The Spirit of Ghost Code*

Jean-Christophe Filliâtre[1,2], Léon Gondelman[1], and Andrei Paskevich[1,2]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893

Abstract. In the context of deductive program verification, ghost code is part of the program that is added for the purpose of specification. Ghost code must not interfere with regular code, in the sense that it can be erased without observable difference in the program outcome. In particular, ghost data cannot participate in regular computations and ghost code cannot mutate regular data or diverge. The idea exists in the folklore since the early notion of auxiliary variables and is implemented in many state-of-the-art program verification tools. However, a rigorous definition and treatment of ghost code is surprisingly subtle and few formalizations exist.

In this article, we describe a simple ML-style programming language with mutable state and ghost code. Non-interference is ensured by a type system with effects, which allows, notably, the same data types and functions to be used in both regular and ghost code. We define the procedure of ghost code erasure and we prove its safety using bisimulation. A similar type system, with numerous extensions which we briefly discuss, is implemented in the program verification environment Why3.

1 Introduction

A common technique in deductive program verification consists in introducing data and computations, traditionally named *ghost code*, that only serve to facilitate specification. Ghost code can be safely erased from a program without affecting its final result. Consequently, a ghost expression cannot be used in a *regular* (non-ghost) computation, it cannot modify a regular mutable value, and it cannot raise exceptions that would escape into regular code. However, a ghost expression can use regular values and its result can be used in program annotations: preconditions, postconditions, loop invariants, assertions, etc. A classical use case for ghost code is to equip a data structure with ghost fields containing auxiliary data for specification purposes. Another example is ghost step counters to prove the time complexity of an algorithm.

When it comes to computing verification conditions, for instance using a weakest precondition calculus, there is no need to make a distinction between ghost and regular code. At this moment, ghost code is just a computation that supplies auxiliary values to use in specification and to simplify proofs. This computation,

* This work is partly supported by the Bware (ANR-12-INSE-0010, `http://bware.lri.fr/`) project of the French national research organization (ANR).

A. Biere and R. Bloem (Eds.): CAV 2014, LNCS 8559, pp. 1–16, 2014.

however, is not necessary for the program itself and thus should be removed when we compile the annotated source code. Therefore we need a way to ensure, by static analysis, that ghost code does not interfere with the rest of the program.

Despite that the concept of ghost code exists since the early days of deductive program verification, and is supported in most state-of-the-art tools [1–4], it is surprisingly subtle. In particular, a sound non-interference analysis must ensure that every ghost sub-expression terminates. Otherwise, one could supply such a sub-expression with an arbitrary postcondition and thus be able to prove anything about the program under consideration. Another non-obvious observation is that structural equality cannot be applied naively on data with ghost components. Indeed, two values could differ only in their ghost parts and consequently the comparison would yield a different result after the ghost code erasure.

There is a number of design choices that show up when conceiving a language with ghost code. First, how explicit should we be in our annotations? For example, should every ghost variable be annotated as such, or can we infer its status by looking at the values assigned to it? Second, how much can be shared between ghost and regular code? For instance, can a ghost value be passed to a function that does not specifically expect a ghost argument? Similarly, can we store a ghost value in a data structure that is not specifically designed to hold ghost data, *e.g.* an array or a tuple? Generally speaking, we should decide where ghost code can appear and what can appear in ghost code.

In this article, we show that, using a tailored type system with effects, we can design a language with ghost code that is both expressive and concise. As a proof of concept, we describe a simple ML-style programming language with mutable state, recursive functions, and ghost code. Notably, our type system allows the same data types and functions to be used in both regular and ghost code. We give a formal proof of the soundness of ghost code erasure, using a bisimulation argument. A type system based on the same concepts is implemented in the verification tool Why3 [4]. The language presented is this paper is deliberately simplified. The more exciting features, listed in Section 4 and implemented in Why3, only contribute to more complex effect tracking in the type system, which is mostly orthogonal to the problem of ghost code non-interference.

Outline. This paper is organized as follows. Section 2 introduces an ML-like language with ghost code. Section 3 defines the operation of ghost code erasure and proves its soundness. Section 4 describes the actual implementation in Why3. We conclude with related work in Section 5 and perspectives in Section 6. An extended version of this paper containing proofs is available at `http://hal.archives-ouvertes.fr/hal-00873187/`.

2 GhostML

We introduce GhostML, a mini ML-like language with ghost code. It features global references (that is, mutable variables), recursive functions, and integer and Boolean primitive types.

2.1 Syntax

The syntax of GhostML is given in Fig. 1. Terms are either values or compound expressions like application, conditional, reference access and modification. We assume a fixed finite set of global references. All the language constructions are standard ML, except for the keyword ghost which turns a term t into ghost code.

$t ::=$	TERMS	$v ::=$	VALUES
$\mid v$	*value*	$\mid c$	*constant*
$\mid t\ v$	*application*	$\mid x^\beta$	*variable*
$\mid \mathsf{let}\ x^\beta = t\ \mathsf{in}\ t$	*local binding*	$\mid \lambda x^\beta : \tau.\ t$	*anonymous function*
$\mid \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t$	*conditional*	$\mid \mathsf{rec}\ x^\beta : \tau^\beta \xRightarrow{\epsilon} \tau.\ \lambda x^\beta : \tau.\ t$	*recursive function*
$\mid r^\beta := v$	*assignment*		
$\mid\ !r^\beta$	*dereference*		
$\mid \mathsf{ghost}\ t$	*ghost code*	$c ::=$	CONSTANTS
		$\mid ()$	*unit*
$\tau ::=$	TYPES	$\mid ..., -1, 0, 1, ...$	*integers*
$\mid \kappa$	*primitive type*	$\mid \mathsf{true}, \mathsf{false}$	*Boolean*
$\mid \tau^\beta \xRightarrow{\epsilon} \tau$	*functional type*	$\mid +, \vee, =, ...$	*operators*
$\kappa ::=$	PRIMITIVE TYPES	$\beta \in \{\bot, \top\}$	GHOST STATUS
$\mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit}$	*primitive types*	$\epsilon \in \{\bot, \top\}$	EFFECT

Fig. 1. Syntax

Every variable is tagged with a ghost status β, which is $\top$ for ghost variables and $\bot$ for regular ones (here and below, "regular" stands for "non-ghost"). Similarly, references and formal function parameters carry their ghost status. Consider the following example:

$$\mathsf{let}\ upd^\top = \lambda x^\bot : \mathsf{int}.\ g^\top := x^\bot\ \mathsf{in}\ upd^\top\ !r^\bot$$

Here, function $upd^\top$ takes one regular parameter $x^\bot$ and assigns it to a ghost reference $g^\top$. Then $upd^\top$ is applied to the contents of a regular reference $r^\bot$.

Note that compound terms obey a variant of *A-normal* form [5]. That is, in application, conditional, and reference assignment, one of the sub-expressions must be a value. This does not reduce expressiveness, since a term such as $(t_1\ (t_2\ v))$ can be rewritten as $\mathsf{let}\ x^\beta = t_2\ v\ \mathsf{in}\ t_1\ x^\beta$, where β depends on the ghost status of the first formal parameter of t_1.

Types are either primitive data-types (int, bool, unit) or function types. A function type is an arrow $\tau_2^\beta \xRightarrow{\epsilon} \tau_1$ where β stands for the function argument's ghost status, and ϵ is the *latent* effect of the function. An effect ϵ is a Boolean value that indicates presence of regular side effects such as modification of a regular reference or possible non-termination.

MiniML Syntax. The syntax of traditional MiniML can be obtained by omitting all ghost indicators β (on references, variables, parameters, and types) and excluding the ghost construct. Equivalently, we could define MiniML as the subset of GhostML where all ghost indicators β are $\bot$ and where terms of the form ghost t do not appear.

2.2 Semantics

Fig. 2 gives a small-step operational semantics to GhostML which corresponds to a deterministic call-by-value reduction strategy. Each reduction step defines a relation between states. A state is a pair $t \mid \mu$ of a term t and a store μ. A store μ maps global references of t to constants. The regular part of a store μ, written $\mu_\bot$, is the restriction of μ to regular references. Rules indicate the store μ only when relevant.

A reduction step can take place directly at the top of a term t. Such a step is called a head reduction and is denoted $t \mid \mu \xrightarrow{\alpha} t' \mid \mu'$. Rule (E-Ghost) expresses that, from the point of view of operational semantics, there is no difference between regular and ghost code. Other head reduction rules are standard. For instance, rules (E-Op-λ) and (E-Op-δ) evaluate the application of a constant c_0 to constants $c_1 ... c_m$. Such an application is either partial ($1 \leq m < \text{arity}(c_0)$), and then turned into a function $\lambda x^\bot : \kappa.\ c_0\ c_1\ \ldots\ c_m\ x^\bot$, or total ($m = \text{arity}(c_0)$), and then some oracle function δ gives the result $\delta(c_0, c_1, \ldots, c_m)$. For instance, $\delta(\mathsf{not}, \mathsf{true}) = \mathsf{false}$, $\delta(+, 47, -5) = 42$, etc.

A reduction step can also be contextual, *i.e.* it takes place in some subexpression. Since our language is in A-normal form, there are only two contextual rules, (E-Context-App) and (E-Context-Let).

As usual, $\to^\star$ denotes the reflexive, transitive closure of $\to$. We say that a closed term t evaluates to v in a store μ if there is a μ' such that $t \mid \mu \to^\star v \mid \mu'$. Note that, since t is closed, v is not a variable. Finally, the divergence of a term t in a store μ is defined co-inductively as follows:

$$\frac{t \mid \mu \to^1 t' \mid \mu' \qquad t' \mid \mu' \to \infty}{t \mid \mu \to \infty}\text{(E-Div)}$$

MiniML Semantics. Since ghost statuses do not play any role in the semantics of GhostML, dropping them (or, equivalently, marking all β as $\bot$) and removing the rule (E-Ghost) results in a standard call-by-value small-step operational semantics for MiniML. For the sake of clarity, we use a subscript m when writing MiniML reduction steps: $t \mid \mu \to_m t' \mid \mu'$.

2.3 Type System

The purpose of the type system is to ensure that "well-typed terms do not go wrong". In our case, "do not go wrong" means not only that well-typed terms verify the classical type soundness property, but also that ghost code *does not*

$$\mathsf{ghost}\ t \xrightarrow{\alpha} t \qquad \text{(E-Ghost)}$$

$$\frac{1 \le m < \mathrm{arity}(c_0)}{c_0\ c_1\ \ldots\ c_m \xrightarrow{\alpha} \lambda x^{\perp} : \kappa.\ c_0\ c_1\ \ldots\ c_m\ x^{\perp}} \qquad \text{(E-Op-}\lambda\text{)}$$

$$\frac{m = \mathrm{arity}(c_0) \qquad \delta(c_0, c_1, \ldots, c_m) \text{ is defined}}{c_0\ c_1\ \ldots\ c_m \xrightarrow{\alpha} \delta(c_0, c_1, \ldots, c_m)} \qquad \text{(E-Op-}\delta\text{)}$$

$$(\lambda x^{\beta} : \tau.\ t)\ v \xrightarrow{\alpha} t[x^{\beta} \leftarrow v] \qquad \text{(E-App-}\lambda\text{)}$$

$$(\mathsf{rec}\ f^{\beta} : \tau^{\beta \xRightarrow{\epsilon}} \tau.\ \lambda x^{\beta} : \tau.\ t)\ v \xrightarrow{\alpha} t[x^{\beta} \leftarrow v, f^{\beta} \leftarrow \mathsf{rec}\ f^{\beta} : \tau^{\beta \xRightarrow{\epsilon}} \tau.\ \lambda x^{\beta} : \tau.\ t] \qquad \text{(E-App-Rec)}$$

$$\mathsf{let}\ x^{\beta} = v_1\ \mathsf{in}\ t_2 \xrightarrow{\alpha} t_2[x^{\beta} \leftarrow v_1] \qquad \text{(E-Let)}$$

$$\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \xrightarrow{\alpha} t_1 \qquad \text{(E-If-True)}$$

$$\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \xrightarrow{\alpha} t_2 \qquad \text{(E-If-False)}$$

$$!r^{\beta}\ |\ \mu \xrightarrow{\alpha} \mu(r^{\beta})\ |\ \mu \qquad \text{(E-Deref)}$$

$$r^{\beta} := c\ |\ \mu \xrightarrow{\alpha} ()\ |\ \mu[r^{\beta} \mapsto c] \qquad \text{(E-Assign)}$$

$$\frac{t\ |\ \mu \xrightarrow{\alpha} t'\ |\ \mu'}{t\ |\ \mu \to t'\ |\ \mu'} \qquad \text{(E-Head)}$$

$$\frac{t_1\ |\ \mu \to t_1'\ |\ \mu'}{(t_1\ v)\ |\ \mu \to (t_1'\ v)\ |\ \mu'} \qquad \text{(E-Context-App)}$$

$$\frac{t_2\ |\ \mu \to t_2'\ |\ \mu'}{\mathsf{let}\ x^{\beta} = t_2\ \mathsf{in}\ t_1\ |\ \mu \to \mathsf{let}\ x^{\beta} = t_2'\ \mathsf{in}\ t_1\ |\ \mu'} \qquad \text{(E-Context-Let)}$$

Fig. 2. Semantics

interfere with regular code. More precisely, non-interference means that ghost code never modifies regular references and that it always terminates. For that purpose, we introduce a type system with effects, where the typing judgment is

$$\Sigma, \Gamma \vdash t : \tau, \beta, \epsilon.$$

Here, τ is the type of term t. Boolean indicators β and ϵ indicate, respectively, the ghost status of t and its regular side effects. Γ is a typing environment that binds variables to types. Σ is a store typing that binds each reference r^{β} to the primitive type of the stored value. We restrict types of stored values to primitive types to avoid a possible non-termination via Landin's knot (that is,

$$\frac{\mathsf{Typeof}(c) = \tau}{\Sigma,\ \Gamma \vdash c : \tau,\ \bot,\ \bot} \quad \text{(T-Const)}$$

$$\frac{(x^{\beta} : \tau) \in \Gamma}{\Sigma,\ \Gamma \vdash x^{\beta} : \tau,\ \beta,\ \bot} \quad \text{(T-Var)}$$

$$\frac{\Sigma,\ \Gamma,\ x^{\beta} : \tau \vdash t : \tau_0,\ \beta_0,\ \epsilon}{\Sigma,\ \Gamma \vdash (\lambda x^{\beta} : \tau.\ t) : \tau \overset{\epsilon}{\Longrightarrow} \tau_0,\ \beta_0,\ \bot} \quad \text{(T-}\lambda\text{)}$$

$$\frac{\Sigma,\ \Gamma,\ f^{\bot} : \tau_2 \overset{\top}{\Longrightarrow} \tau_1 \vdash (\lambda x^{\beta} : \tau_2.\ t) : \tau_2 \overset{\epsilon}{\Longrightarrow} \tau_1,\ \bot,\ \bot}{\Sigma,\ \Gamma \vdash (\mathsf{rec}\ f^{\bot} : \tau_2 \overset{\top}{\Longrightarrow} \tau_1.\ \lambda x^{\beta} : \tau_2.\ t) : \tau_2 \overset{\top}{\Longrightarrow} \tau_1,\ \bot,\ \bot} \quad \text{(T-Rec)}$$

$$\frac{\Sigma,\ \Gamma \vdash v : \mathsf{bool},\ \beta_0,\ \bot \quad \Sigma,\ \Gamma \vdash t_1 : \tau,\ \beta_1,\ \epsilon_1 \quad \Sigma,\ \Gamma \vdash t_2 : \tau,\ \beta_2,\ \epsilon_2}{\Sigma,\ \Gamma \vdash (\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) : \tau,\ \beta_0 \vee \beta_1 \vee \beta_2,\ \epsilon_1 \vee \epsilon_2} \quad \text{(T-If)}$$

$$\frac{\Sigma,\ \Gamma,\ x^{\bot} : \tau_2 \vdash t_1 : \tau_1,\ \beta_1,\ \epsilon_1 \quad \Sigma,\ \Gamma \vdash t_2 : \tau_2,\ \beta_2,\ \epsilon_2}{\Sigma,\ \Gamma \vdash (\mathsf{let}\ x^{\bot} = t_2\ \mathsf{in}\ t_1) : \tau_1,\ \beta_1 \vee \beta_2,\ \epsilon_1 \vee \epsilon_2} \quad \text{(T-Let-Regular)}$$

$$\frac{\Sigma,\ \Gamma,\ x^{\top} : \tau_2 \vdash t_1 : \tau_1,\ \beta_1,\ \epsilon_1 \quad \Sigma,\ \Gamma \vdash t_2 : \tau_2,\ \beta_2,\ \bot}{\Sigma,\ \Gamma \vdash (\mathsf{let}\ x^{\top} = t_2\ \mathsf{in}\ t_1) : \tau_1,\ \beta_1,\ \epsilon_1} \quad \text{(T-Let-Ghost)}$$

$$\frac{\Sigma,\ \Gamma \vdash t : \tau_2 \overset{\epsilon_1}{\Longrightarrow} \tau_1,\ \beta_1,\ \epsilon_2 \quad \Sigma,\ \Gamma \vdash v : \tau_2,\ \beta_2,\ \bot}{\Sigma,\ \Gamma \vdash (t\ v) : \tau_1,\ \beta_1 \vee \beta_2,\ \epsilon_1 \vee \epsilon_2} \quad \text{(T-App-Regular)}$$

$$\frac{\Sigma,\ \Gamma \vdash t : \tau_2 \overset{\epsilon_1}{\Longrightarrow} \tau_1,\ \beta_1,\ \epsilon_2 \quad \Sigma,\ \Gamma \vdash v : \tau_2,\ \beta_2,\ \bot}{\Sigma,\ \Gamma \vdash (t\ v) : \tau_1,\ \beta_1,\ \epsilon_1 \vee \epsilon_2} \quad \text{(T-App-Ghost)}$$

$$\frac{(r^{\beta} : \kappa) \in \Sigma}{\Sigma,\ \Gamma \vdash\ !r^{\beta} : \kappa,\ \beta,\ \bot} \quad \text{(T-Deref)}$$

$$\frac{\Sigma,\ \Gamma \vdash v : \kappa,\ \beta',\ \bot \quad (r^{\beta} : \kappa) \in \Sigma \quad \beta \geq \beta'}{\Sigma,\ \Gamma \vdash (r^{\beta} := v) : \mathsf{unit},\ \beta,\ \neg\beta} \quad \text{(T-Assign)}$$

$$\frac{\Sigma,\ \Gamma \vdash t : \tau,\ \beta,\ \bot}{\Sigma,\ \Gamma \vdash (\mathsf{ghost}\ t) : \tau,\ \top,\ \bot} \quad \text{(T-Ghost)}$$

Fig. 3. Typing rules

recursion encoded using a mutable variable containing a function), which would be undetected in our type system.

Typing rules are given in Fig. 3. To account for non-interference, each rule whose conclusion is a judgement $\Sigma, \Gamma \vdash t : \tau, \beta, \epsilon$ is added the implicit extra side condition

$$(\beta = \top) \Rightarrow (\epsilon \vee \epsilon^+(\tau) = \bot) \tag{1}$$

where $\epsilon^+(\tau)$ is defined recursively on τ as follows:

$$\begin{aligned} \epsilon^+(\kappa) &\triangleq \bot \\ \epsilon^+(\tau_2 \xrightarrow{\epsilon}{}^{\beta} \tau_1) &\triangleq \epsilon \vee \epsilon^+(\tau_1) \end{aligned}$$

In other words, whenever t is ghost code, it must terminate and must not modify any regular reference. In particular, a ghost function whose body is possibly non-terminating or possibly modifies a regular reference is rejected by the type system.

Let us explain some rules in detail. The rule (T-Const) states that any constant c is regular code, (i.e. $\beta = \bot$) yet is pure and terminating (i.e. $\epsilon = \bot$). Moreover, we assume that if c is some constant operation, then its formal parameters are all regular. The type of each constant is given by some oracle function $\mathsf{Typeof}(c)$. For instance, $\mathsf{Typeof}(+) = \mathsf{int}^{\bot} \xrightarrow{\bot} \mathsf{int}^{\bot} \xrightarrow{\bot} \mathsf{int}$.

Recursive functions are typed as follows. For simplicity, we assume that whenever a recursive function is used, we may have non-termination. Therefore, we enforce the latent effect ϵ of any recursive function to be $\top$. Consequently, no recursive function can be used or even occur in ghost code. In practice, however, we do not have to assign a latent non-termination effect to recursive functions whose termination can be established by static analysis (e.g. by a formal proof).

The rule (T-If) shows how ghost code is propagated through conditional expressions: if at least one of the branches or the Boolean condition is ghost code, then the conditional itself becomes ghost. Note, however, that the typing side-condition (1) will reject conditionals where one part is ghost and another part has some effect, as in

$$\mathsf{if\ true\ then}\ r^{\bot} := 42\ \mathsf{else\ ghost}\ ().$$

The rule (T-Ghost) turns any term t into ghost code, with ghost status $\top$, whatever the ghost status of t is, provided that t is pure and terminating. Thus, terms such as $\mathsf{ghost}\ (r^{\bot} := 42)$ or $\mathsf{ghost}\ (\mathsf{fact}\ 3)$ are ill-typed, since their evaluation would interfere with the evaluation of regular code.

The side condition $(\beta \geq \beta')$ of the rule (T-Assign) ensures that regular references cannot be assigned ghost code. (Boolean values are ordered as usual, with $\top > \bot$.) Additionally, the rule conclusion ensures that, if the assigned reference is regular ($\beta = \bot$), then ϵ is $\top$; on the contrary, if the assigned reference is ghost ($\beta = \top$), then ϵ is $\bot$, since ghost reference assignments are not part of regular effects.

The most subtle rules are those for local bindings and application. Rule (T-Let-Ghost) states that, whatever the ghost status of a term t_2 is, as long as

t_2 is pure and terminating, we can bind a ghost variable $x^{\top}$ to t_2. Similarly, by rule (T-App-Ghost) a function that expects a ghost parameter can be applied to both ghost and regular values.

Rule (T-Let-Regular) is somewhat dual to (T-Let-Ghost): it allows us to bind a regular variable $x^{\bot}$ to a ghost term. The difference with the previous case is that, now, the ghost status of the let expression depends on the ghost status of t_2: if t_2 is ghost code, then the "contaminated" let expression becomes ghost itself. Consequently, if t_2 is ghost, then by the implicit side-condition, as $\epsilon_1 \vee \epsilon_2$ must be equal to $\bot$, both t_1 and t_2 must be pure and terminating. Similarly, rule (T-App-Regular) allows us to pass a ghost value to a function expecting a regular parameter, in which case the application itself becomes ghost. In other words, the goal of rules (T-Let-Regular) and (T-App-Regular) is to allow ghost code to use regular code. This was one of our motivations.

It is worth pointing out that there is no sub-typing in our system. That is, in rules for application, the formal parameter and the actual argument must have *exactly* the same type τ_2. In particular, all latent effects and ghost statuses in function types must be the same. For instance, a function expecting an argument of type $\mathsf{int}^{\bot} \xrightarrow{\epsilon} \mathsf{int}$ cannot be applied to an argument of type $\mathsf{int}^{\top} \xrightarrow{\epsilon} \mathsf{int}$.

Type System of MiniML. Similarly to operational semantics, if we drop all ghost statuses (or, equivalently, if we consider them marked as $\bot$) and get rid of typing rule (T-Ghost), we get a standard typing system with effects for MiniML with simple types. For clarity, we add a subscript m when we write typing judgments for MiniML terms: $\Sigma,\ \Gamma \vdash_m t : \tau,\ \epsilon$.

2.4 Type Soundness

The type system of GhostML enjoys the standard soundness property. Any well-typed program either diverges or evaluates to a value. This property is well established in the literature for ML with references [6, 7], and we can easily adapt the proof in our case. Due to lack of space, we only give the main statements.

As usual, we decompose type soundness into *preservation* and *progress* lemmas. First, we define well-typedness of a store with respect to a store typing.

Definition 1. *A store μ is well-typed with respect to a store typing Σ, written $\Sigma \vdash \mu$, if $dom(\mu) \subseteq dom(\Sigma)$ and $\mu(r^{\beta})$ has type $\Sigma(r^{\beta})$ for every $r^{\beta} \in dom(\mu)$.*

With this definition, the *preservation* lemma is stated as follows:

Lemma 1 (Preservation). *If $\Sigma, \Gamma \vdash t : \tau, \beta, \epsilon$ and $\Sigma \vdash \mu$, then $t \mid \mu \rightarrow t' \mid \mu'$ implies that $\Sigma, \Gamma \vdash t' : \tau, \beta', \epsilon'$ and $\Sigma \vdash \mu'$, where $\beta \geq \beta'$ and $\epsilon \geq \epsilon'$.*

The only difference with respect to the standard statement is that ghost statuses and effect indicators can decrease during evaluation.

Lemma 2 (Progress). *If $\Sigma, \emptyset \vdash t : \tau, \beta, \epsilon$, then either t is a value or, for any store μ such that $\Sigma \vdash \mu$, there exists a reduction step $t \mid \mu \rightarrow t' \mid \mu'$.*

Additionally, we have the following results for effect-less programs.

Lemma 3 (Store Preservation). *If* $\Sigma, \emptyset \vdash t : \tau, \beta, \bot$ *and* $\Sigma \vdash \mu$*, then* $t \mid \mu \rightarrow t' \mid \mu'$ *implies* $\mu_\bot = \mu'_\bot$.

Lemma 4 (Program Termination). *If* $\Sigma, \emptyset \vdash t : \tau, \beta, \bot$ *and* $\Sigma \vdash \mu$*, then evaluation of* t *in store* μ *terminates, that is, there is a value* v *and a store* μ' *such that* $t \mid \mu \rightarrow^\star v \mid \mu'$.

A consequence of the previous lemmas and the side condition (1) is that ghost code does not modify the regular store and is terminating.

3 From GhostML to MiniML

This section describes an erasure operation that turns a GhostML term into a MiniML term. The goal is to show that ghost code can be erased from a regular program without observable difference in the program outcome.

The erasure is written either $\mathcal{E}_\beta(.)$, when parameterized by some ghost status β, and simply $\mathcal{E}(.)$ otherwise. First, we define erasure on types and terms. The main idea is to preserve the structure of regular terms and types, and to replace any ghost code by a value of type unit.

Definition 2 (τ-erasure). *Let* τ *be some GhostML type. The erasure* $\mathcal{E}_\beta(\tau)$ *of type* τ *with respect to* β *is defined by induction on the structure of* τ *as follows:*

$$\begin{array}{ll} \mathcal{E}_\top(\tau) & \triangleq \mathsf{unit} \\ \mathcal{E}_\bot(\tau_2 \xrightarrow{\epsilon}{}^{\beta_2} \tau_1) & \triangleq \mathcal{E}_{\beta_2}(\tau_2) \xrightarrow{\epsilon} \mathcal{E}_\bot(\tau_1) \\ \mathcal{E}_\bot(\kappa) & \triangleq \kappa \end{array}$$

In other words, the structure of regular types is preserved and all ghost types are turned into type unit. Now we can define erasure on terms.

Definition 3 (t-Erasure). *Let* t *be such that* $\Sigma, \Gamma \vdash t : \tau, \beta, \epsilon$ *holds. The erasure* $\mathcal{E}_\beta(t)$ *is defined by induction on the structure of* t *as follows:*

$$\begin{array}{ll} \mathcal{E}_\top(t) & \triangleq () \\ \mathcal{E}_\bot(c) & \triangleq c \\ \mathcal{E}_\bot(x^\bot) & \triangleq x \\ \mathcal{E}_\bot(\lambda x^\beta : \tau.\, t) & \triangleq \lambda x : \mathcal{E}_\beta(\tau).\, \mathcal{E}_\bot(t) \\ \mathcal{E}_\bot(\mathsf{rec}\ f^\bot : \tau_2 \xrightarrow{\top}{}^{\beta_2} \tau_1.\, t) & \triangleq \mathsf{rec}\ f : \mathcal{E}_\bot(\tau_2 \xrightarrow{\top}{}^{\beta_2} \tau_1).\, \mathcal{E}_\bot(t) \\ \mathcal{E}_\bot(r^\bot := v) & \triangleq r := \mathcal{E}_\bot(v) \\ \mathcal{E}_\bot(!r^\bot) & \triangleq !r \\ \mathcal{E}_\bot(\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) & \triangleq \mathsf{if}\ \mathcal{E}_\bot(v)\ \mathsf{then}\ \mathcal{E}_\bot(t_1)\ \mathsf{else}\ \mathcal{E}_\bot(t_2) \\ \mathcal{E}_\bot(t\ v) & \triangleq \mathcal{E}_\bot(t)\ \mathcal{E}_{\beta'}(v) \quad \textit{where } t \textit{ has type } \tau_2 \xrightarrow{\epsilon_1}{}^{\beta'} \tau_1 \\ \mathcal{E}_\bot(\mathsf{let}\ x^{\beta'} = t_2\ \mathsf{in}\ t_1) & \triangleq \mathsf{let}\ x = \mathcal{E}_{\beta'}(t_2)\ \mathsf{in}\ \mathcal{E}_\bot(t_1) \end{array}$$

Note that ghosts variables and ghost references do not occur anymore in $\mathcal{E}_\perp(t)$. Note also that a regular function (recursive or not) with a ghost parameter remains a function, but with an argument of type unit. Similarly, a let expression that binds a ghost variable inside a regular code remains a let, but now binds a variable to (). More generally, $\mathcal{E}_\perp(t)$ is a value if and only if t is a value.

Leaving unit values and arguments in the outcome of erasure may seem unnecessary. However, because of latent effects, full erasure of ghost code is not possible. Consider for instance the function

$$\lambda x^{\perp} : \mathsf{int}.\ \lambda y^{\top} : \mathsf{int}.\ r^{\perp} := x$$

where r is a regular reference. Then a partial application of this function to a single argument should not trigger the modification of r. Our solution is to keep a second argument y of type unit.

3.1 Well-Typedness Preservation

We prove that erasure preserves well-typedness of terms. To do so, we first define the erasure of a typing context and of a store typing by a straightforward induction on their size:

Definition 4 (Γ-erasure and Σ-erasure).

$$\begin{array}{lll} \mathcal{E}(\emptyset) & \triangleq \emptyset & \mathcal{E}(\emptyset) \triangleq \emptyset \\ \mathcal{E}(\Gamma,\ x^{\top} : \tau) & \triangleq \mathcal{E}(\Gamma),\ x : \mathsf{unit} & \mathcal{E}(\Sigma,\ r^{\top} : \kappa) \triangleq \mathcal{E}(\Sigma) \\ \mathcal{E}(\Gamma,\ x^{\perp} : \tau) & \triangleq \mathcal{E}(\Gamma),\ x : \mathcal{E}_\perp(\tau) & \mathcal{E}(\Sigma,\ r^{\perp} : \kappa) \triangleq \mathcal{E}(\Sigma),\ r : \kappa \end{array}$$

With these definitions, we prove well-typedness preservation under erasure:

Theorem 1 (Well-typedness Preservation). *If $\Sigma, \Gamma \vdash t : \tau, \perp, \epsilon$ holds, then $\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma) \vdash_m \mathcal{E}_\perp(t) : \mathcal{E}_\perp(\tau),\ \epsilon$ holds.*

3.2 Correctness of Erasure

Finally, we prove correctness of erasure, that is, evaluation is preserved by erasure. To turn this into a formal statement, we first define the erasure of a store μ by a straightforward induction on the store size:

Definition 5 (μ-erasure).

$$\begin{array}{ll} \mathcal{E}(\emptyset) & \triangleq \emptyset \\ \mathcal{E}(\mu \uplus \{r^{\top} \mapsto c\}) & \triangleq \mathcal{E}(\mu) \\ \mathcal{E}(\mu \uplus \{r^{\perp} \mapsto c\}) & \triangleq \mathcal{E}(\mu) \uplus \{r \mapsto c\} \end{array}$$

Notice that $\mathcal{E}(\mu)$ removes ghost annotations, and thus is not the same that $\mu_\perp$. The correctness of erasure means that, for any evaluation $t \mid \mu \rightarrow^{\star} v \mid \mu'$ in GhostML, we have $\mathcal{E}_\perp(t) \mid \mathcal{E}(\mu) \rightarrow_m^{\star} \mathcal{E}_\perp(v) \mid \mathcal{E}(\mu')$ in MiniML and that, for any

diverging evaluation $t \mid \mu \rightarrow \infty$ in GhostML, we have $\mathcal{E}_\perp(t) \mid \mathcal{E}(\mu) \rightarrow_m \infty$ in MiniML. We prove these two statements using a bisimulation argument. First, we need the substitution lemma below, which states that substitution and erasure commute.

Lemma 5 (Substitution Under Erasure). *Let t be a GhostML term and v a GhostML value such that $\Sigma, \Gamma, x^\beta : \tau \vdash t : \tau_0, \perp, \epsilon$ and $\Sigma, \Gamma \vdash v : \tau, \beta', \perp$, with $\beta \geq \beta'$, hold. Then the following holds:*

$$\mathcal{E}_\perp(t)[x \leftarrow \mathcal{E}_\beta(v)] = \mathcal{E}_\perp(t[x^\beta \leftarrow v]).$$

Note that if $\Sigma \vdash \mu$ then $\mathcal{E}(\Sigma) \vdash_m \mathcal{E}(\mu)$. To prove erasure correctness for terminating programs, we use the following forward simulation argument:

Lemma 6 (Forward Simulation of GhostML). *If $\Sigma, \emptyset \vdash t : \tau, \perp, \epsilon$ and, for some store μ such that $\Sigma \vdash \mu$, we have $t \mid \mu \rightarrow t' \mid \mu'$, then the following holds in MiniML: $\mathcal{E}_\perp(t) \mid \mathcal{E}(\mu) \rightarrow_m^{0|1} \mathcal{E}_\perp(t') \mid \mathcal{E}(\mu')$.*

We are now able to prove the first part of the main theorem:

Theorem 2 (Terminating Evaluation Preservation). *If typing judgment $\Sigma, \emptyset \vdash t : \tau, \perp, \epsilon$ holds and $t \mid \mu \rightarrow^\star v \mid \mu'$, for some value v and some store μ such that $\Sigma \vdash \mu$, then $\mathcal{E}_\perp(t) \mid \mathcal{E}(\mu) \rightarrow_m^\star \mathcal{E}_\perp(v) \mid \mathcal{E}(\mu')$.*

To prove the second part of the erasure correctness (non-termination preservation), we use the following simulation argument.

Lemma 7 (Forward Simulation of MiniML). *If $\Sigma, \emptyset \vdash t : \tau, \perp, \epsilon$ holds, then, for any store μ such that $\Sigma \vdash \mu$, if $\mathcal{E}_\perp(t) \mid \mathcal{E}(\mu) \rightarrow_m q \mid \nu$ for some term q and some store ν, then $t \mid \mu \rightarrow^{\geq 1} t' \mid \mu'$ where $\mathcal{E}_\perp(t') = q$ and $\mathcal{E}(\mu') = \nu$.*

Finally, we establish non-termination preservation:

Theorem 3 (Non-termination Preservation). *If $\Sigma, \emptyset \vdash t : \tau, \perp, \epsilon$ holds and $t \mid \mu \rightarrow \infty$, for some store μ such that $\Sigma \vdash \mu$, then $\mathcal{E}_\perp(t)$ also diverges, that is, $\mathcal{E}_\perp(t) \mid \mathcal{E}(\mu) \rightarrow_m \infty$.*

4 Implementation

Our method to handle ghost code is implemented in the verification tool Why3[1]. With respect to GhostML, the language and the type system of Why3 have the following extensions:

Type Polymorphism. The type system of Why3 is first-order and features ML-style type polymorphism. Our approach to associate ghost status with variables and expressions, and not with types, makes this extension straightforward.

[1] Why3 is freely available from `http://why3.lri.fr/`

Local References. Another obvious extension of GhostML is the support of non-global references. As long as such a reference cannot be an alias for another one, the type system of GhostML requires practically no changes. In a system where aliases are admitted, the type system and, possibly, the verification condition generator must be adapted to detect modifications made by a ghost code in locations accessible from regular code. In Why3, aliases are tracked statically, and thus non-interference is ensured purely by type checking.

Data Structures with Ghost Fields. Why3 supports algebraic data types (in particular, records), whose fields may be regular or ghost. Pattern matching on such structures requires certain precautions. Any variable bound in the ghost part of a pattern must be ghost. Moreover, pattern matching over a ghost expression that has at least two branches must make the whole expression ghost, whatever the right-hand sides of the branches are, just as in the case of a conditional over a ghost Boolean expression.

That said, ghost code can use the same data types as regular code. A ghost variable may be a record with regular, mutable fields, which can be accessed and modified in ghost code. Similarly, Why3 has a unique type of arrays and admits both regular and ghost arrays.

Exceptions. Adding exceptions is rather straightforward, since in Why3 exceptions are introduced only at the top level. Indeed, it suffices to add a new effect indicator, that is the set of exceptions possibly raised by a program expression. We can use the same exceptions in ghost and regular code, provided that the ghost status of an expression that raises an exception is propagated upwards until the exception is caught.

Provable Termination. For the sake of simplicity, GhostML forbids the use of recursive functions in ghost code. In Why3, the use of recursive functions and loops in ghost code is allowed. The system requires that such constructs are supplied with a "variant" clause, so that verification conditions for termination are generated.

Example. Let us illustrate the use of ghost code in Why3 on a simple example. Fig. 4 contains an implementation of a mutable queue data type, in Baker's style. A queue is a pair of two immutable singly-linked lists, which serve to amortize push and pop operations. Our implementation additionally stores the pure logical view of the queue as a list, in the third, ghost field of the record. Notice that we use the same `list` type both for regular and ghost data.

We illustrate propagation in function `push` (lines 27–30), where a local variable `v` is used to hold some intermediate value, to be stored later in the ghost field of the structure. Despite the fact that variable `v` is not declared ghost, and the fact that function `append` is a regular function, Why3 infers that `v` is ghost. Indeed, the ghost value `q.view` contaminates the result of `append`. It would therefore generate an error if we tried to store `v` in a non-ghost field of an existing regular structure. Since the expression `append q.view (Cons x Nil)`

```
module Queue

  type elt

  type list = Nil | Cons elt list

  let rec append (l1 l2: list) : list
    variant { l1 }
  = match l1 with
    | Nil → l2
    | Cons x r1 → Cons x (append r1 l2)
    end

  let rec rev_append (l1 l2: list) : list
    variant { l1 }
  = match l1 with
    | Nil → l2
    | Cons x r1 → rev_append r1 (Cons x l2)
    end

  type queue = {
          mutable front: list;
          mutable rear:  list;
    ghost mutable view:  list;
  }

  let push (x: elt) (q: queue) : unit
  = q.rear ← Cons x q.rear;
    let v = append q.view (Cons x Nil) in
    q.view ← v

  exception Empty

  let pop (q: queue): elt
    raises { Empty }
  = match q.front with
    | Cons x f →
        q.front ← f;
        q.view  ← append f (rev_append q.rear Nil);
        x
    | Nil →
        match rev_append q.rear Nil with
        | Nil →
            raise Empty
        | Cons x f →
            q.front ← f;
            q.rear ← Nil;
            q.view ← f;
            x
        end
    end
end
```

Fig. 4. Queue implementation in Why3

is ghost, it must not diverge. Thus Why3 requires function `append` to be terminating. This is ensured by the `variant` clause on line 8. In function `pop` (lines 34–52), the regular function `rev_append` is used both in regular code (line 42) and ghost code (line 39).

The online gallery of verified Why3 programs contains several other examples of use of ghost code[2], in particular, ghost function parameters and ghost functions to supply automatic induction proofs (also known as lemma functions).

5 Related Work

The idea to use ghost code in a program to ease specification exists since the early days (late sixties) of deductive program verification, when so-called auxiliary variables became a useful technique in the context of concurrent programming. According to Jones [8] and Reynolds [9], the notion of auxiliary variable was first introduced by Lucas in 1968 [10]. Since then, numerous authors have adapted this technique in various domains.

It is worth pointing out that some authors, in particular Kleymann [11] and Reynolds [9], make a clear distinction between non-operational variables used in program annotations and specification-purpose variables that can appear in the program itself. The latter notion has gradually evolved into the wider idea that ghost code can be arbitrary code, provided it does not interfere with regular code. For example, Zhang *et al.* [12] discuss the use of auxiliary code in the context of concurrent program verification. They present a simple WHILE language with parallelism and auxiliary code, and prove that the latter does not interfere with the rest of the program. In their case, non-interference is ensured by the stratified syntax of the language. For instance, loops can contain auxiliary code, but auxiliary code cannot contain loops, which ensures termination. They also define auxiliary code erasure and prove that a program with ghost code has no less behaviors than its regular part. Schmaltz [13] proposes a rigorous description of ghost code for a large fragment of C with parallelism, in the context of the VCC verification tool [2]. VCC includes ghost data types, ghost fields in regular structures, ghost parameters in regular functions, and ghost variables. In particular, ghost code is used to manipulate ownership information. A notable difference w.r.t. our work is that VCC does not perform any kind of inference of ghost code. Another difference is that VCC *assumes* that ghost code terminates, and the presence of constructions such as `ghost(goto l)` makes it difficult to reason about ghost code termination.

Another example of a modern deductive verification tool implementing ghost code is the program verifier Dafny [1]. In Dafny, "the concept of ghost versus non-ghost declarations is an integral part of the Dafny language: each function, method, variable, and parameter can be declared as either ghost or non-ghost." [14]. In addition, a class can contain both ghost fields and regular fields. Dafny ensures termination of ghost code. Ghost code can update ghost fields, but is not allowed to allocate memory or update non-ghost fields. Consequently, ghost

[2] `http://toccata.lri.fr/gallery/ghost.en.html`

code cannot obtain full reuse of libraries that allocate and mutate classes or arrays. However, on the fragment of Dafny's language corresponding to GhostML, Dafny provides a semantics of ghost code similar to what is presented here.

The property of non-interference of ghost code is a special case of information flow non-interference [15]. Indeed, one can see ghost code as high-security information and regular code as low-security information, and non-interference precisely means that high-security information does not leak into low-security computations. Information flow properties can be checked using a type system [16] and proofs in that domain typically involve a bisimulation technique (though not necessarily through an erasure operation). Notice that applying an information flow type system to solve our problem is not straightforward, since termination of ghost code is a crucial requirement. For instance, the type system described by Simonet and Pottier [17] simply assumes termination of secret code. To the best of our knowledge, this connection between information flow and ghost code has not been made before, and mainstream deductive verification tools employ syntactical criteria of non-interference instead of type-based ones. In this paper, we develop such a type-based approach, specifically tailored for program verification.

6 Conclusion and Perspectives

In this paper, we described an ML-like language with ghost code. Non-interference between ghost code and regular code is ensured using a type system with effects. We formally proved the soundness of this type system, that is, ghost code can be erased without observable difference. Our type system results in a highly expressive language, where the same data types and functions can be reused in both ghost and regular code.

We see two primary directions of future work on ghost code and Why3. First, ghost code, especially ghost fields, plays an important role in program refinement. Indeed, ghost fields that give sufficient information to specify a data type are naturally shared between the interface and the implementation of this data type. In this way, the glue invariant becomes nothing more than the data type invariant linking regular and ghost fields together. Our intention is to design and implement in Why3 a module system with refinement that makes extensive use of ghost code and data. Second, since ghost code does not have to be executable, it should be possible to use in ghost code various constructs which, up to now, may only appear in specifications, such as quantifiers, inductive predicates, non-deterministic choice, or infinitely parallel computations (cf. the aggregate `forall` statement in Dafny).

Acknowledgments. We are grateful to Sylvain Conchon, Rustan Leino, and François Pottier for comments and discussions regarding earlier versions of this paper.

References

1. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
2. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
3. Jacobs, B., Piessens, F.: The VeriFast program verifier. CW Reports CW520, Department of Computer Science, K.U. Leuven (August 2008)
4. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013)
5. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. SIGPLAN Not. 28(6), 237–247 (1993)
6. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115, 38–94 (1992)
7. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
8. Jones, C.B., Roscoe, A., Wood, K.R.: Reflections on the Work of C.A.R. Hoare, 1st edn. Springer Publishing Company, Incorporated (2010)
9. Reynolds, J.C.: The craft of programming. Prentice Hall International series in computer science. Prentice Hall (1981)
10. Lucas, P.: Two constructive realizations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory, Vienna (June 1968)
11. Kleymann, T.: Hoare logic and auxiliary variables. Formal Asp. Comput. 11(5), 541–566 (1999)
12. Zhang, Z., Feng, X., Fu, M., Shao, Z., Li, Y.: A structural approach to prophecy variables. In: Agrawal, M., Cooper, S.B., Li, A. (eds.) TAMC 2012. LNCS, vol. 7287, pp. 61–71. Springer, Heidelberg (2012)
13. Schmaltz, S.: Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C. PhD thesis, Saarland University, Saarbrcken (2013)
14. Leino, K.R.M., Moskal, M.: Co-induction simply. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 382–398. Springer, Heidelberg (2014)
15. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM 20(2), 504–513 (1977)
16. Pottier, F., Conchon, S.: Information flow inference for free. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), Montréal, Canada, pp. 46–57 (September 2000)
17. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Transactions on Programming Languages and Systems 25(1), 117–158 (2003) ACM