The Spirit of Ghost Code

Jean-Christophe Filliâtre, Léon Gondelman et Andrei Paskevich

LRI, Univ Paris Sud

Journée annuelle du groupe LTP du GDR GPL
LaBRI, Université Bordeaux 1
18 novembre, 2013

Deductive Software Verification

1. specify a program
2. compute verification conditions (VCs)
3. discharge them, using
   - automated theorems provers (ATPs)
   - interactive proof assistants

Example

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

```
let fib n
= aux 0 1 n
```

Specification of fib

```
let rec aux (a b n: int) : int


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


let fib (n: int) : int
  requires {0 ≤ n}
  ensures  {result = F_n}
= aux 0 1 n
```

Specification of aux

```
let rec aux (a b n: int) : int
  requires {0 ≤ n}
  requires {∃k. 0 ≤ k ∧ a = F_k ∧ b = F_{k+1}}
  ensures  {∃k. 0 ≤ k ∧ a = F_k ∧ b = F_{k+1} ∧ result = F_{k+n}}
= if n = 0 then a else aux b (a+b) (n-1)
```

```
let fib (n: int) : int
  requires {0 ≤ n}
  ensures  {result = F_n}
= aux 0 1 n
```

Ghost code

```
let rec aux (k: int) (a b n: int) : int
  requires {0 ≤ k ∧ 0 ≤ n}
  requires {a = F_k ∧ b = F_{k+1}}
  ensures  {result = F_{k+n}}
= if n = 0 then a else aux (k+1) b (a+b) (n-1)
```

```
let fib (n: int) : int
  requires {0 ≤ n}
  ensures  {result = F_n}
= aux 0 0 1 n
```

Extracted OCaml program

the corresponding OCaml program, without specification:

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

let fib n = aux 0 0 1 n
```

however, this program

- is not the original algorithm
- has superfluous computations

Erasing ghost code

since we know that ghost computations

- have no impact on the program outcome
- do not produce any side effect

we can replace the ghost code by a dummy value:

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

let fib n = aux () 0 1 n
```

Erasing ghost code

since we know that ghost computations

- have no impact on the program outcome
- do not produce any side effect

or simply drop the ghost code and obtain the original code

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

let fib n = aux 0 1 n
```

The spirit of ghost code

ghost code serves the purpose of specification and/or proof

the principle of non-interference:

> *ghost code can be erased without observable modification in the program outcome*

as a consequence:

- ghost code may read regular data but can't modify it
- ghost code cannot alter the control flow of regular code
- regular code does not see ghost data

Ghost code in practice

ghost code is supported by most modern verification tools

common features are:

- ghost data of base types (`int`, ...)
- regular functions with ghost parameters
- ghost functions
- ghost variables and ghost fields

for these and other features

- non-interference must be ensured
- there are some design choices to be made

Design choices

1. how explicit should annotations be?

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

let fib n = aux 0 0 1 n
```

Design choices

1. how explicit should annotations be?
2. what can be shared between ghost and regular code?
   - types (int, ...)
   - constructs (`while`, `if`, ...)
   - functions (procedures, methods, ...)

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

let fib n = aux 0 0 1 n
```

Design choices

1. how explicit should annotations be?
2. what can be shared between ghost and regular code?
   - types (int, ...)
   - constructs (`while`, `if`, ...)
   - functions (procedures, methods, ...)

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

let fib n = aux 0 0 1 n
```

- what about complex data types and operations?

User-defined data types

```
type nat = O | S of nat

let rec plus x y = match x with
 | O    → y
 | S x' → S (plus x' y)

let rec aux (ghost k: nat) a b n = match n with
  | O    → a
  | S n' → aux (plus k (S O)) b (plus a b) n'

let fib n = aux O O (S O) n
```

- what about side effects?

Side effects 1/3

side effects can violate non-interference

```
let rec plus x y = match x with
  | O →
      y
  | S x' →
      decr countdown;
      S (plus x' y)

let rec aux (ghost k: nat) a b n = match n with
  | O    → a
  | S n' → aux (plus (S O) k) b (plus a b) n'
```

Side effects 2/3

side effects can violate non-interference

```
let rec plus x y = match x with
  | O →
      if !ghost_countdown = 0 then raise LaunchMissile;
      y
  | S x' →
      decr ghost_countdown;
      S (plus x' y)

let rec aux (ghost k: nat) a b n = match n with
  | O    → a
  | S n' → aux (plus (S O) k) b (plus a b) n'
```

Side effects 3/3

side effects can violate non-interference

```
let rec plus x y = match x with
  | O →
      if !ghost_countdown = 0 then (λx.xx)(λx.xx);
      y
  | S x' →
      decr ghost_countdown;
      S (plus x' y)

let rec aux (ghost k: nat) a b n = match n with
  | O    → a
  | S n' → aux (plus (S O) k) b (plus a b) n'
```

Side effects 3/3

side effects can violate non-interference

```
let rec plus x y = match x with
  | O →
      if !ghost_countdown = 0 then (λx.xx)(λx.xx);
      y
  | S x' →
      decr ghost_countdown;
      S (plus x' y)

let rec aux (ghost k: nat) a b n = match n with
  | O    → a
  | S n' → aux (plus (S O) k) b (plus a b) n'
```

how to guarantee non-interference?

Our approach

a type system with

- minimal annotations
- possibility of sharing
- a proof of non-interference

this type system keeps track of side effects and ghost statuses

GhostML

a proof of concept:

- GhostML = simple ML-like language with:
  - mutable state (global references), recursive functions
  - ghost code
  - usual call-by-value ML semantics

GhostML

a proof of concept:

- GhostML = simple ML-like language with:
  - mutable state (global references), recursive functions
  - ghost code
  - usual call-by-value ML semantics

- erasure $\mathcal{E}$: GhostML $\rightarrow$ MiniML
  - turns ghost code into unit values
  - preserves regular code contents and structure
  - preserves regular code semantics (non-interference)

Type system

type system with boolean indicators: $\beta,\ \epsilon \in \{\top, \bot\}$

- β : ghost status
- ϵ : observable effects (non-ghost write, non-termination)

$$\boxed{\Gamma \ \vdash \ t \ : \ \tau,\ \beta,\ \epsilon}$$

function type with latent effects and parameter's ghost status:

$$\boxed{\Gamma \ \vdash \ t \ : \ (\tau_2 \overset{\beta_2}{\underset{\epsilon_1}{\Longrightarrow}} \tau_1),\ \beta,\ \epsilon}$$

side condition in every typing rule:

$$\boxed{\beta \Rightarrow \neg\epsilon}$$

typing application

two distinct cases for application $(t_1\ t_2)$:

- $t_1 : (\tau_2^{\bot} \stackrel{\epsilon_1}{\Rightarrow} \tau_1)$ (function with regular parameter)
- $t_1 : (\tau_2^{\top} \stackrel{\epsilon_1}{\Rightarrow} \tau_1)$ (function with ghost parameter)

application with a regular parameter

$$\frac{\Gamma \vdash t_1 : (\tau_2^{\bot} \stackrel{\epsilon_1}{\Longrightarrow} \tau_1),\ \beta_1,\ \epsilon_2 \qquad \Gamma \vdash t_2 : \tau_2,\ \beta_2,\ \epsilon_3}{\Gamma \vdash (t_1\ t_2) : \tau_1,\ \beta_1 \vee \beta_2,\ \epsilon_1 \vee \epsilon_2 \vee \epsilon_3}$$

- ghost status of t_2 contaminates the status of the application

regular argument

$$(\lambda v^{\bot} : \mathtt{int}.\ \ g := v\ ;\ !r)\ 42$$

$$\frac{\Gamma \vdash t_1 : (\mathtt{int}^{\bot} \Rrightarrow \mathtt{int}),\ \bot,\ \bot \qquad \Gamma \vdash t_2 : \mathtt{int},\ \bot,\ \bot}{\Gamma \vdash (t_1\ t_2) : \mathtt{int},\ \bot,\ \bot}$$

ghost argument and contamination

$$(\lambda v^{\bot} : \texttt{int}.\ \ g := v\ ;\ !r)\ \ (\text{ghost } 42)$$

$$\frac{\Gamma \vdash t_1 : (\texttt{int}^{\bot} \stackrel{\bot}{\Rightarrow} \texttt{int}),\ \bot,\ \bot \qquad \Gamma \vdash t_2 : \texttt{int},\ \top,\ \bot}{\Gamma \vdash (t_1\ t_2) : \texttt{int},\ \top,\ \bot}$$

ghost argument and contamination

$$(\lambda v^{\bot} : \texttt{int}.\ g := v;\ !r)\ (\text{ghost } 42)$$

$$\frac{\Gamma \vdash t_1 : (\texttt{int}^{\bot} \stackrel{\bot}{\Rightarrow} \texttt{int}),\ \bot,\ \bot \qquad \Gamma \vdash t_2 : \texttt{int},\ \top,\ \bot}{\Gamma \vdash (t_1\ t_2) : \texttt{int},\ \top,\ \bot}$$

ghost argument

$$(\lambda v^{\bot} : \texttt{int}.\ \ g := v\ ;\ \ r := 0\ )\ \ (\text{ghost } 42)$$

is rejected, since

$$\frac{\Gamma \vdash t_1 : (\texttt{int}^{\bot} \stackrel{\top}{\Rightarrow} \texttt{unit}),\ \bot,\ \bot \qquad \Gamma \vdash t_2 : \texttt{int},\ \top,\ \bot}{\Gamma \vdash (t_1\ t_2) : \texttt{unit},\ \top,\ \top}$$

violates the side condition $\beta \Rightarrow \neg\epsilon$

ghost argument

$$(\lambda v^{\bot} : \mathtt{int}.\, g := v;\quad r := 0\,)\ (\text{ghost } 42)$$

is rejected, since

$$\frac{\Gamma \vdash t_1 : (\mathtt{int}^{\bot} \xRightarrow{\top} \mathtt{unit}),\ \bot,\ \bot \qquad \Gamma \vdash t_2 : \mathtt{int},\ \top,\ \bot}{\Gamma \vdash (t_1\ t_2) : \mathtt{unit},\ \top,\ \top}$$

violates the side condition $\beta \Rightarrow \neg\epsilon$

ghost parameter: a friendly ghost

we can repair previous example

$(\lambda v^{\bot} : \texttt{int}.\, g := v;\quad r := 0\,)\ (\text{ghost } 42)$

by changing the ghost status of parameter

$(\lambda\, v^{\top} : \texttt{int}.\ \ g := v\,;\ \ r := 0\,)\ \ (\text{ghost } 42)$

application with a ghost parameter

$$\frac{\Gamma \vdash t_1 : (\tau_2 \overset{\top}{\overset{\epsilon_1}{\Longrightarrow}} \tau_1),\ \beta_1,\ \epsilon_2 \qquad \Gamma \vdash t_2 : \tau_2,\ \beta_2,\ \bot}{\Gamma \vdash (t_1\ t_2) : \tau_1,\ \beta_1,\ \epsilon_1 \vee \epsilon_2}$$

- argument t_2 must be pure
- ghost status of application is that of t_1

Non-interference

we prove that erasure $\mathcal{E}$ preserves regular code semantics

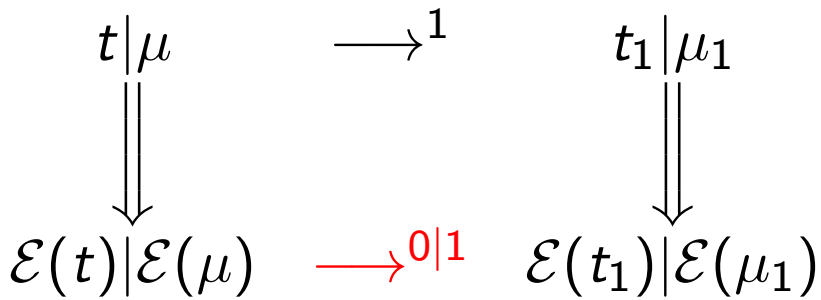
Theorem (non-interference):

for any t such that $\boxed{\Gamma \vdash t : \tau, \bot, \epsilon}$ we have

$$\begin{array}{ccc} t|\mu & \longrightarrow^{\star} & v|\mu' \\ \Downarrow & & \Downarrow \\ \mathcal{E}(t)|\mathcal{E}(\mu) & \longrightarrow^{\star} & \mathcal{E}(v)|\mathcal{E}(\mu') \end{array} \qquad \begin{array}{ccc} t|\mu & \Longrightarrow & \infty \\ \Downarrow & & \Downarrow \\ \mathcal{E}(t)|\mathcal{E}(\mu) & \Longrightarrow & \infty \end{array}$$

Non-intereference 1/2

we prove non-interference for terminating programs
using a forward simulation lemma



Non-intereference 2/2

we prove non-interference for diverging programs
using coinduction on $t|\mu \Rightarrow \infty$ and

$$\begin{array}{ccc} t|\mu & \longrightarrow^{\geq 1} & t_1|\mu_1 \\ \Downarrow & & \Downarrow \\ \mathcal{E}(t)|\mathcal{E}(\mu) & \longrightarrow^{1} & t'|\mu' = \mathcal{E}(t_1)|\mathcal{E}(\mu_1) \end{array}$$

Implementation in Why3

our approach is implemented in the verification tool Why3 (see `http://why3.lri.fr/`)

ghost code features:

- type polymorphism
- local references
- data structures with ghost fields
- exceptions
- provable termination

Example

```
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

...
```

Conclusion

ghost code is folklore, yet

- it is subtle to get it right
- few formalizations exist

our contributions

- use of a type system
- proof of non-interference
- described in a paper submitted to ESOP
  `http://hal.inria.fr/hal-00873187`