psi)$:

Theorem 1 (Elimination of ghosts)
$\forall$ (gst: Gstmt) (Gpost: Gassertion),
let st := transform gst in
let post:= (fun s1 s2 $\Rightarrow$ $\forall$ (sg1: gState),$\exists$ sg2:gState, Gpost s1 sg1 s2 sg2) in
$GRULET$ gst Gpost $\rightarrow$ $RULET$ st (fun s1 s2 $\Rightarrow$ 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 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. Of course, it would have been possible to use such existentially quantified invariant in the first place or even cleverly guess the logically 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.

Corollary 1 (Conservativity of ghosts). ∀ *(s: Gstmt) (post : assertion), GRULETstmt (fun (s1:state)(g1:gState)(s2:state)(g2:gState) ⇒ post s1 s2)→ RULET(transform stmt) post.*

Remark on terminology. What we (and the JML community) call ghost variables is in other situations known as *auxiliary variables*, in particular in the context of Jones' rely-guarantee methodology and also in Reynolds' standard reference [15]. There, the term *ghost parameter* is reserved for what we call auxiliary variables, namely universally quantified parameters used to fix old values of variables. In a Hoare-style logic such auxiliary variables (in our sense) are crucial to express that certain program variables are not modified. In our VDM-style version where assertions have explicit access to pre-states such auxiliary variables or ghost parameters in Reynolds' sense are not needed as pointed out by Kleymann [14] and thus not considered in this paper.

It is on the other hand not possible to use ghost variables to get rid of explicit access to pre-states (or auxiliary variables when using Hoare-style logic). Of course, one can use a ghost variable to store the old value of some variable, but then we cannot — in the absence of \old that is — stipulate in the contract that this ghost variable remains itself unmodified.

7 Ghost Variables for Extra-Functional Properties

So far, we have seen the meaning of ghost variables w.r.t. a standard partial correctness. Such formulation 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 [16,11] 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. Recall that we assume a set *event* modelling observable events, e.g., calls to API methods.

Inductive *reach*: $state \rightarrow stmt \rightarrow list\ event \rightarrow state \rightarrow Prop$

The proposition $reach(\sigma_0, stmt, evs, \sigma_1)$ means that the execution of *stmt* started in state σ_0 reaches the state σ_1 and produces the list of events *evs*. 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 in the usual big step style but this time 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 *reach* and *t_exec* are given in the Appendix. These two definitions delineate the behaviour of extra-functional properties of programs. It is they that must be "openly reviewd by a trusted body experts" so as to ensure that they adequately model the physical behaviour of programs.

Next, we define a logic that allows us to reason about and certify properties of this extra-functional behaviour. The format of the logic relies on relations between pre- and post states for terminating programs and *invariants* delineating the behaviour of possibly nonterminating programs.

Definition *invariant* $:= state \rightarrow list\ event \rightarrow Prop.$

The logic which allows to reason over trace properties is defined in Coq as the inductive type $RULER$(see Appendix). The trace logic uses the logic for partial correctness $RULET$ presented in the previous Section 6 but which is suitably modified to deal with event traces. In particular, the assertions that $RULET$ manipulates now depend not only on the initial and final state but also on the trace of events produced during execution:

Definition *assertion* $:= state \rightarrow list\ event \rightarrow state \rightarrow Prop.$

The soundness statement of the logic requires that if satisfaction of an invariant by a statement is derivable in $RULER$ then every reachable state of the execution of that statement satisfies the invariant:

Proposition 2 (Soundness of trace logic). $\forall$ *stmt (s1 s2 : state) events, (reach s1 stmt events s2)* $\rightarrow \forall$ *inv, RULER stmt inv* $\rightarrow$ *inv s1 events .*

The proof of that 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 $RULET$.

7.1 Program Logic for Trace Properties and Ghost Variables

The logic for trace properties tailored to a language with ghost variables is analogous to the logic for trace properties for a standard language presented in the previous subsection. 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* $\rightarrow$ *gState* $\rightarrow$ *list event* $\rightarrow$ *gState* $\rightarrow$ *Prop.*

Definition *Gassertion* := *state* $\rightarrow$ *gState* $\rightarrow$ *list event* $\rightarrow$ *state* $\rightarrow$ *gState* $\rightarrow$ *Prop*

We now have the following relationship allowing us to effectively eliminate all reference to ghost variables from a proof in the trace logic.

Theorem 2 (Elimination of ghosts from trace logic). $\forall$ *(gstmt:Gstmt) (ginv: Ginvariant),*
let stmt := transform gstmt in
let inv := (fun s1 event $\Rightarrow \forall$ *g1,* $\exists$ *g2, ginv s1 g1 event g2) in*
RULERG gstmt ginv $\rightarrow$ *RULER stmt inv.*

7.2 Example Application

In the example of a mobile phone application purportedly using at most 10M of memory we would insist on the invariant

meminv aState evs = *mem evs* ≤ 10485760

where *mem* : *list event* $\rightarrow$ *nat* is a Coq function extracting the number of allocations from a list of events.

Suppose now that someone has already established the following ghost invariant for the program enriched with assignments to a ghost variable *MEM* purportedly tracking memory allocations thus expressing that at all times the ghost variable *MEM* is at most 10M.

gmeminv aState aGState evs aGState = *aGState.MEM* ≤ 10485760

If indeed the program has been appropriately decorated with assignments to *MEM* at allocation sites and *MEM* has not been tampered with elsewhere then one can automatically produce proof of the invariant

$gcorrinv\ aState\ aGState\ evs\ aGState = aGState.MEM = mem\ evs$

Thus, we obtain a proof of the combined invariant

$ginv\ aState\ aGState\ evs\ aGState = gcorrinv\ aState\ aGState\ evs\ aGState \wedge gmeminv\ aState\ aGState\ evs\ aGState$

Applying the above lemma relating standard and ghost logic we obtain a standard invariant from which *meminv* readily follows.

8 Ghost Variables for Object Invariants

In object oriented languages like Java, it is useful to talk about an object invariant, i.e. a property that must hold in all visible states of an object and on which other objects may rely. For instance, an invariant of objects of class representing a list data structure is that the field length should be always greater or equal 0. Intuitively, the visible states for an object are the initial and final state of every method in the program, see [17] and the object invariant is usually a relation between the components of the object, i.e. the instance fields of the object. In particular, this means that the invariant of an object can be broken during the execution of a method of this object. In order to verify that an object invariant holds it must be proven to hold at the borders of every method in the program, i.e. it is desugared as part of the pre and postcondition of every method in the program. This implies that an object invariant must be "revealed" to all classes in the program and yields the problem of representation exposure. To remedy this, in [10] Barnett et al. describe a modular and sound verification scheme for object invariants based on ghost variables. To do so, a boolean ghost variable *valid* is attached to every object *o*. The correctness of the methodology relies on the following invariant property *INV*[1] in every execution state:

$$INV = \text{If } o.valid \text{ is true then the property } o.I \text{ holds} \qquad (1)$$

Now, the clients of an object *o* can be informed about the validity of *o.I* just by revealing to them the value of the ghost field *o.valid* and avoiding thus representation exposure. In order to enforce this invariant, the specification language is extended with two more constructs - *o.pack* and *o.unpack* which basically set the object state variable *o.valid* and thus mark the region in the program where the object invariant can be broken or not. We have shown that a proof in such verification scheme can be translated in a standard programming logic and in the following, we sketch this.

[1] We focus here on the first part of the article where the authors consider that an object invariant may talk only about the fields of the object.

Let us consider without losing the generality of the problem that we have a simple imperative program provided with an invariant I and a ghost logic formalised in Coq with the inductive type *GHRULE* (see Appendix) based on the principle described in [10]. Because we limit ourselves to a simple language *INV* will look rather like this:

$$INV = \text{If } valid \text{ is true then the property } I \text{ holds} \qquad (2)$$

The rule for assignment should establish that if the invariant *INV* holds in the prestate of the assignment then it will hold after the assignment and the rule will look like this in the Coq system:

```
Inductive GHRULE INV :=
| GHAssignRule: ∀ x e (pre: Gpreassertion)(post: Gassertion),
   (∀ (s1 s2: state) (g1 g2: gState), pre s1 g1 →INV s1 g1 →
   g1 = g2 → s2 = update s1 x (eval_expr s1 e) →
   INV s2 g2 ∧ post s1 g1 s2 g2) → GHRULE INV pre (GAssign x e) post.
```

No proof obligations pertaining to invariants arise in the other rules; see e.g. the rule *GHWhileRule* in Appendix C. However, at all times the invariant can be invoked as formalised by rule *GHInvRule*. This system thus formalises the reasoning scheme proposed in [10].

We have formalised in Coq how a proof in such a verification scheme can be transformed in a proof over a language without ghost variables. The key lemma establishes that a proof in a logic which supports the invariant relation (2) can be transformed into a proof in a partial VDM logic with ghost variables which establishes that the invariant I is preserved:

Theorem 3 (Invariants). ∀ *gstmt pre post INV,*
GHRULE INV pre gstmt post →
*GRULETgstmt (*fun *s1 g1 s2 g2* ⇒ *pre s1 g1* ∧ *INV s1 g1* →
post s1 g1 s2 g2 ∧ *INV s2 g2).*

The lemma is proved by induction over *GHRULE*. Combined with the result described in Section 6, we can conclude that we can transform a proof in a logic with invariants into a standard one.

9 Conclusion and Further Work

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 [18] and the use of existential quantification in situations where ghost variables might appear [19]. 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 to be very robust and did not experience obstacles with the inclusion of recursive methods. 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.

Although orthogonal to the idea of transformation of ghost proofs, our work raises the question if high level specification languages like JML should use ghost variables to model extrafunctional properties. An application of our result can be the extension of JML with special non assignable constructs to denote extrafunctional properties which will benefit of a clear semantics and sound verification framework described in the present article.

In this article we have not covered the use of ghost variables in specification and verification of shared-variable concurrency. Indeed, formalisms such as Owicki-Gries [20] are incomplete without ghost variables. We found that this is due to the fact that in the standard formulation of, e.g., Owicki-Gries, given access to the entire state of the system which includes local variables and program counters of all processes. If such access is provided, ghost variables can, again, be eliminated using the methods from this paper. It is, however, questionable whether assertions should be allowed to mention the global state; indeed, we find that the real issue behind the phenomena around ghost variables in concurrency is the question of how one should specify a stateful component without revealing its internal implementation. We leave a detailed investigation of these questions for future work.

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

References

1. Cataño, N., Huisman, M.: Formal specification of Gemplus' electronic purse case study using ESC/Java. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 272–289. Springer, Heidelberg (2002)
2. Jacobs, B., Marché, C., Rauch, N.: Formal verification of a commercial smart card applet with multiple tools. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, Springer, Heidelberg (2004)
3. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, Berlin, Germany, pp. 234–245. ACM Press, New York (2002)
4. Guttag, J.V., Horning, J.J. (eds.): Larch: Languages and Tools for Formal Specification. In: Garland, S.J., Jones, K.D., Modet, A., Wing, J.M. (eds.) Texts and Monographs in Computer Science, Springer, Heidelberg (1993)
5. Barnett, M., Leino, K., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 151–171. Springer, Heidelberg (2005)

6. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL 1997), Paris, pp. 106–119 (1997)
7. Appel, A.W.: Foundational proof-carrying code. In: Proc. IEEE Symp. Logic in Computer Science (LICS 2001) (2001)
8. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile resource guarantees for smart devices. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 1–26. Springer, Heidelberg (2005)
9. Barthe, G., Beringer, L., Crégut, P., Grégoire, B., Hofmann, M., Müller, P., Poll, E., Puebla, G., Stark, I., Vétillard, E.: Mobius: Mobility, ubiquity, security. objectives and progress report. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, Springer, Heidelberg (2007)
10. Barnett, M., Deline, R., Fähndrich, M., Rustan, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)
11. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.L.: Enforcing high-level security properties for applets. In: Paradinas, P., Quisquater, J.J. (eds.) Proceedings of CARDIS 2004, Toulouse, France, Kluwer Academic Publishers, Dordrecht (2004)
12. Burdy, L., Requet, A., Lanet, J.L.: Java applet correctness: A developer-oriented approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 422–439. Springer, Heidelberg (2003)
13. Jones, C.: Systematic Software Development Using VDM. Prentice Hall, Englewood Cliffs (1990)
14. Kleymann, T.: Hoare logic and auxiliary variables. Formal Aspects of Computing 11(5), 541–566 (1999)
15. Reynolds, J.C.: The craft of programming. Prentice Hall (1981); Out of print. Available as PDF from John Reynolds' home page
16. Beckert, B., Mostowski, W.: A program logic for handling java card's transaction mechanism. In: Pezzé, M. (ed.) ETAPS 2003 and FASE 2003. LNCS, vol. 2621, pp. 246–260. Springer, Heidelberg (2003)
17. Leavens, G.T., et al.: Jml reference manual
18. Beckert, B., Schlager, S.: A sequent calculus for first-order dynamic logic with trace modalities. LNCS, vol. 2083, p. 626 (2001)
19. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. In: Beckert, B., Hähnle, R., Schmitt, P.H. (eds.) Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334, Springer, Heidelberg (2007)
20. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta Inf. 6, 319–340 (1976)

A Functional Behaviours

A.1 Big Step Operational Semantics for a Simple Language

```
Inductive exec_stmt: state → stmt → state → Prop :=
 | ExecAssign: ∀ s x e,
     exec_stmt s (Assign x e)(update s x (eval_expr s e))
 | ExecIf_true: ∀ s1 s2 e stmtT stmtF,
     eval_expr s1 e ≠ 0 → exec_stmt s1 stmtT s2 →
     exec_stmt s1 (If e stmtT stmtF) s2
 | ExecIf_false: ∀ s1 s2 e stmtT stmtF,
     eval_expr s1 e = 0 → exec_stmt s1 stmtF s2 →
     exec_stmt s1 (If e stmtT stmtF) s2
 | ExecWhile_true: ∀ s1 s2 s3 e stmt,
     eval_expr s1 e ≠ 0 → exec_stmt s1 stmt s2 →
     exec_stmt s2 (While e stmt) s3 →
     exec_stmt s1 (While e stmt) s3
 | ExecWhile_false : ∀ s1 e stmt,
     eval_expr s1 e = 0 → exec_stmt s1 (While e stmt) s1
 | ExecSseq: ∀ s1 s2 s3 i stmt,
     exec_stmt s1 i s2 → exec_stmt s2 stmt s3 →
     exec_stmt s1 (Sseq i stmt) s3
 | ExecSkip: ∀ s, exec_stmt 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 (P: program)(B: body): state → stmt → list event → state → Prop :=
 | ExecAssign: ∀ s x e,
    t_exec P B s (Assign 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 :=
 | AssignRule : ∀ x e (post : assertion) ,
    (∀ (s1 s2: state), s2 = update s1 x (eval_expr s1 e) → post s1 nil s2) →
    RULET (Assign 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 event, s1 = s2 → post s1 (event :: nil) s2) → RULET (Signal event) post.
```

B.4 Logic for Trace Properties for the Extended Language

```
Inductive RULER: stmt → invariant → Prop :=
| AssignRuleR: ∀ x e (post: invariant),
    (∀ (s1: state) l, l = nil → post s1 l) → RULER (Assign x e) post
| IfRuleR: ∀ e stmtT stmtF (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
| SignalRuleR: ∀ (post: invariant) event,
    (∀ s1 l, l = nil → post s1 (event::l)) →
    (∀ s1 l, l = nil → post s1 l) → RULER (Signal event) post.
```

C Logic for Dealing with Invariants

```
Inductive GHRULE(I: Invariant): Gpreassertion → Gstmt → Gassertion → Prop :=
| GHAssignRule: ∀ x e (pre: Gpreassertion)(post: Gassertion),
    (∀ (s1 s2 : state)(g1 g2: gState), pre s1 g1 → I s1 g1 →
    g1 = g2 → s2 = update s1 x (eval_expr s1 e) → I s2 g2 ∧ post s1 g1 s2 g2) →
    ¡¡¡¡¡¡¡ .mine GHRULE I pre (GAssign x e) post
  ======= GHRULE I pre (GAssign x e) post
  ¿¿¿¿¿¿¿ .r240  | GHIfRule: ∀ e (stmtT stmtF: Gstmt)(pre pre1 pre2: Gpreassertion)
    (post1 post2 post: Gassertion),
    (∀ s gs, pre s gs → pre1 s gs) → (∀ s gs, pre s gs → pre2 s gs) →
    (∀ (s1 s2 : state ) (g1 g2: gState),
     (not (eval_expr s1 e = 0) → post1 s1 g1 s2 g2) ∧
     (eval_expr s1 e = 0 → post2 s1 g1 s2 g2) → post s1 g1 s2 g2) →
    GHRULE I pre1 stmtT post1 →
    GHRULE I pre2 stmtF post2 →
    GHRULE I pre (GIf e stmtT stmtF) post
```

```
| GHWhileRule: ∀ (stmt: Gstmt)(pre inv: Gpreassertion)
    (post1 post : Gassertion) e ,
    (∀ s gs, pre s gs → inv s gs) →
    (∀ (s1 s2: state)(g1 g2: gState), post1 s1 g1 s2 g2 → post s1 g1 s2 g2 ) →
    (∀ (s1 s2: state)(g1 g2: gState),
     ((inv s1 g1 → inv s2 g2) ∧ eval_expr s2 e = 0 → post1 s1 g1 s2 g2)) →
    GHRULE I (fun s1 g1 ⇒ eval_expr s1 e ≠ 0) stmt
                              (fun s1 g1 s2 g2 ⇒ inv s1 g1 → inv s2 g2) →
   GHRULE I pre (GWhile e stmt) post
| GHSeqRule: ∀ (stmt1 stmt2: Gstmt)(pre pre1 pre2 : Gpreassertion)
    (post1 post2 post: Gassertion),
    (∀ s gs, pre s gs → pre1 s gs) →
    (∀ s1 s2 g1 g2, (∃ p, ∃ gp, post1 s1 g1 p gp ∧ post2 p gp s2 g2) →
     post s1 g1 s2 g2 ) →
    GHRULE I pre1 stmt1 (fun s1 g1 s2 g2 ⇒ pre2 s2 g2 ∧ post1 s1 g1 s2 g2 ) →
    GHRULE I pre2 stmt2 post2 →
    GHRULE I pre (GSseq stmt1 stmt2) post
| GHSkipRule: ∀ (pre pre1: Gpreassertion)(post1 post: Gassertion),
    (∀ s gs, pre s gs → pre1 s gs) →
    (∀ (s1 s2: state)(g1 g2: gState), post1 s1 g1 s2 g2 → post s1 g1 s2 g2 ) →
    (∀ (s1 s2: state)(g1 g2: gState), g1 = g2 ∧ s1 = s2 → post s1 g1 s2 g2 ) →
    GHRULE I pre GSkip post
| GHSetRule: ∀ x (e: gExpr)(pre: Gpreassertion)(post: Gassertion),
    (∀ (s1 s2 : state)(g1 g2: gState), pre s1 g1 → I s1 g1 →
        g2 = gUpdate g1 x (gEval_expr s1 g1 e) ∧ s1 = s2 → I s2 g2 ∧ post s1 g1 s2 g2 ) →
    GHRULE I pre (GSet x e) post
| GHInvRule : ∀ stmt (pre : Gpreassertion) (post : Gassertion) ,
    GHRULE I (fun s1 g1 ⇒ I s1 g1 ∧ pre s1 g1 ) stmt post →
    GHRULE I pre stmt post .
```