Formally Verifying Properties of a Toy Language – Project Report

Formal Verification, EPFL

Guillem Bartrina I Moreno
guillem.bartrinaimoreno@epfl.ch

Franco Sainas
franco.sainas@epfl.ch

Marcin Wojnarowski
marcin.wojnarowski@epfl.ch

January 9, 2024

1 Introduction

The vast majority of available mainstream programming languages were created without much of consideration of their formal foundation. This makes it difficult to reason about safety guarantees. One can see how that leads to memory errors, unsoundness, and general unsafety of the system. When retrospectively trying to fix such issues, languages are often met with a important decision whether to sacrifice backwards compatibility in return for a safer system. One such great example is the C++ language which has been desperately fighting to be seen as a safe language for many years while being very committed to backwards compatibility of the inherited problems of C. As such, even C++'s creator tries to distance himself from the criticism of C++ being unsafe by saying *"There is no "C/C++ language""*[4]. Only a subset of C programs can be formally proven to be memory safe.

Our goal is to focus on creating a language for which we can formally reason about its properties. That is, the language is defined in terms of its properties which guarantee safety of all programs written in the language, not only a subset of them. While there exist languages which have very strong memory safety guarantees such as `unsafe`-free Rust for which the memory model has been formally proven to be correct[2], these formal proofs do not relate to the real implementation of the language in any way. Our approach will differ in the way that the proofs will be conducted on the implementation of the language rather than some abstract model of it. This gives additional guarantees of the absence of bugs (human errors), assuming the properties have been stated properly.

Since functional languages have been studied a lot and the real world is messy, in this report we define a toy imperative programming language with semantics that allow us to formally show that the implementation of it given some properties (Sec. 3) follows an error-free execution. As a backing paper we used the formal definition of the static analysis of the Move language[1] which concerns itself with memory safety issues. It too however proves things on an abstract model rather than the implementation.

We start by defining the language, its properties, how we approached the implementation, and conclude with a discussion of how far we got and what challenges we have faced.

2 Language

We introduce FormalLang, a programming language for which we will develop an interpreter in subsequent sections. This language is designed to prioritize correctness of its execution, exhibiting intentional minimalism in its features. Notably, it exclusively employs a singular data type—boolean—and relies solely on the functionally complete NAND (↑) operator. Since we are not interested in type properties we find the boolean type satisfatory. It is crucial to emphasize that, at this preliminary stage, the language is intentionally unsuitable for real-world applications. Despite this limitation, it serves as a foundational starting point crucial to achieving the project's overarching goal: the formal verification of properties associated with FormalLang. We provide a code snippet to exemplify a valid program written in FormalLang (Lst. 1).

As evident in the code snippet, variables are declared by specifying an expression, and the resulting evaluated value serves as the initialized value for the variable. The language further supports the mutation variables, if and while statements, and the freeing of variables from memory. The formal grammar (Fig. 1), is subsequently provided for comprehensive understanding.

```
let myVar = true
let other = false

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

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

Listing 1: Program snippet written in FormalLang.

$$
\begin{array}{lr}
\langle expr \rangle ::= \text{true} \mid \text{false} & \texttt{Bool} \\
\quad \mid \langle expr \rangle_1 \uparrow \langle expr \rangle_2 & \texttt{Nand} \\
\quad \mid \langle name \rangle & \texttt{Ident} \\
\quad \mid (\langle expr \rangle) & \texttt{Group} \\
\\
\langle stmt \rangle ::= \text{let } \langle name \rangle = \langle expr \rangle & \texttt{Decl} \\
\quad \mid \langle name \rangle := \langle expr \rangle & \texttt{Assign} \\
\quad \mid \text{if } \langle expr \rangle \text{ \{ } \langle stmt \rangle \text{ \} } & \texttt{If} \\
\quad \mid \text{while } \langle expr \rangle \text{ \{ } \langle stmt \rangle \text{ \} } & \texttt{While} \\
\quad \mid \text{free } \langle name \rangle & \texttt{Free} \\
\quad \mid \langle stmt \rangle_1 \ \langle stmt \rangle_2 & \texttt{Seq}
\end{array}
$$

Figure 1: Grammar of FormalLang expressed in BNF-like syntax.

2.1 Operational Semantics

2.1.1 Virtual Model

In our virtual model, we define a state represented by a tuple. This tuple comprises the essential components of the system, including the environment (env), the set of freed variables (freed), the memory (mem), and the next free location in memory (l)

Formally, the components are defined as follows:

- The environment is a mapping from variable names to locations

$$\text{env} : \text{Names} \to \text{Locs}$$

- The set of freed variables is a subset of variable names

$$\text{freed} \subseteq \text{Names}$$

- The memory is a mapping from locations to boolean values

$$\text{mem} : \text{Locs} \to \text{Bool}$$

- The next free location is an element of the set of locations

$$l \in \text{Locs}$$

- The state itself is defined as a tuple

$$\text{state} = (\text{env}, \text{freed}, \text{mem}, l)$$

We now state the operational semantics in big-step.

2.1.2 Expressions

$$\textsc{True}\quad \langle \text{True}, \text{state} \rangle \mapsto_v \textbf{true} \qquad \textsc{False}\quad \langle \text{False}, \text{state} \rangle \mapsto_v \textbf{false}$$

$$\textsc{Nand}\quad \frac{\langle e_1, \text{state} \rangle \mapsto_v v_1 \quad \langle e_2, \text{state} \rangle \mapsto_v v_2}{\langle e_1 \uparrow e_2, \text{state} \rangle \mapsto_v v_1 \uparrow v_2}$$

$$\textsc{Ident}\quad \frac{X \in \text{dom}(\text{env}) \quad \text{env}(X) \in \text{dom}(\text{mem})}{\langle X, (\text{env}, \text{freed}, \text{mem}, l) \rangle \mapsto_v \text{mem}(\text{env}(X))}$$

2.1.3 Statements

$$\textsc{Decl}\quad \frac{X \notin \text{dom}(\text{env}) \quad \langle e, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_v v}{\langle \text{let } X = e, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_s (\text{env}\{l/X\}, \text{freed}, \text{mem}\{v/l\}, l+1)}$$

$$\textsc{Assign}\quad \frac{X \in \text{dom}(\text{env}) \quad \text{env}(X) \in \text{dom}(\text{mem}) \quad X \notin \text{freed} \quad \langle e, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_v v}{\langle X := e, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_s (\text{env}, \text{freed}, \text{mem}\{v/\text{env}(X)\}, l)}$$

$$\textsc{If-true}\quad \frac{\langle e, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_v \textbf{true} \quad \langle s, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_s (\text{env}', \text{freed}', \text{mem}', l')}{\langle \text{if } e\ \{s\}, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_s (\text{env}, \text{freed}', \text{mem}', l')}$$

$$\textsc{If-false}\quad \frac{\langle e, \text{state}\rangle \mapsto_v \textbf{false}}{\langle \text{if } e\ \{s\}, \text{state}\rangle \mapsto_s \text{state}}$$

$$\textsc{While-True}\quad \frac{\langle e, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_v \textbf{true} \quad \langle s, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_s (\text{env}', \text{freed}', \text{mem}', l') \quad \langle \text{while } e\ \{s\}, (\text{env}, \text{freed}', \text{mem}', l')\rangle \mapsto_s (\text{env}'', \text{freed}'', \text{mem}'', l'')}{\langle \text{while } e\ \{s\}, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_s (\text{env}, \text{freed}'', \text{mem}'', l'')}$$

$$\textsc{While-False}\quad \frac{\langle e, \text{state}\rangle \mapsto_v \textbf{false}}{\langle \text{while } e\ \{s\}, \text{state}\rangle \mapsto_s \text{state}} \qquad \textsc{Seq}\quad \frac{\langle s_1, \text{state}\rangle \mapsto_s \text{state}' \quad \langle s_2, \text{state}'\rangle \mapsto_s \text{state}''}{\langle s_1\ s_2, \text{state}\rangle \mapsto_s \text{state}''}$$

$$\textsc{Free}\quad \frac{X \in \text{dom}(\text{env}) \quad X \notin \text{freed}}{\langle \text{free } X, (\text{env}, \text{freed}, \text{mem}, l)\rangle \mapsto_s (\text{env}, \text{freed} \cup \{X\}, \text{mem} \setminus \{\text{env}(X)\}, l)}$$

Here, we denote $map\{y/x\}$ the extension of map with a new mapping from x to y.

Notice how exiting an if statement drops the environment but keeps the information about the freed variables and the changed memory. This implies two semantics: variables have (lexical) scopes and the other feature will become more apparent in the checker, which will have to conservatively assume the condition is true and assume the variable was freed.

3 Properties

Using the abstract model we define on it properties that we wish to prove about the language. A few properties are needed for general correctness of execution while others are for providing some guarantees which we are interested in.

3.1 Closedness

Names of all variable accesses exist in the current environment.

Definition 3.1 (Closedness)**.** A program is closed if whenever evaluating `Ident`⟨name⟩, `Assign`⟨name, expr⟩, or `Free`⟨name⟩ then env(name) is defined.

```
var1 := true # error
if var2 { # error
}
free var3 # error
```

This ensures that all identifiers we use point to declared variables existing within the current scope.

3.2 No redeclarations

A declaration cannot declare an already declared name.

Definition 3.2 (No redeclarations)**.** A program has no redeclarations if whenever evaluating `Decl`⟨name, expr⟩ then env(name) is not defined.

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

This ensures that variables cannot be shadowed. This property was introduced to simplify reasoning about the environment.

3.3 Unique ownership

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

Definition 3.3 (Unique ownership)**.** A program exhibits unique ownership when env is injective at all times.

This has no snippet to show violation as this property cannot be actually exhibited in code. This property is useful for ensuring that a free construct will not create dangling pointers: if there were two names pointing to the same location, freeing one of the variables would invalidate both variables. But since we only track frees of the variables to which the free construct was applied to, we are unable to reason about such cases. Thus we need to make sure that applying free to a variable will not affect any other variables.

3.4 No use-after-free

If a variable has been freed, we can no longer use it.

Definition 3.4 (No use-after-free)**.** A program has no uses-after-free if whenever evaluating `Ident`⟨name⟩, `Assign`⟨name, expr⟩, or `Free`⟨name⟩ then name $\notin freed$.

```
let var = true
free var
var := true # error
let bar = true
if false {
    free bar
}
bar := true # error
```

3.5 No dangling pointers

If a variable has not been freed then it has a valid memory address.

Definition 3.5 (No dangling pointers)**.** A program has no dangling pointers if whenever evaluating `Ident`⟨name⟩, `Assign`⟨name, expr⟩, or `Free`⟨name⟩ and name $\notin freed$ then $mem(env(\text{name}))$ is defined.

This ensures that when we evaluate a valid identifier we can be certain that it points to a valid memory address. We restrict the definition to non-freed variables as freed ones no longer point to valid memory addresses.

4 Approach

As mentioned already, in this project we intend to formally prove properties on the *implementation* of the language rather than on some abstract model of it, which is the usual practice.

When we talk about the *implementation* of a language, we usually refer to an interpreter program that implements its operational semantics. The interpreter takes as input a representation of a valid program and executes it on a virtual machine: applies the sequence of state changes on the virtual machine given by the program statements. In this framework, proving properties on the implementation of a language, and subsequently on the language itself, translates to proving properties about the execution of valid programs on the virtual machine by the interpreter. As a by-product, we can also obtain guarantees about the correctness of the implementation of the interpreter.

The validity of the program supplied to the interpreter is crucial. It is precisely this assumption that provides the evidence needed to prove the properties of its execution.

Our approach for this project has been to implement an interpreter for the presented language and then to prove that the execution of any valid program by such interpreter enjoys some properties.
More precisely,

An instance of a program in our language is represented by an abstract syntax tree, which is just a complex recursive data-structure. The definition of the AST model admits the construction of invalid programs.

The validity of a given program is determined by the checker or validation procedure, which ensures that the program adheres to the specification. The checker also provides additional information that is convenient for the proofs.

The language interpreter takes as input a program and executes it on the virtual machine. The execution can either terminate successfully or be aborted by an exception. Exceptions signal the reachability of some incorrect state of the virtual machine, which should not have been reached. All the exceptions reflect the violation of some of the properties that we want to prove, but not all the properties have an exception associated.

With this, our goal is to prove that given that a program is valid, the execution of this program by the interpreter enjoys the desired properties: no exceptions are thrown, or other properties.

5 Implementation

When starting the project many challenges were quickly identified. There were two big decisions to be made at the very start:

1. Do we want to complete the project in Stainless or in a more manual theorem prover?
2. What implementation of the interpreter do we use, a big-step-like or a small-step-like?

The first question was related to the amount of control we wanted in the proofs. Stainless is incredible at proving things that can be easily seen to be true saving a lot of time. On the other hand interactive theorem provers usually require a lot more manual labor to show even the simplest facts. But on the flip side relying on Stainless' inference can be detrimental once Stainless no longer can see a fact. Familiar with Stainless, we opted for using the tool we know best.

The second question concerns itself with which implementation will be easier to prove properties about. We quickly noticed that having the `While` construct in the language meant that an interpreter that tries to evaluate the whole program at once (big-step-like) would be non terminating. We thought this would lead to big complications so we chose to pursue the small-step-like interpreter.

Even though we had chosen the path we want to pursue, due to hitting many hard blockers on the way we kept experimenting with two alternative paths with hopes of getting further with them. Considering many hours were spent on the alternative implementations, we want to outline them here to contrast the various approaches and then contrast them in Sec. 6.

The whole source code is stored on GitHub.

5.1 Lean

Lean is a functional programming language that can be used also as an interactive theorem prover. Similarly to a different popular theorem prover, Coq, Lean is based on the *Calculus of Constructions*. Mathlib[3] was used to provide foundation of structures such as finite sets and association lists. The Lean implementation can be found on GitHub with the following files:

- `Allocator.lean` Simple bump allocator which manages Locs and their underlying memory.
- `Ast.lean` This module defines the abstract syntax tree of the language.
- `Checker.lean` This module focuses on static properties of a program. Meaning, the analysis that can be performed on source code without actually evaluating it.
- `Helpers.lean` This module stores helper lemmas, functions, utils, etc.
- `Interpreter.lean` This module performs evaluation of the source code. It evaluates an AST given a proof that the type checker has accepted this AST.

5.1.1 Structures

```
structure Name where
  name : String

inductive Expr where
  | true
  | false
  | nand (left right : Expr)
  | ident (name : Name)

inductive Stmt where
  | decl (name : Name) (value : Expr)
  | assign (target : Name) (value : Expr)
  | conditional (condition : Expr) (body : Stmt)
  | while (condition : Expr) (body : Stmt)
  | seq (left right : Stmt)
  | free (name : Name)
```

Listing 2: AST structures defined in Lean

```
/-- A memory location represented as a
    natural number. Basically a newtype over
    Nat. -/
structure Loc where
  loc : Nat

/-- The memory maps variable locations to
    values. -/
abbrev Memory := @AList Loc (fun _ => Bool)

/-- The environment maps variable names to
    memory locations. -/
abbrev Env := @AList Name (fun _ => Loc)

/-- A set of freed variables. -/
abbrev Freed := Finset Name
```

Listing 3: Definition of models in Lean

The AST definition (Lst. 2) follows a standard format. There is a separation between expressions and statements, and Name is a new type to not be confused with a string.

Then follow definitions (Lst. 3) of the model structures needed to reason about the state of the interpreter. We use an association list to represent partial maps. Keys of an association list are unique, and thus form a finite set. This relation is important when relating the checker and the interpreter (to be discussed shortly).

5.1.2 Checker

```
/-- Returns true if the expression is well-formed. -/
def typeCheckExpr (expr : Expr) (vars : Variables) : Bool :=
    match expr with
    | Expr.true => Bool.true
    | Expr.false => Bool.true
    | Expr.nand left right => (typeCheckExpr left vars) && (typeCheckExpr right vars)
    | Expr.ident name => name ∈ vars

def typeCheckStmt (stmt : Stmt) (vars : Variables) : Option Variables :=
    match stmt with
    | Stmt.decl name value =>
        if name ∉ vars then
          let newVar := insert name vars
```

```
        if typeCheckExpr value vars then
          some newVar
        else
          none
      else none
  -- ...

/-- The assertion that `typeCheckStmt` accepted the input, ie it is not `none`. -/
def isTypeCheckedStmt (stmt : Stmt) (vars : Variables) := (typeCheckStmt stmt vars).isSome
```

Listing 4: The checker for a FormalLang program. `typeCheckStmt` shortened to only show handling of declarations.

The checker is a function which traverses each node of the AST once to verify some static properties of a program. Since a checker is not concerned with allocation, it does not keep track of memory nor of any locations. Therefore the `Variables` (Lst. 4) it keeps track of is merely a finite set of names. The checker module additionally defines the `isClosedStmt` and `hasNoRedeclarations` properties, which are in fact just the subset of things being checked by the larger `typeCheckStmt`. The rest of the module defines many useful lemmas (for instance, given that `Stmt.decl` has been accepted by the checker, we know that `Stmt.decl.value` has also been accepted by the checker) and concludes with two larger ones: being accepted by the checker implies that the program is closed and has no redeclarations. To produce such a proof we first need to prove that the `Variables` being tracked by the checker and the properties are the same. This is needed when stepping through `Stmt.seq`. Once we know the tracked variables are indeed the same, it trivially follows that if the checker accepts some input then so will the properties.

5.1.3 Interpreter

The checker module does not need to know anything about the interpreter for the checker only reasons about static properties. However, the converse is not true. For the interpreter to execute correctly, it must have a guarantee that the checker has accepted the input.

```
/-- Entries in env have allocated memory. -/
def hasValidLocs (env : Env) (mem : Memory) := ∀ name, ∀ loc, env.lookup name = some loc → loc ∈ mem

/-- Describes the result of evaluating a statement. Contains the resulting state as well as proofs for invariants. -/
structure EvalResult (stmt : Stmt) (env : Env) where
  newEnv : Env
  newMem : Memory
  -- proof that the env tracked by the type checker and the interpreter is the same
  sameEnv : typeCheckStmt stmt (keySet env) = some (keySet newEnv)
  -- proof that all items in env have entries in mem
  validLocs : hasValidLocs newEnv newMem

/-- Evaluates an expression given a proof that the type checker has accepted this input. -/
def evalExpr
  (expr : Expr) (env : Env) (mem : Memory)
  (h : typeCheckExpr expr (keySet env)) (validLocs : hasValidLocs env mem)
  : Bool := match expr with
  -- ...
  | Expr.ident name =>
    let loc := AList.get name env (typeCheckExpr_ident h)
    let val := AList.get loc mem (by
      have p : AList.lookup name env = some loc := by
        simp_all only [AList.get, Option.isSome_some]
        split
        simp_all only [Option.some.injEq, Option.isSome_some, heq_eq_eq]
      have q := validLocs name loc

      simp_all only [AList.get, Option.isSome_some, forall_true_left]
    )
    val

def evalStmt
  (stmt : Stmt) (env : Env) (mem : Memory)
  (h : isTypeCheckedStmt stmt (keySet env)) (validLocs : hasValidLocs env mem)
  : EvalResult stmt env := match stmt with
  -- ...
```

Listing 5: The interpreter. Evaluates structures given proofs of correctness. Cut for brevity.

The interpreter (Lst. 5) receives as input the AST, the environment, and the memory, but also proofs of correctness. The most important input proof is being accepted by the checker. This allows the interpreter

to formally prove lack of some runtime errors. For instance, since we know the input is closed, we know that when evaluating `Expr.ident` we know that this identifier exists in the environment. This is visible in the listing as `AList.get name env (typeCheckExpr_ident h)`, where we get an element from the environment by providing a proof that it is indeed there. This proof is deduced by a lemma based on `h`, the proof that the input has been accepted by the checker.

The interpreter also has to inductively produce proofs when stepping through the AST. As visible in `EvalResult`, the interpreter maintains proofs such as:

- The key set of the environment is the same as the one computed by the type checker. This equality is again needed when stepping through `Stmt.seq`.
- All values in the environment are valid keys to the memory, as expressed by the proposition `hasValidLocs`. This is of course needed whenever dereferencing memory to make sure the memory entry is actually there. This can be seen being in a proof when accessing the memory to compute `let val`.

As one can see, while the interpreter is not necessarily a terminating functions, due to the evaluation of such constructs as `While`, it is a pure one. The result will be always computed without any errors being encountered on the way.

When new variables are declared they go through an allocator, which mimics a bump allocator. It looks at the current environment and takes the maximum of *env*[Names] and adds one, yielding a unique location. At that location a new memory cells is added to maintain `hasValidLocs`.

5.1.4 Implementation status

The implemention has all language constructs except `Free` which does not actually perform freeing. Not all properties were proven. This implementation not being the primary focus, while still a lot of time, had less time spent on it. Closedness (Sec. 3.1) and lack of redeclarations (Sec. 3.2) have been fully proven. Unique ownership (Sec. 3.3) was not started, but the idea would be simple: given that only `Stmt.decl` alters the environment, we can derive unique ownership from the fact that the allocator returns a location that is not used by any variable. Use-after-free (Sec. 3.4) was not even attempted due to lack of a full `Free` implementation. Finally, no dangling pointers (Sec. 3.5) has a partial implementation with missing steps but with code comments about how the proof should be continued.

5.2 Stainless

As previously indicated, we have implemented both an interpreter and a tracer in Scala. The interpreter follows a big-step operational semantics, executing the entire program in a single step by recursively traversing the abstract syntax tree representation. In contrast, the tracer employs a small-step operational semantics, executing one interpretation step at a time and returning the pair next statement and updated state.

The code is available on GitHub has the following identical structure for both implementations:

- `AST.scala` defines the abstract syntax tree.
- `Model.scala` defines the state model, including memory and environment, along with FormalLang exceptions.
- `Interpreter.scala` contains the implementation of the interpreter.
- `Checker.scala` contains the implementation of the checker.
- `Proofs.scala` contains the Stainless proofs.

5.2.1 Structures

The AST 6 is defined by means of two algebraic data types, one for expressions and one for statements. `_Block` is a special statement used internally by the tracer and cannot be used to construct programs. The model 7 defines the types and structures needed to reason about the state of the interpreter and the tracer.

```
type Name = String

enum Expr {
  case True
  case False
  case Nand(val left: Expr, val right: Expr)
  case Ident(val name: Name)
}

enum Stmt {
  case Decl(val name: Name, val value: Expr)
  case Assign(val to: Name, val value: Expr)
  case If(val cond: Expr, val body: Stmt)

  case Seq(val stmt1: Stmt, val stmt2: Stmt)
  case Free(val name: Name)

  case _Block(val stmt: Stmt)  // Tracer
}
```

Listing 6: AST structures defined in Scala

```
type Loc = BigInt
type Env = Map[Name, Loc]
type Mem = Map[Loc, Boolean]

// --- Interpreter ---
case class State(val env: Env, val mem: Mem, val nextLoc:
     Loc)

// --- Tracer ---
case class State(val envs: List[Env], val mem: Mem, val
    nextLoc: Loc)
enum Conf:
  case St(state: State)
  case Cmd(stmt: Stmt, state: State)
```

Listing 7: Definition of models in Scala

The environment and the memory are standard Stainless maps. The state of the interpreter uses a single environment whereas that of the tracer uses a stack of environments. Additionally, the tracer defines another algebraic data type for the two kinds of configurations handled in its logic.

Much like for Lean, in our proofs we need to reason about sets and maps and their sets of keys. Stainless offers limited interoperability between these abstractions, so we had to add some convenient operations and define some axioms on them.

For instance, we have defined the operation ***`keySet`*** *that takes a map and returns its set of keys. Then we have defined the following axiom: if a given set S is equals to the set of keys of a given map M, then the set obtained by adding some value k to S is equal to the set of keys of the map obtained by adding some pair $k \to v$ to M.*

5.2.2 Exception handling

The following are the exceptions (`LangException`) that can be thrown by the interpreter or the tracer:

- `UndeclaredVariable`: Occurs when attempting to access a variable that has not been declared within the environment;
- `RedeclaredVariable`: Occurs when a variable is declared using an identifier that is already present in the existing environment;
- `InvalidLoc`: Occurs when attempting to access a memory location that is not defined in memory.
- ***`_EmptyEnvStack`****: Special exception used by the tracer. Occurs when attempting to access an empty stack of environments.*

To avoid throwing exceptions in Scala, which are hard to manage with Stainless, the various interpretation functions are defined to return either a set of exceptions or the actual result of the execution. We use the Stainless either, as follows: `Either[Set[LangException], T]`

5.2.3 Checker

The checker consists of a set of functions that traverse the AST once to verify some static properties of the program. Much like for Lean, these are not concerned with allocation, so they do not keep track of memory nor of any locations. A program is deemed valid only when it successfully passes all these checks.

```
def exprIsClosed(expr: Expr, env: Set[Name]):
      Boolean = expr match {
  case True              => true
  case False             => true
  case Nand(left, right) =>
    exprIsClosed(left, env) && exprIsClosed(
        right, env)
  case Ident(name)       => env.contains(name
      )
}
```

Listing 8: Expression closedness check

```
def stmtIsClosed(stmt: Stmt, env: Set[Name]): (Boolean,
     Set[Name]) =
stmt match {
  case Decl(name, value) => (exprIsClosed(value, env), env
       + name)
  case Assign(to, value) =>
    (env.contains(to) && exprIsClosed(value, env), env)
  case If(cond, body)    =>
    val (b, _) = stmtIsClosed(body, env)
    (exprIsClosed(cond, env) && b, env)
  case Seq(stmt1, stmt2) =>
    val (s1, menv) = stmtIsClosed(stmt1, env)
    val (s2, nenv) = stmtIsClosed(stmt2, menv)
    (s1 && s2, nenv)
  case Free(name)        => (env.contains(name), env)
  // --- Tracer ---
  case _Block(stmt0)     =>
    val (b, _) = stmtIsClosed(stmt0, env)
    (b, env)
}
```

Listing 9: Statement closedness check

The functions 8 and 9 verify, respectively, that the given expression or statement is closed. Similarly, `stmtHasNoRedeclarations` verifies that the given statement does not contain redeclarations, or `stmtHasNoBlocks` that does not contain blocks.

5.2.4 Interpreter

The interpreter is a function that, given as input the program and the initial state, executes the entire program and returns the final state.

```
def evalExpr(expr: Expr, state: State): Either[Set[LangException], Boolean]

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

It traverses the program tree once, modifying and/or propagating the current state to and from recursive calls, and short-circuiting when an exception is "thrown".

5.2.5 Interpreter implementation status

The interpreter is the first implementation we tackled, but we soon focused our efforts on the other two. Hence, it is the least developed of the three. The implementation supports all of the language constructs except `While`. This interpreter is the most similar in *form* (i.e. traversal, shape, logic, etc.) to the checker, so all indications are that it would produce the simplest proofs for our properties. However, almost none of the properties have been proved due to the main attention on the other implementations.

Closedness 3.1 is the only property that has been proved. It has a fairly simple inductive proof with a small difficulty. Both the checker and the interpreter check/execute both statements of a `Seq`, passing some results of the first call to the second. To prove closedness we need to prove the lemma that these two intermediate objects, i.e. the set of names and the state, are consistent.

```
case Seq(stmt1, stmt2) =>
  val (c1, mnames) = isStmtClosed(stmt1, names)
  val (c2, fnames) = isStmtClosed(stmt2, mnames)
  (c1 && c2, fnames)
```

```
case Seq(stmt1, stmt2) =>
  evalStmt(stmt1, state) match
    case Left(excep)   => Left(excep)
    case Right(mstate) => evalStmt(stmt2, mstate) match
      case Left(excep)   => Left(excep)
      case Right(fstate) => Right(fstate)
```

`Seq` case, checker on the left and interpreter on the right.

5.2.6 Tracer

Given a program, the tracer executes it statement by statement, going through a trace of states. The tracer consists of two functions: `evalStmt1` and `evalStmt`. The former executes a single step of computation, returning the updated state, while the latter invokes `evalStmt1` recursively.

```
def evalExpr(expr: Expr, state: State): Either[Set[LangException], Boolean]

def evalStmt1(stmt: Stmt, state: State, blocks: BigInt): Either[Set[LangException], Conf]
def evalStmt(stmt: Stmt, state: State): Either[Set[LangException], State]
```

We opted to implement the tracer to attain fine-grained control over the execution process. For instance, when confronted with infinite while statements, the big-step interpreter enters in a non-terminating state. In contrast, the tracer, with its function that executes a single statement and returns the subsequent state, ensures termination even in such scenarios.

For this reasons, the focal point of interest is on reason about properties of `evalStmt1`.

To enable the tracer's functionality, we introduce several changes and additions to the virtual model7. One prominent adjustment involves the evolution of the state tuple's environment tracking. It transforms from a single environment to a stack of environments implemented as a list. This modification proves crucial for managing scopes, as seen in if and while bodies where the environment atop the stack needs to be discarded to eliminate variables declared within it. To manage the environment stack, we introduced a 'synthetic' block in the AST 6. This block, not derivable from a program written in concrete syntax, is exclusively employed by the tracer for scope management. Another significant addition is the `Conf` type, that represents either a state or a pair composed of a statement and a state. Additionally, the function `evalStmt1` includes a new parameter that tracks the number of open blocks.

5.2.7 Tracer implementation status

The majority of our efforts were dedicated to implementing and verifying the tracer, which proved to be the most challenging path of the three. Initially, we anticipated a smoother process due to the increased control we had; however, the difference in *form* (i.e. traversal, shape, logic, etc.) between the checker and the tracer made it difficult to effectively utilizing the evidence provided by the checker in our proofs.

The small-step interpreter processes the program and outputs the next step and the state, effectively storing some state in the form of Blocks, which represents open scopes to be popped when the block will be closed. Additionally, it utilizes a more complex model involving the stack of environments. First we undertook the task of proving various properties to ensure the correct operation of the tracer with respect of the enhanced state and modifications applied to the program. These included establishing the monotonicity of the stack of environments with regards to the subset relation (ensuring that all elements from the previous environment in the stack are present in the current one), verifying the non-emptiness of the stack of environments, and ensuring consistency between the number of blocks and environments in the stack.

As the interpreter, the tracer supports all of the language constructs except `While`.

Given a program and a state, the execution of a single step of the tracer results in the next program and state in the sequence. Therefore, the program is changing at each step. Given the evidence from the checker, we can easily prove Closedness3.1 for a single step of computation. Since the checker is only applied to the initial program, we only have this evidence and can therefore apply the proof for the first step of the sequence. After that, the program has changed, so we no longer have the evidence we need. Therefore, in order to prove Closedness3.1 for the entire execution of the program (i.e. all steps of the tracer), we need to prove that the evidence initially given by the checker is preserved at each step of the computation. In this way we can inductively apply the above proof to prove the subsequent goal.

Most of the effort devoted to the tracer has been focused on proving this preservation lemma. At one point, we thought we had a proof for it, but we discovered a mutual recursion error in the proof that we overlooked because we had termination checking turned off due to other problems. Because of the very different nature of the checker and the tracer, it has proven extremely difficult to reconcile the evidence provided by the checker with our proofs. Ultimately, we have decided to state it as an axiom, hoping to prove it in the future.

A similar story is the absence of redeclarations 3.2. But this time we opted to directly state preservation as an axiom, recognizing the complexity involved.

Unique ownership 3.3 was also attempted. As opposed to other properties, its proof does not depend on evidence supplied by the checker but solely on the operation of the tracer. But it is again a preservation property, of the injectiveness of the environment. Progress has been made toward a first lemma that states that the allocator location is fresh (i.e., it is different from all locations in the environment/state), but it is not finished.

Finally, use-after-free 3.4 and no dangling pointers 3.5 were not attempted because we prioritized proving preservation lemmas, plus `Free` was added to this implementation relatively late.

6 Conclusions

In this project we defined a language and its semantics, defined a set of properties, and finally presented our approach to deriving correctness guarantees. We operated on a real implementation of that language rather than on an abstract definition of it.

6.1 Differences in approaches

We explored various approaches which we wish to now contrast. As mentioned at the beginning of the implementation section, the two most important differences were the choice of tool and interpretation style.

With Lean we enjoyed the fast development-feedback cycle where while writing proofs we are informed about whether the proof step is correct and what is the current goal to prove immediately. The proofs are precise so we do not have to worry about some potential nondeterministic factors. This is much different than our experience with Stainless, where we often encountered proofs being seen as valid tricking us into believing the proof is good enough. But after the cache is invalidated or cleared, our unchanged proof is no longer seen as valid. Fixing that required either hoping that eventually if left running for long enough Stainless would see it as valid once again, or debugging which exact condition is not easy to see for Stainless to then sprinkle some `assert`s to help Stainless see the condition indeed holds. This lead to losing trust in the cache and disabling it in favor of proofs that are more explicit to have them consistently being proven by Stainless. Combined with requiring a long time to give feedback about the proof correctness resulted in a very time consuming development cycle. On the other hand, Stainless was much more pleasant with its incredible power to prove all sorts of theorems without much of our help. In Lean that did not happen, and even the simplest properties needed a tedious proof.

The other difference is using a big-step and small-step style interpreter. The former is easier to define and to relate it to the checker, which is also written in a big-step style. However, as mentioned previously, it suffers from being non-terminating and thus conducting proofs on it end up potentially being non-terminating as well. These kind of proofs are rejected. The small-step style interpreter is more complicated, for instance requiring the usage of a stack of environments, and thus results in more complicated proofs. For the tracer many lemmas had to be proven about consistency and the relation with the checker before any interesting properties can be tackled.

6.2 Future work

Other than fully completing the goals that has been set, the language could be always extended with new constructs. The nature of requiring formal proofs of the implementation will force a careful design to see how a feature interacts with all properties. Finally more properties can be added, such as lack of memory leaks which the current operational semantics do not guarantee (leaving an if-statement drops the environment but keeps the memory causing a leak).

References

[1] Sam Blackshear et al. *The Move Borrow Checker*. 2022. arXiv: 2205.05181 [cs.PL].

[2] Ralf Jung et al. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154.

[3] *mathlib4 – The math library of Lean 4*. URL: https://github.com/leanprover-community/mathlib4.

[4] Bjarne Stroustrup. *Delivering Safe C++*. 2023. URL: https://youtu.be/I8UvQKvOSSw?t=163.