The Spirit of Ghost Code

Jean-Christophe Filliâtre, Léon Gondelman, Andrei Paskevich

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 any 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 compute verification conditions, for instance using a weakest preconditions 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).

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].

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.

2 GhostML

We introduce GhostML, a mini ML-like language with ghost code. It features global references, 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, etc. Each variable is tagged with a

t ::=		TERMS
	v	*value*
|	$(t\ v)$	*application*
|	$\mathsf{let}\ x^{\beta}\ =\ t\ \mathsf{in}\ t$	*local binding*
|	$\mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t$	*conditional*
|	$r^{\beta} := v$	*assignment*
|	$!r^{\beta}$	*dereference*
|	$\mathsf{ghost}\ t$	*ghost code*
τ ::=		TYPES
|	κ	*built-in type*
|	$\mathsf{ref}\ \kappa$	*reference type*
|	$\tau^{\beta} \overset{(\theta,\rho)}{\Longrightarrow} \tau$	*functional type*
κ ::=		BUILT-IN TYPES
|	$\mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit}$	*built-in types*

v ::=		VALUES
|	c	*constant*
|	x^{β}	*variable*
|	$\lambda x^{\beta} : \tau.\, t$	*anonymous function*
|	$\mathsf{rec}\ x^{\beta}\colon \tau^{\beta} \overset{(\theta,\rho)}{\Longrightarrow} \tau.\, t$	*recursive function*
c ::=		CONSTANTS
|	$()$	*unit*
|	$\ldots, -1, 0, 1, \ldots$	*integers*
|	$\mathsf{true}, \mathsf{false}$	*Boolean*
|	$+, \vee, =, \ldots$	*operators*
		EFFECTS
θ	$\in \{\bot, \top\}$	*assignement*
ρ	$\in \{\bot, \top\}$	*recursive call*
β	$\in \{\bot, \top\}$	GHOST STATUS

Fig. 1. Syntax.

ghost status β, which is $\top$ for ghost variables and $\bot$ for regular ones. Terms can assign or access global references. We assume that a finite set of global references with distinct names is given. Similar to local variables, references are tagged with a ghost status. Note that each *occurrence* of a reference or a variable is explicitly annotated with its ghost status. Formal parameters of functions are also introduced with a ghost status. For instance, the following term introduces a local function $upd^{\top}$, that updates a ghost reference $g^{\top}$ with its parameter, and applies it to the contents of a regular reference $r^{\bot}$:

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

The keyword ghost turns a term t into ghost code. For instance, ghost 42 is a ghost term. This way, the same constant 42 can be used in both regular and ghost code.

Note that compound terms obey to 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 $\mathsf{let}\ x^{\bot}\ =\ (t_2\ v)\ \mathsf{in}\ (t_1\ x^{\bot})$.

Types are either primitive data-types (int, bool, unit), reference types, or function types. A function type is an arrow $\tau_2^{\beta} \overset{(\theta,\rho)}{\Longrightarrow} \tau_1$ where β stands for the function argument's ghost status, and (θ, ρ) is the *latent* effect of the function. θ stands for the presence of non-ghost reference assignment, and ρ stands for the presence of 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 $\mu_{\mid\bot}$, is the restriction of μ to regular references. Rules indicate the store μ only when relevant.

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

$$\frac{1 \le m < \mathsf{arity}(c_0)}{c_0\ c_1 \ldots c_m \xrightarrow{\epsilon}_g \lambda x^{\bot}{:}\kappa.\, c_0\ c_1 \ldots c_m\ x} \qquad \text{(E-Op-}\lambda\text{)}$$

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

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

$$(\mathsf{rec}\ f^{\beta}\colon \tau^{\beta} \overset{(\theta,\rho)}{\Longrightarrow} \tau.\, \lambda x^{\beta} : \tau.\, t)\ v \xrightarrow{\epsilon}_g t[x^{\beta} \leftarrow v, f^{\beta} \leftarrow \mathsf{rec}\ f^{\beta}\colon \tau^{\beta} \overset{(\theta,\rho)}{\Longrightarrow} \tau.\, \lambda x^{\beta} : \tau.\, t] \qquad \text{(E-App-Rec)}$$

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

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

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

$$!r^{\beta} \mid \mu \xrightarrow{\epsilon}_g \mu(r^{\beta}) \mid \mu \qquad \text{(E-Deref)}$$

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

$$\frac{t \mid \mu \xrightarrow{\epsilon}_g t' \mid \mu'}{t \mid \mu \to_g t' \mid \mu'} \qquad \text{(E-Head)}$$

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

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

Fig. 2. Semantics.

A reduction step can take place directly *on the top* of a term t. Such step is called a head reduction and is denoted $t \mid \mu \xrightarrow{\epsilon}_g t \mid \mu'$.

The rule (E-Ghost) expresses that, from the semantics point of view, there is no difference between regular and ghost code. Other head reduction rules are standard. For instance, rules (E-Const-λ) and (E-Const-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 $\lambda x^{\bot} : \kappa.\, c_1 ... c_m$, or total ($m = \text{arity}(c_0)$), and then some oracle function δ gives the result $\delta(c_0, c_1, ... c_m)$. For instance, $\delta(\mathsf{not}, \mathsf{true}) = \mathsf{false}$, $\delta(+, 47, -5) = 42$, etc. Note also that the rule (E-App-Rec) only reduces recursive functions whose body is a function. Said otherwise, only functions can be defined recursively.

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, $\rightarrow_g^\star$ denotes the reflexive, transitive closure of $\rightarrow_g$. We say that a closed term t evaluates to v in a store μ if there exists μ' such that $t \mid \mu \rightarrow_g^\star v \mid \mu'$. 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:

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

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 SOS semantics for MiniML.

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 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

$$\Sigma,\ \Gamma\ \vdash_{\boldsymbol{g}}\ t\ :\ \tau,\ \beta,\ (\theta,\ \rho).$$

Here, Σ is a store typing, that binds references to types, and Γ is a typing environment, that binds variables to types. Boolean indicators β, θ, and ρ indicate, respectively, the ghost status of t, whether some *regular* reference is possibly assigned by t, and whether t is possibly non-terminating.

Typing rules are given in Fig. 3. In each rule, whose conclusion is a judgment $\Sigma,\ \Gamma\ \vdash_{\boldsymbol{g}}\ t\ :\ \tau,\ \beta,\ (\theta,\ \rho)$, we add the implicit extra side-condition

$$(\beta = \top) \Rightarrow (\theta = \bot \wedge \rho = \bot) \tag{1}$$

$$\frac{\mathsf{Typeof}(c) = \tau}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} c : \tau,\ \bot,\ (\bot,\ \bot)} \quad (\mathrm{T}_g\text{-Const})$$

$$\frac{x^{\beta} : \tau \in \Gamma}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} x^{\beta} : \tau,\ \beta,\ (\bot,\ \bot)} \quad (\mathrm{T}_g\text{-Var})$$

$$\frac{\Sigma,\ \Gamma,\ x^{\beta} : \tau \vdash_{\boldsymbol{g}} t : \tau_0,\ \beta_0,\ (\theta,\ \rho)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \lambda x^{\beta}{:}\tau.\, t : \tau^{\beta} \overset{(\theta,\rho)}{\Longrightarrow} \tau_0,\ \beta_0,\ (\bot,\ \bot)} \quad (\mathrm{T}_g\text{-}\lambda)$$

$$\frac{\Sigma,\ \Gamma,\ f^{\bot} : \tau \vdash_{\boldsymbol{g}} \lambda x^{\beta}{:}\tau_2.\, t : \tau_2^{\beta} \overset{(\theta,\rho)}{\Longrightarrow} \tau_1,\ \bot,\ (\bot,\ \bot) \quad \text{where } \tau \triangleq \tau_2^{\beta} \overset{(\theta,\top)}{\Longrightarrow} \tau_1}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \mathsf{rec}\ f^{\bot}{:}\ \tau.\, \lambda x^{\beta}{:}\tau_2.\, t : \tau,\ \bot,\ (\bot,\ \bot)} \quad (\mathrm{T}_g\text{-Rec})$$

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} v : \mathsf{bool},\ \beta_0,\ (\bot,\ \bot) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t_1 : \tau,\ \beta_1,\ (\theta_1,\ \rho_1) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t_2 : \tau,\ \beta_2,\ (\theta_2,\ \rho_2)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \tau,\ \beta_0 \vee \beta_1 \vee \beta_2,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)} \quad (\mathrm{T}_g\text{-If})$$

$$\frac{\Sigma,\ \Gamma,\ x^{\bot} : \tau_2 \vdash_{\boldsymbol{g}} t_1 : \tau_1,\ \beta_1,\ (\theta_1,\ \rho_1) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t_2 : \tau_2,\ \beta_2,\ (\theta_2,\ \rho_2)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \mathsf{let}\ x^{\bot} = t_2\ \mathsf{in}\ t_1 : \tau_1,\ \beta_1 \vee \beta_2,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)} \quad (\mathrm{T}_g\text{-Let-Regular})$$

$$\frac{\Sigma,\ \Gamma,\ x^{\top} : \tau_2 \vdash_{\boldsymbol{g}} t_1 : \tau_1,\ \beta_1,\ (\theta_1,\ \rho_1) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t_2 : \tau_2,\ \beta_2,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \mathsf{let}\ x^{\top} = t_2\ \mathsf{in}\ t_1 : \tau_1,\ \beta_1,\ (\theta_1,\ \rho_1)} \quad (\mathrm{T}_g\text{-Let-Ghost})$$

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t : \tau_2^{\bot} \overset{(\theta_1,\rho_1)}{\Longrightarrow} \tau_1,\ \beta_1,\ (\theta_2,\ \rho_2) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} v : \tau_2,\ \beta_2,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} (t\ v) : \tau_1,\ \beta_1 \vee \beta_2,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)} \quad (\mathrm{T}_g\text{-App-Regular})$$

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t : \tau_2^{\top} \overset{(\theta_1,\rho_1)}{\Longrightarrow} \tau_1,\ \beta_1,\ (\theta_2,\ \rho_2) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} v : \tau_2,\ \beta_2,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} (t\ v) : \tau_1,\ \beta_1,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)} \quad (\mathrm{T}_g\text{-App-Ghost})$$

$$\frac{r^{\beta} : \mathsf{ref}\ \kappa \in \Sigma}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} !r^{\beta} : \kappa,\ \beta,\ (\bot,\ \bot)} \quad (\mathrm{T}_g\text{-Deref})$$

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} v : \kappa,\ \beta',\ (\bot,\ \bot) \quad r^{\beta} : \mathsf{ref}\ \kappa \in \Sigma \quad \beta' \Rightarrow \beta}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} r^{\beta} := v : \mathsf{unit},\ \beta,\ (\neg\beta,\ \bot)} \quad (\mathrm{T}_g\text{-Assign})$$

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t : \tau,\ \beta,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \mathsf{ghost}\ t : \tau,\ \top,\ (\bot,\ \bot)} \quad (\mathrm{T}_g\text{-Ghost})$$

Fig. 3. Typing rules.

to account for non-interference. Said otherwise, whenever t is ghost code, then it must terminate and must not assign any regular reference.

Let us explain some rules in details. The rule (T_g-Const) states that any constant c is regular code ($\beta = \bot$) yet is pure and terminating ($\theta = \bot, \rho = \bot$). The type of each constant is given by some oracle function $\mathsf{Typeof}(c)$. For instance, $\mathsf{Typeof}(+) = \mathsf{int}^{\bot} \xrightarrow{(\bot,\bot)} \mathsf{int}^{\bot} \xrightarrow{(\bot,\bot)} \mathsf{int}$.

Recursive functions are typed as follows. For simplicity, we assume that whenever a recursive function is used, we may have non-termination. (In Section 4, we will discuss more a elaborate solution.) To do that, we enforce the latent effect ρ of any recursive function to be $\top$. Consequently, we declare any recursive function as non-ghost. Note that the body of a recursive function could be terminating, as in $\mathsf{rec}\ f^{\bot}\colon \tau.\ \lambda x^{\bot} : \mathsf{int}.\ x^{\bot}$ yet we give it the type $\tau = \mathsf{int}^{\bot} \xrightarrow{(\bot,\top)} \mathsf{int}$ with a non-termination effect.

The rule (T_g-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_g-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 interference with the the evaluation of regular code.

The side-condition of the rule (T_g-Assign) checks that regular references cannot be assigned ghost code. Additionally, the rule conclusion ensures that, if the assigned reference is regular ($\beta = \bot$), then θ is $\top$; on the contrary, if the assigned reference is ghost ($\beta = \top$), then θ is $\bot$, since ghost reference assignments are not part of effects.

The most subtle rules are those for local bindings and application. Rule (T_g-let-ghost) states that, whatever the ghost status of a term t_2 is, as long as a term t_2 is pure and terminating, we can bind a ghost variable $x^{\top}$ to t_2. Similarly, Rule (T_g-app-ghost) allows a function with a ghost parameter to be passed a regular value.

Rule (T_g-let-regular) is somehow dual to (T_g-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 $\theta_1 \vee \theta_2$ and $\rho_1 \vee \rho_2$ must both be equal to $\bot$, both t_1 and t_2 must be pure and terminating. Similarly, rule (T_g-app-regular) allows us to pass a ghost value to a function with a regular parameter, in which case the application itself becomes ghost. In other words, the goal of rules (T_g-let-regular) and (T_g-let- 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 $\mathsf{int}^{\bot} \xrightarrow{(\theta,\ \rho)} \mathsf{int}$ cannot be applied to an argument of type $\mathsf{int}^{\top} \xrightarrow{(\theta,\ \rho)} \mathsf{int}$, though this would be safe.

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_g-GHOST), we get a standard typing system with effects for MiniML with simple types.

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. To do so, we first define well-typedness of a store with respect to a store typing.

Definition 1. *A store μ is said to be well typed with respect to a store typing Σ, written $\Sigma \vdash_g \mu$, 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 $\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} p : \tau,\ \beta,\ (\theta,\ \rho)$ and $\Sigma \vdash_g \mu$, then $p \mid \mu \rightarrow_g p' \mid \mu'$ implies $\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} p' : \tau,\ \beta',\ (\theta',\ \rho')$ and $\Sigma \vdash_g \mu'$, where $\beta \geq \beta'$, $\theta \geq \theta'$, and $\rho \geq \rho'$.*

The only difference with respect to the standard statement is that ghost statuses and effect indicators cannot increase during evaluation. (Boolean values are ordered as usual, with $\bot \leq \top$.)

Lemma 2 (Progress). *If $\Sigma,\ \emptyset \vdash_{\boldsymbol{g}} t : \tau,\ \beta,\ (\theta,\ \rho)$, then either t is a value or, for any store μ such that $\Sigma \vdash_g \mu$, there exists a reduction step $t \mid \mu \rightarrow_g t' \mid \mu'$.*

Additionally we have the following results.

Lemma 3 (Store Preservation). *If $\Sigma,\ \emptyset \vdash_{\boldsymbol{g}} t : \tau,\ \beta,\ (\bot,\ \rho)$ and, for some μ, we have $\Sigma \vdash_g \mu$ and $t \mid \mu \rightarrow_g t' \mid \mu'$, then $\mu_{|\bot} = \mu'_{|\bot}$.*

Lemma 4 (Program Termination). *If $\Sigma,\ \emptyset \vdash_{\boldsymbol{g}} t : \tau,\ \beta,\ (\theta,\ \bot)$, then evaluation of t terminates. That is, for any store μ such that $\Sigma \vdash_g \mu$, there is a value v and a store μ' such that $t \mid \mu \rightarrow_g^{\star} v \mid \mu'$.*

A consequence of the previous two lemmas is that ghost code does not modify the regular store and is terminating.

Corollary 1 (Ghost Code Behavior). *If* $\Sigma,\ \emptyset \vdash_{\boldsymbol{g}} t : \tau,\ \top,\ (\bot,\ \bot)$*, then, for any state* $t \mid \mu$ *such that* $\Sigma \vdash_g \mu$*, there exists a value* v *and a store* μ' *such that* $t \mid \mu \ \rightarrow_g^\star\ v \mid \mu'$ *and* $\mu_{\mid \bot} = \mu'_{\mid \bot}$*.*

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 the ghost code can be erased from a regular program without any observable difference in the program outcome.

The erasure is written either $\mathcal{E}_\beta(.)$, when parameterized by some ghost status β, or 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:*

$$\begin{array}{ll}
\mathcal{E}_\top(\tau) & = \mathsf{unit} \\
\mathcal{E}_\bot(\tau_2^{\beta_2} \overset{(\theta,\rho)}{\Longrightarrow} \tau_1) & = \mathcal{E}_{\beta_2}(\tau_2) \overset{(\theta,\rho)}{\Longrightarrow} \mathcal{E}_\bot(\tau_1) \\
\mathcal{E}_\bot(\kappa) & = \kappa \\
\mathcal{E}_\bot(\mathsf{ref}\ \kappa) & = \mathsf{ref}\ \kappa
\end{array}$$

Said otherwise, 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* $\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t : \tau,\ \beta,\ (\theta, \rho)$ *holds. The erasure* $\mathcal{E}_\beta(t)$ *is defined by induction on the structure of* t *as follows:*

$$\begin{array}{ll}
\mathcal{E}_\top(t) & = () \\
\mathcal{E}_\bot(c) & = c \\
\mathcal{E}_\bot(x^\bot) & = x \\
\mathcal{E}_\bot(\lambda x^\beta : \tau.\, t) & = \lambda x : \mathcal{E}_\beta(\tau).\, \mathcal{E}_\bot(t) \\
\mathcal{E}_\bot(\mathsf{rec}\ f^\bot \colon \tau_2^{\beta_2} \overset{(\theta,\top)}{\Longrightarrow} \tau_1.\, t) & = \mathsf{rec}\ f \colon \mathcal{E}_\bot(\tau_2^{\beta_2} \overset{(\theta,\top)}{\Longrightarrow} \tau_1).\, \mathcal{E}_\bot(t) \\
\mathcal{E}_\bot(r^\bot := v) & = r := \mathcal{E}_\bot(v) \\
\mathcal{E}_\bot(!r^\bot) & = !r \\
\mathcal{E}_\bot(\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2) & = \mathsf{if}\ \mathcal{E}_\bot(v)\ \mathsf{then}\ \mathcal{E}_\bot(t_1)\ \mathsf{else}\ \mathcal{E}_\bot(t_2) \\
\mathcal{E}_\bot(t\ v) & = \mathcal{E}_\bot(t)\ \mathcal{E}_{\beta'}(v) \quad \textit{where } t \textit{ has type } \tau_2{}^{\beta'} \overset{(\theta_1,\rho_1)}{\Longrightarrow} \tau_1 \\
\mathcal{E}_\bot(\mathsf{let}\ x^{\beta'} = t_2\ \mathsf{in}\ t_1) & = \mathsf{let}\ x = \mathcal{E}_{\beta'}(t_2)\ \mathsf{in}\ \mathcal{E}_\bot(t_1)
\end{array}$$

Note that a ghost variable $x^\top$ or a ghost reference $r^\top$ does not occur anymore in $\mathcal{E}_\bot(t)$. 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 it binds a variable to (). More generally, $\mathcal{E}_\bot(t)$ is a value if and only if t is a value.

Leaving unit values and arguments in the outcome of erasure may seem intrusive. Indeed, after instrumenting a MiniML program with ghost code (for the purpose of verification) and then erasing it, we will not recover the original program. However, the program after erasure is semantically equivalent to the original one, with only extra administrative reduction steps (a term coined by Plotkin [8]) that do not impact the program complexity and can be eliminated at compile time. These reductions facilitate the proofs of forthcoming theorems.

3.1 Well-typedness Preservation

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

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

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

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

Theorem 1 (Well-typedness Preservation). *If* $\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t : \tau,\ \bot,\ (\theta,\ \rho)$ *holds, then* $\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma) \vdash \mathcal{E}_{\bot}(t) : \mathcal{E}_{\bot}(\tau),\ (\theta, \rho)$ *holds.*

Proof. By induction on the typing derivation, with case analysis on the last applied rule.

Case (T_g-λ). $$\frac{\Sigma,\ \Gamma,\ x^{\beta} : \tau \vdash_{\boldsymbol{g}} t : \tau_0,\ \bot,\ (\theta,\ \rho)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \lambda x^{\beta} : \tau.\, t : \tau^{\beta} \overset{(\theta,\rho)}{\Longrightarrow} \tau_0,\ \bot,\ (\bot,\ \bot)}$$

By induction hypothesis, we have $\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma),\ x : \mathcal{E}_{\beta}(\tau) \vdash \mathcal{E}_{\bot}(t) : \mathcal{E}_{\bot}(\tau_0),\ (\theta,\ \rho))$. Therefore, we conclude that

$$\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma) \vdash \lambda x : \mathcal{E}_{\beta}(\tau).\, \mathcal{E}_{\bot}(t) : \mathcal{E}_{\beta}(\tau) \overset{(\theta,\rho)}{\Longrightarrow} \mathcal{E}_{\bot}(\tau_0),\ (\bot,\ \bot)$$

Case (T_g-App-Regular).

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t : \tau_2^{\bot} \overset{(\theta_1,\rho_1)}{\Longrightarrow} \tau_1,\ \beta_1,\ (\theta_2,\ \rho_2) \qquad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} v : \tau_2,\ \beta_2,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} (t\ v) : \tau_1,\ \beta_1 \vee \beta_2,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)}$$

By hypothesis $\beta_1 \vee \beta_2 = \bot$. By induction hypotheses on the rule premises,

$$\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma) \vdash \mathcal{E}_{\bot}(t) : \mathcal{E}_{\bot}(\tau_2) \overset{(\theta_1,\rho_1)}{\Longrightarrow} \mathcal{E}_{\bot}(\tau_1),\ (\theta_2,\ \rho_2)$$

$$\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma) \vdash \mathcal{E}_{\bot}(v) : \mathcal{E}_{\bot}(\tau_2),\ (\bot,\ \bot).$$

Thus, $\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma) \vdash \mathcal{E}_{\bot}(t)\ \mathcal{E}_{\bot}(v) : \mathcal{E}_{\bot}(\tau_1),\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)$

Case (T_g-App-Ghost).

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t : \tau_2^{\top} \overset{(\theta_1,\rho_1)}{\Longrightarrow} \tau_1,\ \beta_1,\ (\theta_2,\ \rho_2) \qquad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} v : \tau_2,\ \beta_2,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} (t\ v) : \tau_1,\ \beta_1,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)}$$

By hypothesis $\beta_1 = \bot$. Thus, $\mathcal{E}_\bot(t\ v) = \mathcal{E}_\bot(t)\ ()$. By induction hypothesis,

$$\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma)\ \vdash\ \mathcal{E}_\bot(t)\ :\ \mathsf{unit}\ {}^{(\theta_1,\rho_1)}\!\Longrightarrow \mathcal{E}_\bot(\tau_1),\ (\theta_2,\ \rho_2)$$

Since $\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma)\ \vdash\ ()\ :\ \mathsf{unit},\ (\bot,\ \bot)$, we conclude that

$$\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma)\ \vdash\ \mathcal{E}_\bot(t)\ ()\ :\ \mathcal{E}_\bot(\tau_1),\ (\theta_1\ \vee\ \theta_2,\ \rho_1\ \vee\ \rho_2)$$

Case (T_g-Assign): $\dfrac{\Sigma,\ \Gamma\ \vdash_{\boldsymbol{g}}\ v\ :\ \kappa,\ \beta',\ (\bot,\ \bot) \qquad r^\beta\ :\ \mathsf{ref}\ \kappa\ \in\ \Sigma \qquad \beta'\ \Rightarrow\ \beta}{\Sigma,\ \Gamma\ \vdash_{\boldsymbol{g}}\ r^\beta\ :=\ v\ :\ \mathsf{unit},\ \beta,\ (\neg\beta,\ \bot)}$

By hypothesis $\beta = \bot$, and by premise side-condition $\beta'\ \Rightarrow\ \beta$, so $\beta' = \bot$ as well. Also, $r^\beta\ :\ \mathsf{ref}\ \kappa\ \in\ \Sigma$, so $r\ :\ \mathsf{ref}\ \kappa\ \in\ \mathcal{E}(\Sigma)$. Thus, by induction hypothesis on the rule premise, we have $\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma)\ \vdash\ \mathcal{E}_\bot(v)\ :\ \kappa,\ (\bot,\ \bot)$. Therefore:

$$\frac{\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma)\ \vdash\ \mathcal{E}_\bot(v)\ :\ \kappa,\ (\bot,\ \bot) \qquad r\ :\ \mathsf{ref}\ \kappa\ \in\ \mathcal{E}(\Sigma)}{\mathcal{E}(\Sigma),\ \mathcal{E}(\Gamma)\ \vdash\ r\ :=\ \mathcal{E}_\bot(v)\ :\ \mathsf{unit},\ (\top,\ \bot)}$$

Other cases are easily proved in a similar way. ∎

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).**

$$\begin{aligned}\mathcal{E}(\emptyset) &= \emptyset\\ \mathcal{E}(\mu \uplus \{r^\top \mapsto c\}) &= \mathcal{E}(\mu)\\ \mathcal{E}(\mu \uplus \{r^\bot \mapsto c\}) &= \mathcal{E}(\mu) \uplus \{r \mapsto c\}\end{aligned}$$

The correctness of erasure means that, for any evaluation $t \mid \mu\ \to_g^\star\ v \mid \mu'$ in GhostML, we have $\mathcal{E}_\bot(t) \mid \mathcal{E}(\mu)\ \to_g^\star\ \mathcal{E}_\bot(v) \mid \mathcal{E}(\mu')$ in MiniML and that, for any diverging evaluation $t \mid \mu\ \to_g\ \infty$ in GhostML, we have $\mathcal{E}_\bot(t) \mid \mathcal{E}(\mu)\ \to\ \infty$ 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 (Substition Under Erasure). *Let t be some GhostML term and v some GhostML value such that $\Sigma,\ \Gamma, x^\beta\ :\ \tau\ \vdash_{\boldsymbol{g}}\ t\ :\ \tau_0,\ \bot,\ (\theta, \rho)$ and $\Sigma,\ \Gamma\ \vdash_{\boldsymbol{g}}\ v\ :\ \tau,\ \beta',\ (\bot, \bot)$ where $\beta \geq \beta'$, hold. Then the following holds:*

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

Proof. By straightforward induction on the structure of t. ∎

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

Lemma 6 (Forward Simulation). *If* $\Sigma,\ \emptyset \vdash_{\boldsymbol{g}} t : \tau,\ \bot,\ (\theta,\ \rho)$ *and, for some store* μ *such that* $\Sigma \vdash_g \mu$*, we have* $t \mid \mu \rightarrow_g t' \mid \mu'$*, then the following holds in MiniML:* $\mathcal{E}_\bot(t) \mid \mathcal{E}_\bot(\mu) \rightarrow^{0|1} \mathcal{E}_\bot(t') \mid \mathcal{E}_\bot(\mu')$.

Proof. By induction on the derivation $\rightarrow_g$.

Case (E-Head). By case analysis on the head reduction $\xrightarrow{\epsilon}_g$. We only give the proof for (E-App-λ); other cases are proved in a similar way.

Sub-Case (E-App-λ). $(\lambda x^{\beta} : \tau.\, t)\ v \xrightarrow{\epsilon}_g t[x^{\beta} \leftarrow v]$.

If $\beta = \top$, then

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \lambda x^{\top}{:}\tau_2.\, t : \tau_2^{\top} \overset{(\theta_1,\rho_1)}{\Longrightarrow} \tau_1,\ \bot,\ (\theta_2,\ \rho_2) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} v : \tau_2,\ \beta_2,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} (\lambda x^{\top}{:}\tau_2.\, t\ v) : \tau_1,\ \bot,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)}$$

Therefore, by substitution under erasure (Lemma 5), we get

$$\mathcal{E}_\bot((\lambda x^{\top} : \tau.\, t)\ v) = (\lambda x : \mathsf{unit}.\, \mathcal{E}_\bot(t))\ () \xrightarrow{\epsilon} \mathcal{E}_\bot(t)[x \leftarrow ()] = \mathcal{E}_\bot(t[x^{\top} \leftarrow v]).$$

If $\beta = \bot$, then we have

$$\frac{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \lambda x^{\bot}{:}\tau_2.\, t : \tau_2^{\bot} \overset{(\theta_1,\rho_1)}{\Longrightarrow} \tau_1,\ \bot,\ (\theta_2,\ \rho_2) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} v : \tau_2,\ \bot,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} (\lambda x^{\bot}{:}\tau_2.\, t\ v) : \tau_1,\ \bot,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)}$$

Again, by substitution under erasure (Lemma 5), we get

$$\mathcal{E}_\bot((\lambda x^{\bot} : \tau.\, t)\ v) \xrightarrow{\epsilon} \mathcal{E}_\bot(t)[x \leftarrow \mathcal{E}_\bot(v)] = \mathcal{E}_\bot(t[x^{\bot} \leftarrow v]).$$

Case (E-Context-Let). $\dfrac{t_2 \mid \mu \rightarrow_g t_2' \mid \mu'}{\mathsf{let}\ x^{\beta} = t_2\ \mathsf{in}\ t_1 \mid \mu \rightarrow_g \mathsf{let}\ x^{\beta} = t_2'\ \mathsf{in}\ t_1 \mid \mu'}$.

If $\beta = \top$, then $\dfrac{\Sigma,\ \Gamma,\ x^{\top} : \tau_2 \vdash_{\boldsymbol{g}} t_1 : \tau_1,\ \bot,\ (\theta_1,\ \rho_1) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t_2 : \tau_2,\ \beta_2,\ (\bot,\ \bot)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \mathsf{let}\ x^{\top} = t_2\ \mathsf{in}\ t_1 : \tau_1,\ \bot,\ (\theta_1,\ \rho_1)}$

By store preservation (Lemma 3), $\mu_{|\bot} = \mu'_{|\bot}$, so $\mathcal{E}(\mu) = \mathcal{E}(\mu')$. Therefore,

$$\mathcal{E}_\bot(\mathsf{let}\ x^{\top} = t_2\ \mathsf{in}\ t_1) \mid \mathcal{E}(\mu) \rightarrow_g^0 \mathcal{E}_\bot(\mathsf{let}\ x^{\top} = t_2'\ \mathsf{in}\ t_1) \mid \mathcal{E}(\mu').$$

If $\beta = \bot$, then $\dfrac{\Sigma,\ \Gamma,\ x^{\bot} : \tau_2 \vdash_{\boldsymbol{g}} t_1 : \tau_1,\ \bot,\ (\theta_1,\ \rho_1) \quad \Sigma,\ \Gamma \vdash_{\boldsymbol{g}} t_2 : \tau_2,\ \bot,\ (\theta_2,\ \rho_2)}{\Sigma,\ \Gamma \vdash_{\boldsymbol{g}} \mathsf{let}\ x^{\bot} = t_2\ \mathsf{in}\ t_1 : \tau_1,\ \bot,\ (\theta_1 \vee \theta_2,\ \rho_1 \vee \rho_2)}$

By induction hypothesis on sub-derivation $t_2 \mid \mu \rightarrow_g t_2' \mid \mu'$, we get

$$\mathcal{E}_\bot(t_2) \mid \mathcal{E}_\bot(\mu) \rightarrow^{0|1} \mathcal{E}_\bot(t_2') \mid \mathcal{E}_\bot(\mu').$$

The result trivially holds when this reduction has 0 step. Otherwise, we have

$$\frac{\mathcal{E}_\bot(t_2) \mid \mathcal{E}_\bot(\mu) \rightarrow \mathcal{E}_\bot(t_2') \mid \mathcal{E}_\bot(\mu')}{\mathsf{let}\ x = \mathcal{E}_\bot(t_2)\ \mathsf{in}\ \mathcal{E}_\bot(t_1) \mid \mathcal{E}_\bot(\mu) \rightarrow \mathsf{let}\ x = \mathcal{E}_\bot(t_2')\ \mathsf{in}\ \mathcal{E}_\bot(t_1) \mid \mathcal{E}_\bot(\mu')}$$

The remaining case (E-Context-App) is similar. ∎

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

Theorem 2 (Terminating Evaluation Preservation). *If* $\Sigma,\ \emptyset\ \vdash_{\boldsymbol{g}}\ t\ :\ \tau,\ \bot,\ (\theta,\ \rho)$ *holds and* $t\ \mid \mu\ \rightarrow_g^\star\ v\ \mid \mu'$, *for some value* v *and some store* μ *such that* $\Sigma \vdash_g \mu$, *then* $\mathcal{E}_\bot(t)\ \mid \mathcal{E}_\bot(\mu)\ \rightarrow^\star\ \mathcal{E}_\bot(v)\ \mid \mathcal{E}_\bot(\mu')$.

Proof. By induction on the length of the evaluation $t\ \mid \mu\ \rightarrow_g^\star\ v\ \mid \mu'$. If $t\ \mid \mu\ \rightarrow_g^0\ v\ \mid \mu'$, then the result trivially holds since $t = v$ and $\mu' = \mu$. Now, assume that $t\ \mid \mu\ \rightarrow_g^1\ t''\ \mid \mu''\ \rightarrow_g^n\ v\ \mid \mu'$ for some intermediate store μ''. By preservation (Lemma 1), $\Sigma,\ \Gamma\ \vdash_{\boldsymbol{g}}\ t''\ :\ \tau,\ \bot,\ (\theta'',\ \rho'')$ and $\Sigma \vdash_g \mu''$ for some θ'' and ρ'', such that $\theta' \geq \theta''$ and $\rho' \geq \rho''$. By induction hypothesis on t'', $\mathcal{E}_\bot(t'')\ \mid \mathcal{E}_\bot(\mu'')\ \rightarrow^\star\ \mathcal{E}_\bot(v)\ \mid \mathcal{E}_\bot(\mu')$. By forward simulation (Lemma 6), $\mathcal{E}_\bot(t)\ \mid \mathcal{E}_\bot(\mu)\ \rightarrow^{0|1}\ \mathcal{E}_\bot(t'')\ \mid \mathcal{E}_\bot(\mu'')$. Putting pieces together we conclude $\mathcal{E}_\bot(t)\ \mid \mathcal{E}_\bot(\mu)\ \rightarrow^{0|1}\ \mathcal{E}_\bot(t'')\ \mid \mathcal{E}_\bot(\mu'')\ \rightarrow^\star\ \mathcal{E}_\bot(v)\ \mid \mathcal{E}_\bot(\mu')$. ∎

To prove the second part of the main theorem (non-termination preservation), we use the following backward simulation argument.

Lemma 7 (Backward Simulation). *If* $\Sigma,\ \emptyset\ \vdash_{\boldsymbol{g}}\ t\ :\ \tau,\ \bot,\ (\theta,\ \rho)$ *holds, then, for any store* μ *such that* $\Sigma \vdash_g \mu$, *if* $\mathcal{E}_\bot(t)\ \mid \mathcal{E}(\mu)\ \rightarrow\ q\ \mid \nu$ *for some term* q *and some state* ν, *then* $t\ \mid \mu\ \rightarrow_g^{\geq 1}\ t'\ \mid \mu'$ *where* $\mathcal{E}_\bot(t') = q$ *and* $\mathcal{E}(\mu') = \nu$.

Proof. By induction on the term t.

Case 1: t is $\mathsf{let}\ x^\top\ =\ t_2\ \mathsf{in}\ t_1$. If t_2 is not a value, then by GhostML progress (Lemma 2), it can be reduced. By hypothesis, t is well-typed, with a typing judgment for t_2 being $\Sigma,\ \emptyset\ \vdash_{\boldsymbol{g}}\ t_2\ :\ \tau,\ \beta,\ (\bot, \bot)$. By program termination (Lemma 4), we have that $t_2\ \mid \mu\ \rightarrow_g^\star\ v\ \mid \mu'$ for some v and μ'. Consequently,

$$\mathsf{let}\ x^\top\ =\ t_2\ \mathsf{in}\ t_1\ \mid \mu\ \rightarrow_g^\star\ \mathsf{let}\ x^\top\ =\ v\ \mathsf{in}\ t_1\ \mid \mu'\ \rightarrow_g\ t_1[x^\top \leftarrow v]\ \mid \mu'$$

Therefore we have

$$\mathcal{E}_\bot(t)\ \mid \mathcal{E}(\mu)\ \rightarrow^0\ \mathcal{E}_\bot(\mathsf{let}\ x^\top\ =\ v\ \mathsf{in}\ t_1)\ \mid \mathcal{E}(\mu)\ \rightarrow\ \mathcal{E}_\bot(t_1)[x \leftarrow ()]\ \mid \mathcal{E}(\mu)$$

By t-erasure definition, $\mathcal{E}_\bot(t_1)[x \leftarrow ()] = \mathcal{E}(t_2[x^\top \leftarrow v])$. Finally, by store preservation (Lemma 3), $\mu_{|\bot} = \mu'_{|\bot}$. Therefore, $\mathcal{E}(\mu) = \mathcal{E}(\mu')$, which allows us to to conclude.

Case 2: t is $\mathsf{let}\ x^\bot\ =\ t_2\ \mathsf{in}\ t_1$. We have by hypothesis that t_1 is regular code. Then we also have that t_2 is regular too. If t_2 is a value, the result follows immediately. If t_2 is not a value, then the result follows from the results of GhostML soundness (safety and anti-monotony of statuses), using the induction hypothesis on t_2, since t-erasure preserves the structure of regular terms.

Other cases are proved in a similar way. ∎

Finally, we establish the second part of the main theorem:

Theorem 3 (Non-termination Preservation). *If* $\Sigma,\ \emptyset\ \vdash_{\boldsymbol{g}}\ t\ :\ \tau,\ \bot,\ (\theta,\ \rho)$ *holds and* $t\ \mid \mu\ \rightarrow_g\ \infty$, *for some store* μ *such that* $\Sigma \vdash_g \mu$, *then* $\mathcal{E}_\bot(t)$ *also diverges, that is* $\mathcal{E}_\bot(t)\ \mid \mathcal{E}(\mu)\ \rightarrow\ \infty$.

Proof. By co-induction on $t \mid \mu \to_g \infty$. Observe that, since t diverges, t is not a value. Thus $\mathcal{E}_\bot(t)$ is not a value either. By well-typedness preservation (Theorem 1), $\mathcal{E}_\bot(t)$ is well-typed, with $\mathcal{E}(\Sigma), \emptyset \vdash \mathcal{E}_\bot(t) : \mathcal{E}_\bot(\tau), (\theta, \rho)$. We have $\mathcal{E}(\Sigma) \vdash \mathcal{E}(\mu)$, since $\Sigma \vdash_g \mu$. Therefore, by progress in MiniML, there exist some term q and some store ν such that $\mathcal{E}_\bot(t) \mid \mu \to q \mid \nu$. By backward simulation (Lemma 7), $t \mid \mu \to_g^{\geq 1} t' \mid \mu'$ with $q = \mathcal{E}_\bot(t')$ and $\nu = \mathcal{E}_\bot(\mu')$. Thus, the reduction $t \mid \mu \to_g \infty$ decomposes into

$$t \mid \mu \to_g^{\geq 1} t' \mid \mu' \to_g \infty$$

By preservation (Lemma 1), $\Sigma, \Gamma \vdash_{\boldsymbol{g}} t' : \tau, \bot, (\theta, \rho)$ and $\Sigma \vdash_g \mu'$. Finally, by co-induction hypothesis, we get $\mathcal{E}_\bot(q) \mid \nu \to \infty$ and thus $\mathcal{E}_\bot(t) \mid \mathcal{E}(\mu) \to \infty$. ∎

4 Implementation

Our method of ghost code handling is implemented in the verification tool Why3[3]. 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 Hindley-Milner type polymorphism. Our approach to associate ghost status with variables and expressions, and not with types, makes this extension straightforward.

Local References. Anothier obvious extension of GhostML is the support of non-global references. As long as such a reference cannot be an alias for another, existing 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 in 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.

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.

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

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

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 implementations additionally stores the pure logical view of the queue as a list, in a 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)` 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 on-line gallery of verified Why3 programs contains several other examples of use of ghost code[4], in particular, ghost function parameters and ghost functions to supply automatic induction proofs (aka 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 [9] and Reynolds [10], the notion of auxiliary variable was first introduced by Lucas in 1968 [11]. Since then, numerous authors have adapted this technique in various domains.

It is worth pointing out that some authors, in particular Kleymann [12] and Reynolds [10], 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.* [13] 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, the non-interference is ensured by

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

```
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.

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 [14] 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." [15]. In addition, a class can contain both ghost fields and regular fields. Dafny ensures termination of ghost code. However, apart from base types and operations, regular data types, such as arrays, cannot be reused in ghost code.

The property of non-interference of ghost code is not that different from non-interference is the context of information flow [16]. Indeed, one could see ghost code as high security level information and regular code as low level information, and non-interference precisely means that low level information is not computed out of high level one. Information flow can be checked using a type system [17] and proofs in that domain typically involve a bisimulation technique (though not necessarily through an erasure operation). Yet reducing the problem of ghost code analysis to information flow analysis still has to be investigated in details.

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, play 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. 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. `forall` loop construct in Dafny).

References

1. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In Springer, ed.: LPAR-16. Volume 6355. (2010) 348–370
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: Theorem Proving in Higher Order Logics (TPHOLs). Volume 5674 of Lecture Notes in Computer Science., Springer (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.: Proceedings of the 22nd European Symposium on Programming. Volume 7792 of Lecture Notes in Computer Science., Springer (March 2013) 125–128
5. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. SIGPLAN Not. **28**(6) (June 1993) 237–247
6. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115** (1992) 38–94
7. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
8. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2) (1975) 125–159
9. Jones, C.B., Roscoe, A., Wood, K.R.: Reflections on the Work of C.A.R. Hoare. 1st edn. Springer Publishing Company, Incorporated (2010)
10. Reynolds, J.C.: The craft of programming. Prentice Hall International series in computer science. Prentice Hall (1981)
11. Lucas, P.: Two constructive realizations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory, Vienna (June 1968)
12. Kleymann, T.: Hoare logic and auxiliary variables. Formal Asp. Comput. **11**(5) (1999) 541–566
13. Zhang, Z., Feng, X., Fu, M., Shao, Z., Li, Y.: A structural approach to prophecy variables. In Agrawal, M., Cooper, S., Li, A., eds.: 9th annual conference on Theory and Applications of Models of Computation (TAMC). Volume 7287 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2012) 61–71
14. Schmaltz, S.: Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C. PhD thesis, Saarland University, Saarbrücken (2013)
15. Leino, K.R.M., Moskal, M.: Co-induction simply: Automatic co-inductive proofs in a program verifier. Technical Report MSR-TR-2013-49, Microsoft Research (July 2009) Available from `http://research.microsoft.com/pubs`.
16. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM **20**(2) (July 1977) 504–513
17. Pottier, F., Conchon, S.: Information flow inference for free. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montréal, Canada (September 2000) 46–57