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

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

```
let myVar = true
let other = false

if other ↑ other {
    myVar := myVar ↑ other
    free other
}

while myVar {
    let p = myVar ↑ true
    myVar := p ↑ (p ↑ p)
}
```

$$
\begin{array}{llll}
\langle expr \rangle & ::= & \mathsf{true} \mid \mathsf{false} & \texttt{Bool} \\
 & \mid & \langle expr \rangle_1 \uparrow \langle expr \rangle_2 & \texttt{Nand} \\
 & \mid & \langle name \rangle & \texttt{Ident} \\
 & \mid & (\langle expr \rangle) & \texttt{Group}
\end{array}
$$

$$
\begin{array}{llll}
\langle stmt \rangle & ::= & \mathsf{let}\ \langle name \rangle = \langle expr \rangle & \texttt{Decl} \\
 & \mid & \langle name \rangle := \langle expr \rangle & \texttt{Assign} \\
 & \mid & \mathsf{if}\ \langle expr \rangle\ \{\ \langle stmt \rangle\ \} & \texttt{If} \\
 & \mid & \mathsf{while}\ \langle expr \rangle\ \{\ \langle stmt \rangle\ \} & \texttt{While} \\
 & \mid & \mathsf{free}\ \langle name \rangle & \texttt{Free} \\
 & \mid & \langle stmt \rangle_1\ \langle stmt \rangle_2 & \texttt{Seq}
\end{array}
$$

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
```

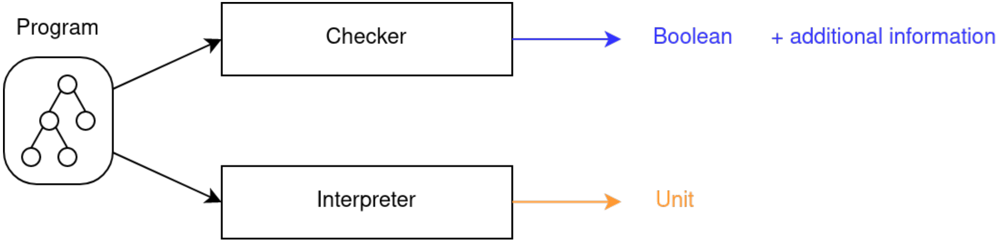
- `Decl` cannot shadow

```
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

$$\begin{aligned}
\text{Environment} \quad & \mathit{env} : \text{Name} \to \text{Abstract location} \\
\text{Memory} \quad & \mathit{mem} : \text{Abstract location} \to \text{Value} \\
\text{Allocator} \quad & \mathit{alloc} : _ \to \text{Abstract location}
\end{aligned}$$

Abstract state: $(\mathit{env}, \mathit{mem}, \mathit{alloc})$

Properties

Closedness

All variable accesses exist in the current environment.

Definition (Closedness)

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

```
var1 := true # error
if var2 { # error
}
```

No redeclarations

A declaration cannot declare an already declared name.

Definition (No redeclarations)

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

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

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 $\texttt{Ident}\langle\text{name}\rangle$ or $\texttt{Assign}\langle\text{name}, \text{expr}\rangle$, $\mathit{mem}(\mathit{env}(\text{name}))$ is defined.

```
let var = true
free var
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.

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

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

- 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.

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

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

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

Lean interpreter

```
partial def evalStmt
    (stmt : Stmt) (env : Env) (mem : Memory)
    (h : isTypeCheckedStmt stmt (keySet env))
    : Env × Memory := match stmt with
  -- ...
  | Stmt.conditional condition body =>
    let cond := evalExpr condition env mem (typeCheck_conditionalCond h)
    let (newEnv, newMem) := if cond
        then evalStmt body env mem (typeCheckStmt_conditionalBody h)
        else (env, mem)

    -- we drop the new env, but keep the new mem
    (env, newMem)
  -- ...
```

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!