

Formally verifying properties of a toy language

Guillem Bartrina I Moreno Franco Sainas Marcin Wojnarowski

École Polytechnique Fédérale de Lausanne (EPFL)

CS550 - Formal Verification

December 2023

Outline

1 Introduction

2 Language

- Description
- Syntax
- Semantics

3 Approach

- Abstract state machine

4 Properties

- Closedness
- No redeclarations
- Unique ownership
- No use-after-free

5 Implementation

- Stainless interpreter
- Stainless tracer
- Lean interpreter

6 Conclusions

Introduction

- Vast majority of programming languages do not have a formally verified core
- Functional languages are studied a lot, but the real world is messy and dominated by imperative languages
- The goal is to define a toy language for which we can formally reason about some properties

Language

Description

- Imperative
- The only value type is a **boolean**
- All variables are "heap" allocated
- Lexical scoping
- Started ambitious (product types, references, deep mutability), *quickly* humbled

Description

- Imperative
- The only value type is a **boolean**
- All variables are "heap" allocated
- Lexical scoping
- Started ambitious (product types, references, deep mutability), *quickly* humbled

Syntax

```

1  let myVar = true
2  let other = false
3
4  if other ↑ other {
5      myVar := myVar ↑ other
6      free other
7  }
8
9  while myVar {
10     let p = myVar ↑ true
11     myVar := p ↑ (p ↑ p)
12 }
    
```

$\langle expr \rangle$	$::=$ true false	Bool
	$\langle expr \rangle_1 \uparrow \langle expr \rangle_2$	Nand
	$\langle name \rangle$	Ident
	$(\langle expr \rangle)$	Group
$\langle stmt \rangle$	$::=$ let $\langle name \rangle = \langle expr \rangle$	Decl
	$\langle name \rangle := \langle expr \rangle$	Assign
	if $\langle expr \rangle$ { $\langle stmt \rangle$ }	If
	while $\langle expr \rangle$ { $\langle stmt \rangle$ }	While
	free $\langle name \rangle$	Free
	$\langle stmt \rangle_1 \langle stmt \rangle_2$	Seq

Semantics

- Free conservatively forbids further usage of the variable

```
let var = true
if false {
  free var
}
var = false # illegal
```

- Decl defines a variable in its scope

```
if false {
  let var = true
}
var = false # not accessible
```

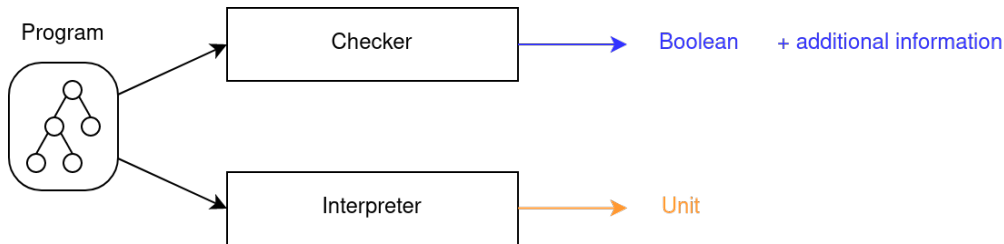
- Decl cannot shadow

```
let var = true
if false {
  let var = true # illegal
}
```


Approach

Approach

Implement an interpreter for our toy language. Given that a program is valid, prove that its execution by the interpreter enjoys some properties.



Goal: Program is valid \Rightarrow Program executes successfully (does not throw)

Abstract state machine

Environment $env : \text{Name} \rightarrow \text{Abstract location}$
 Memory $mem : \text{Abstract location} \rightarrow \text{Value}$
 Allocator $alloc : _ \rightarrow \text{Abstract location}$

Abstract state: $(env, mem, alloc)$

Properties

Closedness

All variable accesses exist in the current environment.

Definition (Closedness)

A program is closed if whenever evaluating $\text{Ident}\langle\text{name}\rangle$ or $\text{Assign}\langle\text{name}, \text{expr}\rangle$, $\text{env}(\text{name})$ is defined.

```
1 var1 := true # error
2 if var2 { # error
3 }
```

No redeclarations

A declaration cannot declare an already declared name.

Definition (No redeclarations)

A program has no redeclarations if whenever evaluating $\text{Decl}\langle \text{name}, \text{expr} \rangle$, $\text{env}(\text{name})$ is not defined.

```
1 let var = true
2 let var = false # error
3 if true {
4     let var = false # error
5 }
```

Unique ownership

No two variables in the environment point to the same location.

Definition (Unique ownership)

A program exhibits unique ownership when *env* is injective at all times.

No use-after-free

All variable accesses point to existing memory.

Definition (No use-after-free)

A program has no uses-after-free if whenever evaluating $\text{Ident}\langle\text{name}\rangle$ or $\text{Assign}\langle\text{name}, \text{expr}\rangle$, $\text{mem}(\text{env}(\text{name}))$ is defined.

```
1 let var = true
2 free var
3 var := true # error
```


Implementation

Implementation

- Stainless interpreter (big-step flavour)
- Lean interpreter
- Stainless tracer (small-step flavour)

Implementation details:

- Avoid throwing in Stainless: interpretation functions return either a set of exceptions or the actual result.
- Limited interoperability between Maps and Sets in Stainless: introduce several axioms.

Stainless interpreter

Interpreter : $Prog \rightarrow State$

Given a program *Prog*, the interpreter returns the final state *State*.

`def evalStmt(stmt: Stmt, state: State): Either[Set[LangException], State]`

- Pros: Most natural design, straightforward implementation, *closer* to the checker
- Cons: Symmetries with checker and proofs

Stainless tracer

Interpreter problem with whiles: non termination.

Given a program P the tracer returns a list of states.

We mainly focus on the part of the tracer that given a program P and a state S returns the program P' the state S' given by one step of execution.

$$T : \text{Prog} \rightarrow \text{State}^*$$

$$T_1 : \text{Prog} \times \text{State} \rightarrow \text{Prog} \times \text{State}$$

- Pros: More control over intermediate states, interesting properties about the trace.
- Cons: Many preconditions about the input state, prove preservation of properties.

Stainless tracer

Interpreter problem with whiles: non termination.

Given a program P the tracer returns a list of states.

We mainly focus on the part of the tracer that given a program P and a state S returns the program P' the state S' given by one step of execution.

$$T : \text{Prog} \rightarrow \text{State}^*$$

$$T_1 : \text{Prog} \times \text{State} \rightarrow \text{Prog} \times \text{State}$$

- Pros: More control over intermediate states, interesting properties about the trace.
- Cons: Many preconditions about the input state, prove preservation of properties.

Stainless tracer

Interpreter problem with whiles: non termination.

Given a program P the tracer returns a list of states.

We mainly focus on the part of the tracer that given a program P and a state S returns the program P' the state S' given by one step of execution.

$$T : \text{Prog} \rightarrow \text{State}^*$$

$$T_1 : \text{Prog} \times \text{State} \rightarrow \text{Prog} \times \text{State}$$

- Pros: More control over intermediate states, interesting properties about the trace.
- Cons: Many preconditions about the input state, prove preservation of properties.



Lean interpreter

```
1 partial def evalStmt
2   (stmt : Stmt) (env : Env) (mem : Memory)
3   (h : isTypeCheckedStmt stmt (keySet env))
4   : Env × Memory := match stmt with
5   -- ...
6   | Stmt.conditional condition body =>
7     let cond := evalExpr condition env mem (typeCheck_conditionalCond h)
8     let (newEnv, newMem) := if cond
9       then evalStmt body env mem (typeCheckStmt_conditionalBody h)
10      else (env, mem)
11
12     -- we drop the new env, but keep the new mem
13     (env, newMem)
14   -- ...
```

Conclusions

Discussion

- Performance has to be traded for provable correctness
- Symmetricity between properties and implementation
- Requires intermediate lemmas proving correlation between properties and implementation
- Proving correctness is hard and very time consuming
- Despite the language being a subset of our original design, we are happy with the results

Future work

- Lack of memory leaks
- More language features

Merci!