The Spirit of Ghost Code

Jean-Christophe Filliâtre, Léon Gondelman, Andrei Paskevich

To cite this version:

Jean-Christophe Filliâtre, Léon Gondelman, Andrei Paskevich. The Spirit of Ghost Code. Formal Methods in System Design, Springer Verlag, 2016, 48 (3), pp.152-174. 10.1007/s10703-016-0243-x . hal-01396864

HAL Id: hal-01396864
https://hal.archives-ouvertes.fr/hal-01396864

Submitted on 15 Nov 2016

Noname manuscript No.
(will be inserted by the editor)

The Spirit of Ghost Code

Jean-Christophe Filliâtre ·
Léon Gondelman · Andrei Paskevich

Received: date / Accepted: date

Abstract In the context of deductive program verification, ghost code is a 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, ghost code deserves rigorous definition and treatment, 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.

Keywords Ghost code · Deductive software verification

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

This work is partly supported by the Bware (ANR-12-INSE-0010, http://bware.lri.fr/) project of the French national research agency (ANR). This is an extended version of a conference paper [1].

Jean-Christophe Filliâtre · Léon Gondelman · Andrei Paskevich
Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405,
Inria Saclay – Île-de-France, Orsay, F-91893

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, 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 [2,3,4,5], it deserves a proper formal treatment. 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 [5]. The language presented is this paper is deliberately simplified. The more exciting features, listed in Section 5 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.

This paper is organized as follows. Sec. 2 illustrates the use of ghost code on an example. Sec. 3 introduces an ML-like language with ghost code. Sec. 4 defines the operation of ghost code erasure and proves its soundness. Sec. 5 describes the actual implementation in Why3. We conclude with related work in Sec. 6 and perspectives in Sec. 7.

2 A Tiny Program Featuring Ghost Code

Consider the following program `fibo` that computes $\mathcal{F}_{\mathtt{n}}$, the n-th Fibonacci number, using an auxiliary function `aux`:

```
let rec aux a b n =
  if n = 0 then a else aux b (a+b) (n-1)

let fibo n = aux 0 1 n
```

To prove the correctness of this program using deductive verification, we supply a suitable specification. Let us assume a specification language with contracts [6], where `requires` introduces a precondition and `ensures` introduces a postcondition. A possible specification would be the following:

```
let rec aux (a b n: int) : int
  requires {$0 \leq \mathtt{n}$}
  requires {$\exists \mathtt{k}. 0 \leq \mathtt{k} \land \mathtt{a} = \mathcal{F}_{\mathtt{k}} \land \mathtt{b} = \mathcal{F}_{\mathtt{k}+1}$}
  ensures  {$\exists \mathtt{k}. 0 \leq \mathtt{k} \land \mathtt{a} = \mathcal{F}_{\mathtt{k}} \land \mathtt{b} = \mathcal{F}_{\mathtt{k}+1} \land \mathtt{result} = \mathcal{F}_{\mathtt{k}+\mathtt{n}}$}
= if n = 0 then a else aux b (a+b) (n-1)

let fibo (n: int) : int
  requires {$0 \leq \mathtt{n}$}
  ensures  {$\mathtt{result} = \mathcal{F}_{\mathtt{n}}$}
= aux 0 1 n
```

Even if acceptable, this specification has several drawbacks: first, some information is duplicated between the pre- and the postcondition; second, it uses existential quantifiers, which makes the specification less readable and the proof less amenable for automation.

One way to simplify the specification is to turn quantifiers into something more constructive, where the existence of `k` is materialized by an extra argument to function `aux`, as follows:

```
let rec aux (k: int) (a b n: int) : int
  requires {$0 \leq \mathtt{k} \land 0 \leq \mathtt{n}$}
  requires {$\mathtt{a} = \mathcal{F}_{\mathtt{k}} \land \mathtt{b} = \mathcal{F}_{\mathtt{k}+1}$}
  ensures  {$\mathtt{result} = \mathcal{F}_{\mathtt{k}+\mathtt{n}}$}
= if n = 0 then a else aux (k+1) b (a+b) (n-1)
```

When performing the recursive call to `aux`, we provide the suitable value for this extra argument. Similarly, we provide the initial value for `k` when calling `aux` in `fibo`:

```
let fibo (n: int) : int
  requires {$0 \leq \mathtt{n}$}
  ensures  {$\mathtt{result} = \mathcal{F}_{\mathtt{n}}$}
= aux 0 0 1 n
```

Such an extra computation, used only for the purpose of specification, is precisely what we call ghost code (here in blue). The point is that we can erase ghost code, once the verification is completed, to recover the original program. On this example, it is easy to check that ghost computations have no impact on the program outcome.

Generally speaking, we need a way to check/enforce non-interference of ghost code with regular computations. The purpose of this article is to introduce a type system to do this.

This example also illustrates a desirable feature of ghost code, that is, the ability to reuse the same data types and operations in both regular and ghost code. Here, we reuse type `int`, numeric constants, and addition. Generally speaking, we may want to do this with arbitrary user-defined types and operations. We will show that our type system is sufficiently flexible to permit such reuse.

3 GhostML

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

3.1 Syntax

The syntax of GhostML types and expressions is given in Fig. 1 and 2, respectively. Types are either primitive data types or function types. A function type is an arrow $\tau_1^{\beta} \overset{\epsilon}{\Longrightarrow} \tau_2$, where τ_1 is the type of the argument and τ_2 is the type of the result. The superscript β attached to t_1 indicates the ghost status of the parameter: $\top$ denotes a ghost parameter and $\bot$ a regular one (here and below, “regular” stands for “non-ghost”). The latent effect of the function, denoted ϵ, is a Boolean value which indicates possible non-termination or modification of a regular reference. This latent effect is realised whenever the function is applied.

τ ::=	TYPES
$\mid \kappa$	*primitive type*
$\mid \tau^{\beta} \overset{\epsilon}{\Longrightarrow} \tau$	*functional type*
κ ::=	PRIMITIVE TYPES
$\mid$ int $\mid$ bool $\mid$ unit	*primitive types*
$\beta \in \{\bot, \top\}$	GHOST STATUS
$\epsilon \in \{\bot, \top\}$	EFFECT

Fig. 1 Types and effects.

Terms are either values or compound expressions like application, conditional, reference access, and modification. All the language constructions are standard ML, except for the keyword ghost which turns a term into ghost code. We write $x^{\beta}, y^{\beta}, f^{\beta}, g^{\beta}$ and r^{β}, s^{β} to denote respectively variables and references, that is, we distinguish references and variables syntactically. Unlike variables, references are not valid expressions, they can only be used in a term via assignment and dereference. We assume a fixed finite set of global references.

$t ::=$	TERMS
$\mid v$	*value*
$\mid t\ v$	*application*
$\mid \mathsf{let}\ x^{\beta} = t\ \mathsf{in}\ t$	*local binding*
$\mid \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t$	*conditional*
$\mid r^{\beta} := v$	*assignment*
$\mid\ !r^{\beta}$	*dereference*
$\mid \mathsf{ghost}\ t$	*ghost code*
$v ::=$	VALUES
$\mid x^{\beta}$	*variable*
$\mid c$	*constant*
$\mid \lambda x^{\beta} : \tau .\, t$	*anonymous function*
$\mid \mathsf{rec}\ f^{\beta} : \tau^{\beta} \overset{\epsilon}{\Longrightarrow} \tau .\, \lambda x^{\beta} : \tau .\, t$	*recursive function*
$c ::=$	CONSTANTS
$\mid ()$	*unit*
$\mid \ldots, -1, 0, 1, \ldots$	*integers*
$\mid \mathsf{true}, \mathsf{false}$	*Booleans*
$\mid +, \vee, =, \ldots$	*operators*

Fig. 2 Syntax of terms.

Every variable, function parameter, and reference is annotated with its ghost status β. Consider the following example:

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

Here, ghost function $upd^{\top}$ takes one regular parameter $x^{\perp}$ and assigns it to a ghost reference $s^{\top}$. Then $upd^{\top}$ is applied to the contents of a regular reference $r^{\perp}$. In practice, some ghost status annotations (e.g., those of locally defined variables) can be inferred using the typing rules, and do not need to be supplied by the user.

Note that compound terms follow a variant of *A-normal* form [7]. 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 $(t_1\ t_2)$ can be rewritten as $\mathsf{let}\ x^{\beta} = t_2\ \mathsf{in}\ t_1\ x^{\beta}$, where β is the ghost status of the first formal parameter of t_1.

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 $\perp$ and where terms of the form $\mathsf{ghost}\ t$ do not appear.

3.2 Semantics

In Fig. 3, we give a small-step operational semantics to GhostML which corresponds to a deterministic call-by-value reduction strategy. An execution state is a pair $\mu \cdot t$ of

$$\mathsf{ghost}\ t_1 \ \rightarrow\ t_1 \qquad \text{(E-Ghost)}$$

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

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

$$(\lambda x^{\beta} : \tau.\, t_1)\ v \ \rightarrow\ t_1[x^{\beta} \leftarrow v] \qquad \text{(E-App-}\lambda\text{)}$$

$$\frac{v_0 = \mathsf{rec}\ f^{\beta} : \tau^{\beta} \overset{\epsilon}{\Longrightarrow} \tau.\, \lambda x^{\beta} : \tau.\, t_1}{(v_0\ v) \ \rightarrow\ t_1[x^{\beta} \leftarrow v, f^{\beta} \leftarrow v_0]} \qquad \text{(E-App-Rec)}$$

$$\mathsf{let}\ x^{\beta} = v\ \mathsf{in}\ t_1 \ \rightarrow\ t_1[x^{\beta} \leftarrow v] \qquad \text{(E-Let)}$$

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

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

$$\frac{\mu(r^{\beta}) = c}{\mu \cdot !r^{\beta} \ \rightarrow\ \mu \cdot c} \qquad \text{(E-Deref)}$$

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

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

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

Fig. 3 Semantics.

a term t and a store μ, which maps global references to constants. Each reduction rule defines a state transition during one step of execution. The contextual reduction rules (E-Context-App) and (E-Context-Let) identify the subterm to be reduced (called the *redex*) whereas other rules apply to redex itself. In the rules, we omit the store μ when it is not changed during the reduction step.

Rule (E-Ghost) expresses that, from the point of view of operational semantics, there is no difference between regular and ghost code. Other rules are standard. For instance, rules (E-Op-λ) and (E-Op-δ) evaluate the application of a constant c_0 to constants $c_1 \ldots c_m$. Such an application either is partial ($1 \leq m < \mathsf{arity}(c_0)$), and then turns into a function $\lambda x^{\bot} : \kappa.\, c_0\ c_1 \ldots c_m\ x^{\bot}$, or is total ($m = \mathsf{arity}(c_0)$), 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.

As usual, $\rightarrow^{\star}$ denotes the reflexive, transitive closure of $\rightarrow$. We say that a closed term t evaluates to value v in a store μ if there is a μ' such that $\mu \cdot t \ \rightarrow^{\star}\ \mu' \cdot v$. Note that, since t is closed, and the reduction rules do not free variables, v is either a constant or a function, and not a variable. Finally, the divergence of a term t in a store μ is defined co-inductively as follows:

$$\frac{\mu \cdot t \ \to\ \mu' \cdot t' \qquad \mu' \cdot t' \ \to\ \infty}{\mu \cdot t \ \to\ \infty}\,(\textsc{E-Div})$$

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: $\mu \cdot t \ \to_m\ \mu' \cdot t'$.

3.3 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 judgement is

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

Here, τ is the type of term t. Boolean values β and ϵ indicate, respectively, the ghost status of t and its regular side effects. Typing environment Γ binds variables to types. Store typing Σ binds each global reference r^{β} to the primitive type of the stored value. We restrict types of stored values to primitive types in order to avoid a possible non-termination via Landin's knot (that is, recursion encoded using a reference containing a function), which would be undetected in our type system.

Typing rules are given in Fig. 4. To account for non-interference, each rule whose conclusion is a judgement $\Gamma \cdot \Sigma \vdash t : \tau \cdot \beta \cdot \epsilon$ comes with the implicit side condition

$$\beta \implies \neg\epsilon \wedge \neg\epsilon^{+}(\tau) \tag{1}$$

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

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

In other words, side condition (1) stands for the type system invariant to ensure that, whenever t is ghost, it must terminate and must not modify regular references. Note that a weaker form of the invariant $\beta \implies \neg\epsilon$ is acceptable, but we strengthen it with $\neg\epsilon^{+}(\tau)$ to reject functions whose latent effect would violate the non-interference once the function is applied.

Let us explain some rules in detail. The rule (T-Const) states that any constant c is regular, pure, and terminating (i.e., $\beta = \bot$ and $\epsilon = \bot$). 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, $\mathsf{Typeof}(+) = \mathsf{int}^{\bot} \overset{\bot}{\Longrightarrow} \mathsf{int}^{\bot} \overset{\bot}{\Longrightarrow} \mathsf{int}$.

$$\frac{\mathsf{Typeof}(c) = \tau}{\Gamma \cdot \Sigma \vdash c : \tau \cdot \bot \cdot \bot}\ \text{(T-Const)} \qquad \frac{\Gamma(x^{\beta}) = \tau}{\Gamma \cdot \Sigma \vdash x^{\beta} : \tau \cdot \beta \cdot \bot}\ \text{(T-Var)}$$

$$\frac{\Sigma(r^{\beta}) = \kappa}{\Gamma \cdot \Sigma \vdash\, !r^{\beta} : \kappa \cdot \beta \cdot \bot}\ \text{(T-Deref)} \qquad \frac{\Gamma \cdot \Sigma \vdash t_1 : \tau \cdot \beta \cdot \bot}{\Gamma \cdot \Sigma \vdash (\mathsf{ghost}\ t_1) : \tau \cdot \top \cdot \bot}\ \text{(T-Ghost)}$$

$$\frac{[x^{\beta} \mapsto \tau]\Gamma \cdot \Sigma \vdash t_1 : \tau_1 \cdot \beta_1 \cdot \epsilon}{\Gamma \cdot \Sigma \vdash (\lambda x^{\beta} : \tau . t_1) : \tau^{\beta} \overset{\epsilon}{\Longrightarrow} \tau_1 \cdot \beta_1 \cdot \bot}\ \text{(T-}\lambda\text{)}$$

$$\frac{\begin{array}{c} \epsilon_0 = \mathtt{CheckTermination}(\mathsf{rec}\ f^{\beta_1} : \tau_0 . \lambda x^{\beta} : \tau_2 . t_1) \qquad \tau_0 = (\tau_2^{\beta} \overset{\epsilon_1}{\Longrightarrow} \tau_1) \\ [f^{\beta_1} \mapsto \tau_0]\Gamma \cdot \Sigma \vdash (\lambda x^{\beta} : \tau_2 . t_1) : \tau_0 \cdot \beta_0 \cdot \bot \qquad \beta_1 \geq \beta_0 \qquad \epsilon_1 \geq \epsilon_0 \end{array}}{\Gamma \cdot \Sigma \vdash (\mathsf{rec}\ f^{\beta_1} : \tau_0 . \lambda x^{\beta} : \tau_2 . t_1) : \tau_0 \cdot \beta_1 \cdot \bot}\ \text{(T-Rec)}$$

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

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

$$\frac{\Gamma \cdot \Sigma \vdash t_1 : \tau_1 \cdot \beta_1 \cdot \bot \quad [x^{\top} \mapsto \tau_1]\Gamma \cdot \Sigma \vdash t_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash (\mathsf{let}\ x^{\top} = t_1\ \mathsf{in}\ t_2) : \tau_2 \cdot \beta_2 \cdot \epsilon_2}\ \text{(T-Let}^{\top}\text{)}$$

$$\frac{\Gamma \cdot \Sigma \vdash t_1 : \tau_2^{\bot} \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \beta_1 \cdot \epsilon_2 \quad \Gamma \cdot \Sigma \vdash v : \tau_2 \cdot \beta_2 \cdot \bot}{\Gamma \cdot \Sigma \vdash (t_1\ v) : \tau_1 \cdot \beta_1 \vee \beta_2 \cdot \epsilon_1 \vee \epsilon_2}\ \text{(T-App}^{\bot}\text{)}$$

$$\frac{\Gamma \cdot \Sigma \vdash t_1 : \tau_2^{\top} \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \beta_1 \cdot \epsilon_2 \quad \Gamma \cdot \Sigma \vdash v : \tau_2 \cdot \beta_2 \cdot \bot}{\Gamma \cdot \Sigma \vdash (t_1\ v) : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \vee \epsilon_2}\ \text{(T-App}^{\top}\text{)}$$

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

Fig. 4 Typing rules.

The rule (T-λ) shows that the ghost status of an abstraction is the ghost status of its body. Note also that effects ϵ of the body become latent, while function itself has no effects.

The rule (T-Rec) gives the typing for recursive functions. Recall that ghost code must always terminate, and that termination is in general undecidable. The easiest solution therefore would be to enforce that no recursive function is ever used in ghost code, that is, to assign recursive functions a latent non-termination effect. In practice, however, we want to accept recursive functions whose termination can be established by static analysis. To do so, we assume the existence of an oracle function `CheckTermination` which, given a recursive definition, returns $\bot$ if it can prove the function terminating, and $\top$ otherwise. By terminating, we mean that any call terminates, whatever the store and the argument. A typical example of such an oracle would be the generation of verification conditions from user-provided variants (as in our forthcoming example on page 21). Note that the oracle is not taking non-interference into account, so it does not have to distinguish between ghost and regular code. The rule (T-Rec) imposes that the latent effect of the recursive function (ϵ_1) is no smaller than the effect computed by the oracle, that is $\epsilon_1 \geq \epsilon_0$. Similarly, if the body of the recursive function is ghost ($\beta_0 = \top$), then the recursive function must be ghost as well ($\beta_1 = \top$), that is $\beta_1 \geq \beta_0$.

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 side condition (1) will reject conditionals where one part is ghost and another part has some effect, as in

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

The rule (T-Ghost) turns any term t into ghost code, provided that t is pure and terminating. Thus, terms like ghost $(r^{\perp} := 42)$ 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 $\top > \perp$.) Additionally, the rule conclusion ensures that ϵ is $\top$ if and only if the assigned reference is regular ($\beta = \perp$), *i.e.*, ghost reference assignments are not part of regular effects.

The most subtle rules are those for local binding and application. Rule (T-Let$^{\top}$) states that, whatever the ghost status of a term t_1 is, as long as t_1 is pure and terminating, we can bind a ghost variable $x^{\top}$ to t_1. Notice that the resulting let-expression is not necessarily ghost: the type system knows that the value stored in $x^{\top}$ is ghost and will ensure the non-interference of subsequent uses of $x^{\top}$. Similarly, by rule (T-App$^{\top}$) a function that expects a ghost parameter can be applied to both ghost and regular values. Once again, the result may be regular.

Consider the following example[1] where we trace an update of a regular reference $r^{\perp}$ by using a ghost function $succ^{\top}$:

$$\begin{array}{l}
\mathsf{let}\ succ^{\top} = \lambda y^{\top} : \mathsf{int}.\,(y^{\top} + 1)\ \mathsf{in} \\
\mathsf{let}\ x^{\perp} = !r^{\perp}\ \mathsf{in} \\
\mathsf{let}\ trace^{\top} = succ^{\top}\ x^{\perp}\ \mathsf{in} \\
r^{\perp} := x^{\perp} + 1
\end{array}$$

With the rule (T-App$^{\top}$), we can apply $succ^{\top}$ to the current value of $r^{\perp}$, even if we store it in the regular variable $x^{\perp}$. Then, thanks to rule (T-Let$^{\top}$) we are able to “contain” ghost computation ($succ^{\top}\ x^{\perp}$) in $trace^{\top}$ which may be used in subsequent computations without interfering with regular code.

Rule (T-Let$^{\perp}$) is somewhat dual to (T-Let$^{\top}$): when binding a regular variable $x^{\perp}$ to a ghost term, the type system expands the ghost status to the whole let-expression, “contaminating” it. Similarly, rule (T-App$^{\perp}$) allows us to pass a ghost value to a function expecting a regular parameter, in which case the application itself becomes ghost.

Let us look again at the previous example. We can notice that since successor function $succ^{\top}$ is ghost, we cannot use it in regular code and thus cannot replace $r^{\perp} := x^{\perp} + 1$ by $r^{\perp} := succ^{\top}\ x^{\perp}$. Given that the successor function is pure and

[1] For the sake of readability, we slightly relax the constraints of the A-normal form and accept arbitrary terms in the reference assignment.

terminating, it is more natural to define it as regular code and then reuse it in ghost computations. The rule (T-LET$^{\bot}$) allows us to improve the code in this way:

$\mathsf{let}\ \mathit{succ}^{\bot} = \lambda y^{\bot} : \mathsf{int}.\,(y^{\bot} + 1)\ \mathsf{in}$
$\mathsf{let}\ x^{\bot} = !r^{\bot}\ \mathsf{in}$
$\mathsf{let}\ \mathit{trace}^{\top} = \mathit{succ}^{\bot}\ x^{\bot}\ \mathsf{in}$
$r^{\bot} := \mathit{succ}^{\bot}\ x^{\bot}$

To sum up, while rules (T-LET$^{\top}$) and (T-APP$^{\top}$) stop the expansion of ghost code thanks to the explicit ghost status of bound variables, the purpose of rules (T-LET$^{\bot}$) and (T-APP$^{\bot}$) is to allow ghost code to use regular variables and functions. This was one of motivations for this work.

Finally, it is worth pointing out that there is no sub-typing in our system. That is, in the 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 $\mathsf{int}^{\bot} \overset{\epsilon}{\Longrightarrow} \mathsf{int}$ cannot be applied to an argument of type $\mathsf{int}^{\top} \overset{\epsilon}{\Longrightarrow} \mathsf{int}$.

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 judgements for MiniML terms: $\Gamma \cdot \Sigma \vdash_m t : \tau \cdot \epsilon$.

3.4 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 [8,9], 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 $\mathit{dom}(\mu) \subseteq \mathit{dom}(\Sigma)$ and $\mu(r^{\beta})$ has type $\Sigma(r^{\beta})$ for every $r^{\beta} \in \mathit{dom}(\mu)$.

To prove preservation, we need the following substitution lemma:

Lemma 1 (Substitution Lemma)
Let $[x^{\beta_0} \mapsto \tau_0]\Gamma \cdot \Sigma \vdash t : \tau \cdot \beta_1 \cdot \epsilon$ *and* $\Gamma \cdot \Sigma \vdash v : \tau_0 \cdot \beta_2 \cdot \bot$ *be two derivable judgements such that* $\epsilon^{+}(\tau) \vee \epsilon = \top$ *and* $\beta_0 = \bot$ *implies* $\beta_2 = \bot$*. Then,* $\Gamma \cdot \Sigma \vdash t[x^{\beta_0} \leftarrow v] : \tau \cdot \beta_3 \cdot \epsilon$ *is derivable with*

i) $\beta_0 = \bot \Rightarrow \beta_1 \vee \beta_2 \geq \beta_3$
ii) $\beta_0 = \top \Rightarrow \beta_1 \geq \beta_3$

Proof By induction on the derivation of the statement

$$[x^{\beta_0} \mapsto \tau_0]\Gamma \cdot \Sigma \vdash t : \tau \cdot \beta_1 \cdot \epsilon.$$

The statement is almost a standard version of substitution lemma for variables. The only difference is the conditions we impose on ghost statuses in hypotheses and conclusion. The intuition is that we use this lemma to prove preservation theorem in the cases for let-expression $\mathsf{let}\ x^{\beta} = v\ \mathsf{in}\ t$, or for application $(t\ v)$. In both cases, by assumption, the corresponding redex will be a well-typed, so we will know that the side condition (1) holds for the *union* of effects and ghost statuses of t and v. However, we do not have this information explicitly in the lemma, so we need to strengthen its statement. We detail the proof for some interesting cases.

Case (T-Var): $t = z^{\beta}$ with $([x^{\beta_0} \mapsto \tau_0]\Gamma)(z^{\beta}) = \tau_0'$. There are two subcases to consider, depending on whether z^{β} is equal x^{β_0} or not. If $z^{\beta} \neq x^{\beta_0}$, then $\beta_1 = \beta_3$, so the result follows, whatever β_0 is. If $z^{\beta} = x^{\beta_0}$, then $t[x^{\beta_0} \leftarrow v] = v$ with $\beta_0 = \beta_1$ and $\beta_2 = \beta_3$. So, if $\beta_0 = \bot$, we have $\beta_1 \vee \beta_2 = \beta_2 \geq \beta_2 = \beta_3$; if $\beta_0 = \top$, then $\top = \beta_1 \geq \beta_3$ which holds for any β_3.

Case (T-Assign): $t = r^{\beta_1} := v'$. The typing of t is

$$\frac{\Gamma \cdot \Sigma \vdash v' : \kappa \cdot \beta' \cdot \bot \quad \Sigma(r^{\beta_1}) = \kappa \quad \beta_1 \geq \beta'}{\Gamma \cdot \Sigma \vdash r^{\beta_1} := v' : \mathsf{unit} \cdot \beta_1 \cdot \neg\beta_1}\,(\text{T-Assign})$$

where $\Gamma = \Gamma', x^{\beta_0} : \tau_0$.

Sub-Case $\beta_0 = \top$: by induction hypothesis on the typing subderivation for v', we get $\Gamma' \cdot \Sigma \vdash v'[x^{\beta_0} \leftarrow v] : \kappa \cdot \beta_3' \cdot \bot$ with $\beta' \geq \beta_3'$. Since $\beta_1 \geq \beta'$ by hypothesis, we have by transitivity $\beta_1 \geq \beta_3'$. Thus the typing judgement $\Gamma \cdot \Sigma \vdash (r^{\beta_1} := v')[x^{\beta_0} \leftarrow v] : \mathsf{unit} \cdot \beta_1 \cdot \neg\beta_1$ is derivable.

Sub-Case $\beta_0 = \bot, \beta_1 = \top$: by induction hypothesis on the typing subderivation for v', we get $\Gamma' \cdot \Sigma \vdash v'[x^{\beta_0} \leftarrow v] : \kappa \cdot \beta_3' \cdot \bot$ with $\top = \beta_1 \vee \beta_2 \geq \beta_3'$. Thus the side condition $\top \geq \beta_3'$ is trivially verified and the typing judgement $\Gamma \cdot \Sigma \vdash (r^{\beta_1} := v')[x^{\beta_0} \leftarrow v] : \mathsf{unit} \cdot \beta_1 \cdot \neg\beta_1$ is derivable.

Sub-Case $\beta_0 = \bot, \beta_1 = \bot$: Since $\epsilon = \top$, we can assume that $\beta_2 = \bot$. Since $\beta' = \bot$ as well, the induction hypothesis on the typing subderivation of v' gives

$$\Gamma' \cdot \Sigma \vdash v'[x^{\beta_0} \leftarrow v] : \kappa \cdot \beta_3' \cdot \bot$$

where $\beta_3' = \bot$ since $\bot = \beta' \vee\ \beta_2 \geq \beta_3'$. Thus the side condition $\bot \geq \beta_3'$ is trivially verified and the judgement $\Gamma \cdot \Sigma \vdash (r^{\beta_1} := v')[x^{\beta_0} \leftarrow v] : \mathsf{unit} \cdot \beta_1 \cdot \neg\beta_1$ is derivable.

Case (T-If): $t = \mathsf{if}\ v'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$. The typing of t is

$$\frac{\Gamma \cdot \Sigma \vdash v' : \mathsf{bool} \cdot \beta_{10} \cdot \bot \quad \Gamma \cdot \Sigma \vdash t_1 : \tau \cdot \beta_{11} \cdot \epsilon_1 \quad \Gamma \cdot \Sigma \vdash t_2 : \tau \cdot \beta_{12} \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash \mathsf{if}\ v'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \tau \cdot \beta_{10}\ \vee\ \beta_{11}\ \vee\ \beta_{12} \cdot \epsilon_1\ \vee\ \epsilon_2}$$

where $\Gamma = \Gamma', x^{\beta_0} : \tau_0$.

Sub-Case $\beta_0 = \top$: by induction hypotheses on the subderivations, we obtain the following typing judgements:

$\Gamma' \cdot \Sigma \vdash v'[x^{\beta_0} \leftarrow v] : \mathsf{bool} \cdot \beta_{30} \cdot \bot$ with $\beta_{10} \geq \beta_{30}$
$\Gamma' \cdot \Sigma \vdash t_1[x^{\beta_0} \leftarrow v] : \tau \cdot \beta_{31} \cdot \epsilon_1$ with $\beta_{11} \geq \beta_{31}$
$\Gamma' \cdot \Sigma \vdash t_2[x^{\beta_0} \leftarrow v] : \tau \cdot \beta_{32} \cdot \epsilon_2$ with $\beta_{12} \geq \beta_{32}$

Thus, the typing judgement

$\Gamma' \cdot \Sigma \vdash (\mathsf{if}\ v'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)[x^{\beta_0} \leftarrow v] : \tau \cdot \beta_{30}\ \vee\ \beta_{31}\ \vee\ \beta_{32} \cdot \epsilon_1\ \vee\ \epsilon_2$

is derivable with $\beta_1 \geq \beta_3 = \beta_{30} \vee\ \beta_{31} \vee\ \beta_{32}$.

Sub-Case $\beta_0 = \bot$: if $\epsilon^+ \vee \epsilon_1 \vee \epsilon_2 = \top$, then we may assume that $\beta_2 = \bot$ and the subcase is proved in the same way; if $\epsilon^+ \vee \epsilon_1 \vee \epsilon_2 = \bot$, then by induction hypotheses on the typing subderivations, we obtain

$\Gamma' \cdot \Sigma \vdash v'[x^{\beta_0} \leftarrow v] : \mathsf{bool} \cdot \beta_{30} \cdot \bot$ with $\beta_{10} \vee \beta_2 \geq \beta_{30}$

$\Gamma' \cdot \Sigma \vdash t_1[x^{\beta_0} \leftarrow v] : \tau \cdot \beta_{31} \cdot \epsilon_1$ with $\beta_{11} \vee \beta_2 \geq \beta_{31}$

$\Gamma' \cdot \Sigma \vdash t_2[x^{\beta_0} \leftarrow v] : \tau \cdot \beta_{32} \cdot \epsilon_2$ with $\beta_{12} \vee \beta_2 \geq \beta_{32}$

Thus, the typing judgement

$\Gamma' \cdot \Sigma \vdash (\mathsf{if}\ v'\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)[x^{\beta_0} \leftarrow v] : \tau \cdot \beta_{30} \vee \beta_{31} \vee \beta_{32} \cdot \epsilon_1 \vee \epsilon_2$

is derivable with $\beta_1 \vee \beta_2 \geq \beta_3 = \beta_{30} \vee \beta_{31} \vee \beta_{32}$.

The other cases are proved similarly (with standard auxiliary weakening and permutation lemmas in the cases for λ-abstractions and `let-in` expressions). □

Lemma 2 (Preservation) *If* $\Gamma \cdot \Sigma \vdash t : \tau \cdot \beta \cdot \epsilon$ *and* $\Sigma \vdash \mu$*, then* $\mu \cdot t \to \mu' \cdot t'$ *implies that* $\Gamma \cdot \Sigma \vdash t' : \tau \cdot \beta' \cdot \epsilon'$ *and* $\Sigma \vdash \mu'$*, where* $\beta \geq \beta'$ *and* $\epsilon \geq \epsilon'$*.*

Proof By induction on the derivation of $\Gamma \cdot \Sigma \vdash t : \tau \cdot \beta \cdot \epsilon$, using lemmas above. The only difference with respect to the standard statement is that ghost statuses and effect indicators can decrease during evaluation. At each step of the induction, we assume that the desired result holds for all subderivations and proceed by case analysis on the last rule used in the derivation. The four cases where t is a value cannot happen.

Case (T-IF): $t = \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$ with typing judgement

$$\frac{\Gamma \cdot \Sigma \vdash v : \mathsf{bool} \cdot \beta_0 \cdot \bot \quad \Gamma \cdot \Sigma \vdash t_1 : \tau \cdot \beta_1 \cdot \epsilon_1 \quad \Gamma \cdot \Sigma \vdash t_2 : \tau \cdot \beta_2 \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \tau \cdot \beta_0 \vee \beta_1 \vee \beta_2 \cdot \epsilon_1 \vee \epsilon_2}$$

There are two rules by which $\mu \cdot t \to \mu' \cdot t'$ can be derived: (E-IF-TRUE) and (E-IF-FALSE).

Subcase (E-IF-TRUE): $\mu \cdot t \to \mu \cdot t_1$. As can be seen in the typing subderivation of t_1 above, the desired property holds, since $\beta_0 \vee \beta_1 \vee \beta_2 \geq \beta_1$ and $\epsilon_1 \vee \epsilon_2 \geq \epsilon_1$.

Subcase (E-IF-FALSE): $\mu \cdot t \to \mu \cdot t_2$. Identical to the previous subcase.

Case (T-LET$^\bot$): $t = \mathsf{let}\ x^\bot = t_2\ \mathsf{in}\ t_1$. The typing judgement is

$$\frac{[x^\bot \mapsto \tau_2]\Gamma \cdot \Sigma \vdash t_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \quad \Gamma \cdot \Sigma \vdash t_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash \mathsf{let}\ x^\bot = t_2\ \mathsf{in}\ t_1 : \tau_1 \cdot \beta_1 \vee \beta_2 \cdot \epsilon_1 \vee \epsilon_2}$$

There are two rules by which $\mu \cdot t \to \mu' \cdot t'$ can be derived: (E-CONTEXT-LET) and (E-LET).

Subcase (E-LET): $t_2 = v_2$ and $\mu \cdot \mathsf{let}\ x^\bot = v_2\ \mathsf{in}\ t_1 \to \mu \cdot t_1[x^\bot \leftarrow v_2]$. By substitution lemma 1, we have $\Gamma \cdot \Sigma \vdash t_1[x^\bot \leftarrow v_2] : \tau_1 \cdot \beta_3 \cdot \epsilon_1$ where $\beta_1 \vee \beta_2 \geq \beta_3$.

Subcase (E-CONTEXT-LET): $\mu \cdot \mathsf{let}\ x^\bot = t_2\ \mathsf{in}\ t_1 \to \mu' \cdot \mathsf{let}\ x^\bot = t_2'\ \mathsf{in}\ t_1$ with subderivation $\mu \cdot t_2 \to \mu' \cdot t_2'$. The typing judgement for t is

$$\frac{[x^\bot \mapsto \tau_2]\Gamma \cdot \Sigma \vdash t_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \quad \Gamma \cdot \Sigma \vdash t_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash \mathsf{let}\ x^\bot = t_2\ \mathsf{in}\ t_1 : \tau_1 \cdot \beta_1 \vee \beta_2 \cdot \epsilon_1 \vee \epsilon_2}$$

By induction hypothesis on $\Gamma \cdot \Sigma \vdash t_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2$ we have $\Gamma \cdot \Sigma \vdash t'_2 : \tau_2 \cdot \beta'_2 \cdot \epsilon'_2$ with $\epsilon_2 \geq \epsilon'_2$ $\beta_2 \geq \beta'_2$. So we can derive

$$\frac{[x^{\bot} \mapsto \tau_2]\Gamma \cdot \Sigma \vdash t_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \quad \Gamma \cdot \Sigma \vdash t'_2 : \tau_2 \cdot \beta'_2 \cdot \epsilon'_2}{\Gamma \cdot \Sigma \vdash \mathsf{let}\ x^{\bot} = t'_2\ \mathsf{in}\ t_1 : \tau_1 \cdot \beta_1 \vee \beta'_2 \cdot \epsilon_1 \vee \epsilon'_2}$$

with $(\epsilon_1 \vee \epsilon_2) \geq (\epsilon_1 \vee \epsilon'_2)$ and $(\beta_1 \vee \beta_2) \geq (\beta_1 \vee \beta'_2)$.

Case (T-Let$^{\top}$): $t = \mathsf{let}\ x^{\top} = t_2\ \mathsf{in}\ t_1$. There are two rules by which $\mu \cdot t \ \rightarrow\ \mu' \cdot t'$ can be derived: (E-Let) and (E-Context-Let).

Subcase (E-Let): $t_2 = v_2$ and $\mu \cdot \mathsf{let}\ x^{\top} = v_2\ \mathsf{in}\ t_1 \ \rightarrow\ \mu \cdot t_1[x^{\top} \leftarrow v_2]$ The result follows from the substitution lemma 1.

Subcase (E-Context-Let): $\mu \cdot \mathsf{let}\ x^{\top} = t_2\ \mathsf{in}\ t_1 \ \rightarrow\ \mu' \cdot \mathsf{let}\ x^{\top} = t'_2\ \mathsf{in}\ t_1$ with subderivation $\mu \cdot t_2 \ \rightarrow\ \mu' \cdot t'_2$. The typing judgement for t is

$$\frac{[x^{\top} \mapsto \tau_2]\Gamma \cdot \Sigma \vdash t_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \quad \Gamma \cdot \Sigma \vdash t_2 : \tau_2 \cdot \beta_2 \cdot \bot}{\Gamma \cdot \Sigma \vdash \mathsf{let}\ x^{\top} = t_2\ \mathsf{in}\ t_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1}$$

By induction hypothesis on $\Gamma \cdot \Sigma \vdash t_2 : \tau_2 \cdot \beta_2 \cdot \bot$ we have $\Gamma \cdot \Sigma \vdash t'_2 : \tau_2 \cdot \beta'_2 \cdot \bot$ with $\beta_2 \geq \beta'_2$. So we can derive

$$\frac{[x^{\top} \mapsto \tau_2]\Gamma \cdot \Sigma \vdash t_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \quad \Gamma \cdot \Sigma \vdash t'_2 : \tau_2 \cdot \beta'_2 \cdot \bot}{\Gamma \cdot \Sigma \vdash \mathsf{let}\ x^{\top} = t'_2\ \mathsf{in}\ t_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1}$$

Case (T-App$^{\bot}$): $t = t_1\ v_2$ with $\Gamma \cdot \Sigma \vdash t_1 : \tau_2^{\bot} \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \beta_1 \cdot \epsilon_2$. Similar to the case (T-Let$^{\bot}$).

Case (T-App$^{\top}$): $t = t_1\ v_2$ with $\Gamma \cdot \Sigma \vdash t_1 : \tau_2^{\top} \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \beta_1 \cdot \epsilon_2$. Similar to the case (T-Let$^{\top}$).

Case (T-Deref): $t = !r^{\beta}$ with $\mu \cdot !r^{\beta} \ \rightarrow\ \mu \cdot \mu(r^{\beta})$. We have the following typing for t:

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

By hypothesis, $\Sigma \vdash \mu$, so $\mu(r^{\beta})$ has type κ. Moreover, $\mu(r^{\beta})$ is necessarily a constant. Therefore we can conclude by deriving the typing judgement for $\mu(r^{\beta})$:

$$\frac{\mathsf{Typeof}(\mu(r^{\beta})) = \kappa}{\Gamma \cdot \Sigma \vdash \mu(r^{\beta}) : \kappa \cdot \bot \cdot \bot}\,(\text{T-Const})$$

Case (T-Assign): $t = r^{\beta} := v$. The typing rule for t is

$$\frac{\Gamma \cdot \Sigma \vdash v : \kappa \cdot \beta' \cdot \bot \quad \Sigma(r^{\beta}) = \kappa \quad \beta \geq \beta'}{\Gamma \cdot \Sigma \vdash r^{\beta} := v : \mathsf{unit} \cdot \beta \cdot \neg\beta}$$

The only rule that allows us to derive $\mu \cdot t \ \rightarrow\ \mu' \cdot t'$ is (E-Assign). That is v is necessarily a constant and the derivation is $\mu \cdot r^{\beta} := c \ \rightarrow\ \mu[r^{\beta} \mapsto c] \cdot ()$. It remains then to check that $\Sigma \vdash \mu[r^{\beta} \mapsto c]$, which is true, since we have $\mu \backslash \{r^{\beta}\} = \mu' \backslash \{r^{\beta}\}$ and $c = \mu'(r^{\beta})$ has type $\kappa = \Sigma(r^{\beta})$.

Case (T-Ghost): $t = \mathsf{ghost}\ t_1$ with $\Gamma \cdot \Sigma \vdash t_1 : \tau \cdot \beta \cdot \bot$. The result follows immediately. □

Lemma 3 (Progress) *If $\varnothing \cdot \Sigma \vdash t : \tau \cdot \beta \cdot \epsilon$, then either t is a value or, for any store μ such that $\Sigma \vdash \mu$, there exists a reduction step $\mu \cdot t \ \rightarrow \ \mu' \cdot t'$.*

Proof Straightforward induction on the typing derivation of t. □

Additionally, we have the following results for effectless programs.

Lemma 4 (Store Preservation) *Let $\mu_\bot$ denote the regular part of store μ, that is the restriction of μ to regular references. Then, If $\varnothing \cdot \Sigma \vdash t : \tau \cdot \beta \cdot \bot$ and $\Sigma \vdash \mu$, then $\mu \cdot t \ \rightarrow \ \mu' \cdot t'$ implies $\mu_\bot = \mu'_\bot$.*

Proof By induction on the derivation of $\mu \cdot t \ \rightarrow \ \mu' \cdot t'$.
Case (E-Assign): $t = r^\beta := c$ with $\qquad \mu \cdot r^\beta := c \ \rightarrow \ \mu[r^\beta \mapsto c] \cdot ()$.

By hypothesis, $\epsilon = \bot$, so in the typing of t, $\beta = \top$:

$$\frac{\Gamma \cdot \Sigma \vdash v : \kappa \cdot \beta' \cdot \bot \quad \Sigma(r^\beta) = \kappa \qquad \beta \geq \beta'}{\Gamma \cdot \Sigma \vdash (r^\beta := v) : \mathsf{unit} \cdot \beta \cdot \neg\beta}$$

that is r^β is necessarily a ghost reference, so the regular part of the store remains the same.

Case (E-Context-App): $\mu \cdot t_1 \ v \ \rightarrow \ \mu' \cdot t'_1 \ v$. As t is well-typed by hypothesis, t_1 is well-typed too with the typing

$$\Gamma \cdot \Sigma \vdash t_1 : \tau^{\beta''} \overset{\epsilon_2}{\Longrightarrow} \tau_2 \cdot \beta' \cdot \epsilon_1$$

for some β', β'' that depend on typing rule for t (either (T-App$^\bot$) or (T-App$^\top$)). In both cases, by hypothesis, we have $\bot = \epsilon_1 \vee \epsilon_2$. As $\epsilon_1 = \bot$, we can apply the induction hypothesis on the sub-derivation $\mu \cdot t_1 \ \rightarrow \ \mu' \cdot t'_1$, which gives the desired result $\mu_\bot = \mu'_\bot$

Case (E-Context-Let). Identical to the previous case.

Other cases are trivial since the store is not modified by reduction. □

Lemma 5 (Program Termination) *If $\varnothing \cdot \Sigma \vdash t : \tau \cdot \beta \cdot \bot$, $\Sigma \vdash \mu$, then evaluation of t in store μ terminates, that is, there is a value v and a store μ' such that $\mu \cdot t \ \rightarrow^\star \ \mu' \cdot v$.*

Proof Let us assume first that the reduction of t does not involve any call to a recursive function. Since the set of global references is finite and each global reference r_i stores a value of a primitive type κ_i (whatever its ghost status), the program t can be translated into a simply-typed λ-term, using the state monad $M \ \tau \triangleq \kappa_1 \times \cdots \times \kappa_n \rightarrow \tau \times \kappa_1 \times \cdots \times \kappa_n$. The result follows by normalization of simply-typed λ-calculus.

Now, let us assume we have an infinite reduction from t involving calls to recursive functions. Since t has effect $\bot$, it only calls recursive functions for which the oracle (presumed to be sound) says they terminate. Thus we can replace any finite sub-reduction corresponding to a recursive function call by $(\lambda x.x) \ v$ where v is the result value. We still have an infinite reduction, which contradicts the first part of the proof. □

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

4 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:

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

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 $\Gamma \cdot \Sigma \vdash t : \tau \cdot \beta \cdot \epsilon$ holds. The erasure $\mathcal{E}_{\beta}(t)$ is defined by induction on the structure of t as follows:

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

Note that ghosts variables and ghost references do 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 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 unnecessary. However, because of latent effects, full erasure of ghost code is not possible. Consider for instance the function

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

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. These reductions facilitate the proofs of forthcoming theorems. Notice that the program after erasure is semantically equivalent to the original one, with only extra administrative reduction steps (a term coined by Plotkin [10]) that do not impact the program complexity and can be eliminated at compile time.

4.1 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)

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

With these definitions, we have immediately that $\mathcal{E}([x^{\top} \mapsto \tau]\Gamma) = [x \mapsto \mathsf{unit}]\mathcal{E}(\Gamma)$ and $\mathcal{E}([x^{\bot} \mapsto \tau]\Gamma) = [x \mapsto \mathcal{E}_{\bot}(\tau)]\mathcal{E}(\Gamma)$. Now we can prove well-typedness preservation under erasure:

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

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

Case (T-λ):
$$\frac{[x^{\beta} \mapsto \tau]\Gamma \cdot \Sigma \vdash t_1 : \tau_1 \cdot \bot \cdot \epsilon}{\Gamma \cdot \Sigma \vdash (\lambda x^{\beta} : \tau . t_1) : \tau^{\beta} \overset{\epsilon}{\Longrightarrow} \tau_1 \cdot \bot \cdot \bot}$$

By induction hypothesis, we have $[x \mapsto \mathcal{E}_{\beta}(\tau)]\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_{\bot}(t_1) : \mathcal{E}_{\bot}(\tau_1) \cdot \epsilon$. Therefore, we conclude that $\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \lambda x : \mathcal{E}_{\beta}(\tau). \mathcal{E}_{\bot}(t) : \mathcal{E}_{\beta}(\tau) \overset{\epsilon}{\Longrightarrow} \mathcal{E}_{\bot}(\tau_1) \cdot \bot$.

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

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

$$\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_{\bot}(t) : \mathcal{E}_{\bot}(\tau_2) \overset{\epsilon_1}{\Longrightarrow} \mathcal{E}_{\bot}(\tau_1) \cdot \epsilon_2$$
$$\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_{\bot}(v) : \mathcal{E}_{\bot}(\tau_2) \cdot \bot.$$

Thus, $\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_{\bot}(t)\ \mathcal{E}_{\bot}(v) : \mathcal{E}_{\bot}(\tau_1) \cdot \epsilon_1 \vee \epsilon_2$.

Case (T-App$^{\top}$):
$$\frac{\Gamma \cdot \Sigma \vdash t : \tau_2^{\top} \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \beta_1 \cdot \epsilon_2 \quad \Gamma \cdot \Sigma \vdash v : \tau_2 \cdot \beta_2 \cdot \bot}{\Gamma \cdot \Sigma \vdash (t\ v) : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \vee \epsilon_2}$$

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

$$\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_{\bot}(t) : \mathsf{unit} \overset{\epsilon_1}{\Longrightarrow} \mathcal{E}_{\bot}(\tau_1) \cdot \epsilon_2$$

As $\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m () : \mathsf{unit} \cdot \bot$, we conclude that $\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_{\bot}(t)\ () : \mathcal{E}_{\bot}(\tau_1) \cdot \epsilon_1 \vee \epsilon_2$.

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

By hypothesis $\beta = \bot$, and by premise side condition $\beta \geq \beta'$, so $\beta' = \bot$ as well. By induction hypothesis on the rule premise, we obtain $\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_{\bot}(v) : \kappa \cdot \bot$. Also, $\Sigma(r^{\bot}) = \kappa$, so $(\mathcal{E}(\Sigma))(r) = \kappa$. Therefore we conclude with

$$\frac{\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_{\bot}(v) : \kappa \cdot \bot \qquad (\mathcal{E}(\Sigma))(r) = \kappa}{\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m r := \mathcal{E}_{\bot}(v) : \mathsf{unit} \cdot \top}$$

Other cases are easily proved in a similar way. □

4.2 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)**

$$\begin{array}{lcl} \mathcal{E}(\varnothing) & \triangleq & \varnothing \\ \mathcal{E}(\mu \uplus \{r^{\top} \mapsto c\}) & \triangleq & \mathcal{E}(\mu) \\ \mathcal{E}(\mu \uplus \{r^{\bot} \mapsto c\}) & \triangleq & \mathcal{E}(\mu) \uplus \{r \mapsto c\} \end{array}$$

Notice that $\mathcal{E}(\mu)$ removes ghost annotations, and thus is not the same that $\mu_{\bot}$. The correctness of erasure means that, for any evaluation $\mu \cdot t \ \rightarrow^{\star} \ \mu' \cdot v$ in GhostML, we have $\mathcal{E}(\mu) \cdot \mathcal{E}_{\bot}(t) \ \rightarrow^{\star}_{m} \ \mathcal{E}(\mu') \cdot \mathcal{E}_{\bot}(v)$ in MiniML and that, for any diverging evaluation $\mu \cdot t \ \rightarrow \ \infty$ in GhostML, we have $\mathcal{E}(\mu) \cdot \mathcal{E}_{\bot}(t) \ \rightarrow_{m} \ \infty$ 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 6 **(Substitution Under Erasure)** *Let t be a GhostML term and v a GhostML value such that $[x^{\beta} \mapsto \tau]\Gamma \cdot \Sigma \vdash t : \tau_0 \cdot \bot \cdot \epsilon$ and $\Gamma \cdot \Sigma \vdash v : \tau \cdot \beta' \cdot \bot$, with $\beta \geq \beta'$, hold. Then the following holds:*

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

Proof Straightforward induction on the structure of t. □

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 7 **(Forward Simulation of GhostML)** *If $\varnothing \cdot \Sigma \vdash t : \tau \cdot \bot \cdot \epsilon$ and, for some store μ such that $\Sigma \vdash \mu$, we have $\mu \cdot t \ \rightarrow \ \mu' \cdot t'$, then the following holds in MiniML:*

$$\mathcal{E}(\mu) \cdot \mathcal{E}_{\bot}(t) \ \rightarrow^{0|1}_{m} \ \mathcal{E}(\mu') \cdot \mathcal{E}_{\bot}(t').$$

Proof By induction on the derivation $\rightarrow$.
Case (E-App-λ): $(\lambda x^{\beta} : \tau_2 . t) \ v \ \rightarrow \ t[x^{\beta} \leftarrow v]$. If $\beta = \top$, then

$$\frac{\Gamma \cdot \Sigma \vdash \lambda x^{\top} : \tau_2 . t : \tau_2^{\top} \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \bot \cdot \bot \qquad \Gamma \cdot \Sigma \vdash v : \tau_2 \cdot \beta_1 \cdot \bot}{\Gamma \cdot \Sigma \vdash (\lambda x^{\top} : \tau_2 . t \ v) : \tau_1 \cdot \bot \cdot \epsilon_1}$$

Therefore, by substitution under erasure (Lemma 6), we get

$$\mathcal{E}_{\bot}((\lambda x^{\top} : \tau_2 . t) \ v) = (\lambda x : \mathsf{unit} . \mathcal{E}_{\bot}(t)) \ ()) \rightarrow_{m} \mathcal{E}_{\bot}(t)[x \leftarrow ()] = \mathcal{E}_{\bot}(t[x^{\top} \leftarrow v]).$$

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

$$\frac{\Gamma \cdot \Sigma \vdash \lambda x^{\bot} : \tau_2 . t : \tau_2^{\bot} \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \bot \cdot \bot \qquad \Gamma \cdot \Sigma \vdash v : \tau_2 \cdot \bot \cdot \bot}{\Gamma \cdot \Sigma \vdash (\lambda x^{\bot} : \tau_2 . t \ v) : \tau_1 \cdot \bot \cdot \epsilon_1}$$

Again, by substitution under erasure (Lemma 6), we get

$$\mathcal{E}_{\bot}((\lambda x^{\bot} : \tau_2 . t) \ v) \rightarrow_{m} \mathcal{E}_{\bot}(t)[x \leftarrow \mathcal{E}_{\bot}(v)] = \mathcal{E}_{\bot}(t[x^{\bot} \leftarrow v]).$$

Case (E-Context-Let): $\dfrac{\mu\cdot t_2 \;\to\; \mu'\cdot t_2'}{\mu\cdot \mathsf{let}\ x^{\beta} = t_2\ \mathsf{in}\ t_1 \;\to\; \mu'\cdot \mathsf{let}\ x^{\beta} = t_2'\ \mathsf{in}\ t_1}$

If $\beta = \top$, then $\dfrac{[x^{\top} \mapsto \tau_2]\Gamma\cdot\Sigma \vdash t_1 : \tau_1\cdot\bot\cdot\epsilon_1 \quad \Gamma\cdot\Sigma\vdash t_2 : \tau_2\cdot\beta_2\cdot\bot}{\Gamma\cdot\Sigma\vdash \mathsf{let}\ x^{\top} = t_2\ \mathsf{in}\ t_1 : \tau_1\cdot\bot\cdot\epsilon_1}.$

As t_2 is an effectless term, we have by store preservation (Lemma 4), $\mu_\bot = \mu'_\bot$. Thus we have $\mathcal{E}(\mu) = \mathcal{E}(\mu')$. Therefore,

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

If $\beta = \bot$, then $\dfrac{[x^{\bot} \mapsto \tau_2]\Gamma\cdot\Sigma \vdash t_1 : \tau_1\cdot\bot\cdot\epsilon_1 \quad \Gamma\cdot\Sigma\vdash t_2 : \tau_2\cdot\bot\cdot\epsilon_2}{\Gamma\cdot\Sigma\vdash \mathsf{let}\ x^{\bot} = t_2\ \mathsf{in}\ t_1 : \tau_1\cdot\bot\cdot\epsilon_1\vee\epsilon_2}.$

By induction hypothesis on sub-derivation $\mu\cdot t_2 \;\to\; \mu'\cdot t_2'$, we get

$$\mathcal{E}(\mu)\cdot\mathcal{E}_\bot(t_2) \;\to_m^{0|1}\; \mathcal{E}(\mu')\cdot\mathcal{E}_\bot(t_2').$$

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

$$\dfrac{\mathcal{E}(\mu)\cdot\mathcal{E}_\bot(t_2) \;\to_m\; \mathcal{E}(\mu')\cdot\mathcal{E}_\bot(t_2')}{\mathcal{E}(\mu)\cdot\mathsf{let}\ x = \mathcal{E}_\bot(t_2)\ \mathsf{in}\ \mathcal{E}_\bot(t_1) \;\to_m\; \mathcal{E}(\mu')\cdot\mathsf{let}\ x = \mathcal{E}_\bot(t_2')\ \mathsf{in}\ \mathcal{E}_\bot(t_1)}$$

Other cases are proved in a similar way.

□

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

Theorem 2 (Terminating Evaluation Preservation) *If $\varnothing\cdot\Sigma\vdash t : \tau\cdot\bot\cdot\epsilon$ holds and $\mu\cdot t \;\to^\star\; \mu'\cdot v$, for some value v and some store μ such that $\Sigma\vdash\mu$, then $\mathcal{E}(\mu)\cdot\mathcal{E}_\bot(t) \;\to_m^\star\; \mathcal{E}(\mu')\cdot\mathcal{E}_\bot(v)$.*

Proof By induction on the length of the evaluation $\mu\cdot t \;\to^\star\; \mu'\cdot v$.
If $\mu\cdot t \;\to^0\; \mu'\cdot v$, then the result trivially holds since $t = v$ and $\mu' = \mu$. Now, assume that $\mu\cdot t \;\to^1\; \mu''\cdot t'' \;\to^n\; \mu'\cdot v$ for some intermediate store μ''. By preservation (Lemma 2), $\Gamma\cdot\Sigma\vdash t'' : \tau\cdot\bot\cdot\epsilon''$ and $\Sigma\vdash\mu''$ for some ϵ'', such that $\epsilon' \geq \epsilon''$. By induction hypothesis on t'', we obtain $\mathcal{E}(\mu'')\cdot\mathcal{E}_\bot(t'') \;\to_m^\star\; \mathcal{E}(\mu')\cdot\mathcal{E}_\bot(v)$. By forward simulation (Lemma 7), it follows that $\mathcal{E}(\mu)\cdot\mathcal{E}_\bot(t) \;\to_m^{0|1}\; \mathcal{E}(\mu'')\cdot\mathcal{E}_\bot(t'')$. Putting pieces together, we conclude that $\mathcal{E}(\mu)\cdot\mathcal{E}_\bot(t) \;\to_m^{0|1}\; \mathcal{E}(\mu'')\cdot\mathcal{E}_\bot(t'') \;\to_m^\star\; \mathcal{E}(\mu')\cdot\mathcal{E}_\bot(v)$. □

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

Lemma 8 (Forward Simulation of MiniML) *If $\varnothing\cdot\Sigma\vdash t : \tau\cdot\bot\cdot\epsilon$ holds, then, for any store μ such that $\Sigma\vdash\mu$, if $\mathcal{E}(\mu)\cdot\mathcal{E}_\bot(t) \;\to_m\; \nu\cdot q$ for some term q and some store ν, then $\mu\cdot t \;\to^{\geq 1}\; \mu'\cdot t'$ 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 the GhostML progress property (Lemma 3), it can be reduced. By hypothesis, t is well-typed, with a typing judgement for t_2 being $\varnothing\cdot\Sigma\vdash t_2 : \tau\cdot\beta\cdot\bot$. By program termination (Lemma 5), we have $\mu\cdot t_2 \;\to^\star\; \mu'\cdot v$ for some v and μ'. Consequently,

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

Therefore, by forward simulation in GhostML (Lemma 7), we have

$$\mathcal{E}(\mu)\cdot\mathcal{E}_{\perp}(t)\ \rightarrow^{0}_{m}\ \mathcal{E}(\mu)\cdot\mathcal{E}_{\perp}(\mathsf{let}\ x^{\top}=v\ \mathsf{in}\ t_1)\ \rightarrow_{m}\ \mathcal{E}(\mu)\cdot\mathcal{E}_{\perp}(t_1)[x\leftarrow()].$$

Finally, by store preservation (Lemma 4), $\mu_{\perp}=\mu'_{\perp}$. Therefore, $\mathcal{E}(\mu)=\mathcal{E}(\mu')$, which allows us to conclude.

Case 2: t is $\mathsf{let}\ x^{\perp}=t_2\ \mathsf{in}\ t_1$. 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 non-termination preservation:

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

Proof By co-induction, it suffices to prove that there exist t' and μ' such that the typing judgement $\varnothing\cdot\Sigma\vdash t':\tau\cdot\perp\cdot\epsilon'$ and $\Sigma\vdash\mu'$, and

$$\mathcal{E}(\mu)\cdot\mathcal{E}_{\perp}(t)\rightarrow^{1}_{m}\mathcal{E}(\mu')\cdot\mathcal{E}_{\perp}(t')\ \wedge\ \mu'\cdot t'\ \rightarrow\ \infty.$$

Observe that, since t diverges, t is not a value. Thus $\mathcal{E}_{\perp}(t)$ is not a value either. By well-typedness preservation (Theorem 1), $\mathcal{E}_{\perp}(t)$ is well-typed, with

$$\varnothing\cdot\mathcal{E}(\Sigma)\vdash_{m}\mathcal{E}_{\perp}(t):\mathcal{E}_{\perp}(\tau)\cdot\epsilon.$$

We have $\mathcal{E}(\Sigma)\vdash_{m}\mathcal{E}(\mu)$, since $\Sigma\vdash\mu$. Therefore, by progress in MiniML, there exist some term q and some store ν such that $\mathcal{E}(\mu)\cdot\mathcal{E}_{\perp}(t)\ \rightarrow_{m}\ \nu\cdot q$. By simulation (Lemma 8), $\mu\cdot t\ \rightarrow^{\geq 1}\ \mu'\cdot t'$ with $q=\mathcal{E}_{\perp}(t')$ and $\nu=\mathcal{E}(\mu')$. The reduction being deterministic, t' diverges and we have the following reductions:

$$\begin{tikzcd}
\mu\cdot t \arrow[r, "\geq 1"] \arrow[d, Rightarrow] & \mu'\cdot t' \arrow[r] \arrow[d, Rightarrow] & \infty \\
\mathcal{E}(\mu)\cdot\mathcal{E}_{\perp}(t) \arrow[r, "{}^{1}_{m}"] & \nu\cdot q &
\end{tikzcd}$$

By preservation (Lemma 2), $\Gamma\cdot\Sigma\vdash t':\tau\cdot\perp\cdot\epsilon'$, with $\epsilon\geq\epsilon'$, and $\Sigma\vdash\mu'$. Now, by co-induction hypothesis, $\nu\cdot q\ \rightarrow_{m}\ \infty$ and thus $\mathcal{E}(\mu)\cdot\mathcal{E}_{\perp}(t)\ \rightarrow_{m}\ \infty$. □

5 Implementation

Our method to handle ghost code is implemented in the verification tool Why3[2]. With respect to GhostML, the language and the type system of Why3 have the following extensions:

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

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.

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 needed alterations to the type system of GhostML are minimal. 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 [5], and thus non-interference is ensured purely by type checking.

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. As in GhostML, Why3 allows the use of recursive functions in ghost code. Loops can be used in ghost code as well. 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. 5 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)` is ghost, it must not diverge. Thus Why3 requires function `append` to be terminating.

```
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. 5 Queue implementation in Why3.

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[3], in particular, ghost function parameters and ghost functions to supply automatic induction proofs (also known as lemma functions).

6 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 [11] and Reynolds [12], the notion of auxiliary variable was first introduced by Lucas in 1968 [13]. Since then, numerous authors have adapted this technique in various domains.

It is worth noting that some authors, in particular, Reynolds [12] and Kleymann [14], 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.* [15] 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 [16] proposes a rigorous description of ghost code for a large fragment of C with parallelism, in the context of the VCC verification tool [3]. 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 [2]. 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." [17]. 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 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.

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

The property of non-interference of ghost code is a special case of information flow non-interference [18]. 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 [19] 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 [20] 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.

Erasure of ghost code is similar to the mechanism of program extraction in the Coq proof assistant [21,22]. Indeed, the purpose of such a mechanism is to remove from the program what is computationally irrelevant. For instance, in a proof of $\exists x. P(x)$, that is a witness t and a proof of $P(t)$, only t is kept in the extracted program. To do this, the Coq extraction is guided by sorts, namely Set for programs and Prop for proofs, and non-interference is ensured by Coq type-checking. Coq users must choose between Set and Prop beforehand. For instance, the type nat of natural numbers has sort Set and therefore natural numbers are never removed during extraction, even when they are only used for the purpose of specification and/or proof. In our approach, the same type of natural numbers can be used for both regular and ghost code and is kept only when relevant.

7 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. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The spirit of ghost code. In Biere, A., Bloem, R., eds.: 26th International Conference on Computer Aided Verification. Volume 8859 of Lecture Notes in Computer Science., Vienna, Austria, Springer (July 2014) 1–16
2. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR-16. Volume 6355 of Lecture Notes in Computer Science., Springer (2010) 348–370
3. 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)
4. Jacobs, B., Piessens, F.: The VeriFast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven (August 2008)
5. 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
6. Hatcliff, J., Leavens, G.T., Leino, K.R.M., Müller, P., Parkinson, M.: Behavioral interface specification languages. Technical report CS-TR-09-01, University of Central Florida, School of EECS (2009) A survey paper, Draft.
7. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. SIGPLAN Not. **28**(6) (June 1993) 237–247
8. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115** (1992) 38–94
9. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
10. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2) (1975) 125–159
11. Jones, C.B., Roscoe, A., Wood, K.R.: Reflections on the Work of C.A.R. Hoare. 1st edn. Springer Publishing Company, Incorporated (2010)
12. Reynolds, J.C.: The craft of programming. Prentice Hall International series in computer science. Prentice Hall (1981)
13. Lucas, P.: Two constructive realizations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory, Vienna (June 1968)
14. Kleymann, T.: Hoare logic and auxiliary variables. Formal Asp. Comput. **11**(5) (1999) 541–566
15. 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
16. Schmaltz, S.: Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C. PhD thesis, Saarland University, Saarbrücken (2013)
17. Leino, K.R.M., Moskal, M.: Co-induction simply: Automatic co-inductive proofs in a program verifier. In Jones, C., Pihlajasaari, P., Sun, J., eds.: FM 2014: Formal Methods. Volume 8442 of Lecture Notes in Computer Science., Springer (May 2014) 382–398
18. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM **20**(2) (July 1977) 504–513
19. 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
20. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Transactions on Programming Languages and Systems **25**(1) (January 2003) 117–158 ©ACM.
21. Paulin-Mohring, C.: Extracting F_ω's programs from proofs in the Calculus of Constructions. In: Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, ACM Press (January 1989)
22. Paulin-Mohring, C.: Extraction de programmes dans le Calcul des Constructions. Thèse d'université, Paris 7 (January 1989)