Verificación de programas en Elixir
Program Verification in Elixir

Trabajo de Fin de Máster
Curso 2021–2022

Autor
Adrián Enríquez Ballester

Director
Manuel Montenegro Montes

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Verificación de programas en Elixir
Program Verification in Elixir

Trabajo de Fin de Máster en Métodos Formales en Ingeniería Informática
Departamento de Sistemas Informáticos y computación

Autor
Adrián Enríquez Ballester

Director
Manuel Montenegro Montes

Convocatoria: *Junio 2022*
Calificación: *Nota*

Máster en Métodos Formales en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

21 de junio de 2022

Dedication

TODO

Acknowledgements

TODO

Resumen

Verificación de programas en Elixir

TODO

Un resumen en castellano de media página, incluyendo el título en castellano. A continuación, se escribirá una lista de no más de 10 palabras clave.

Palabras clave

Máximo 10 palabras clave separadas por comas

Abstract

Program Verification in Elixir

TODO

An abstract in English, half a page long, including the title in English. Below, a list with no more than 10 keywords.

Keywords

10 keywords max., separated by commas.

Contents

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Goals . . . 1
  - 1.2.1. Sub-goals . . . 1
  - 1.2.2. Non-goals . . . 1
- 1.3. Work plan . . . 2

2. State of the Art **3**
- 2.1. Program verification . . . 3
  - 2.1.1. Dafny . . . 3
  - 2.1.2. Intermediate representations . . . 4
  - 2.1.3. Intermediate representations for verification . . . 4
  - 2.1.4. SMT solvers . . . 4
- 2.2. Correctness in Elixir . . . 5
  - 2.2.1. Dynamic . . . 5
  - 2.2.2. Static . . . 5

3. Preliminaries **7**
- 3.1. Elixir . . . 7
  - 3.1.1. General description . . . 7
  - 3.1.2. Macros . . . 12
  - 3.1.3. Interoperability . . . 13
- 3.2. Satisfiability Modulo Theories . . . 13
  - 3.2.1. SMT-LIB . . . 14
  - 3.2.2. Z3 . . . 15

4. SMT Solver Integration in Elixir **17**
- 4.1. SMT-LIB interpreter binding . . . 17
  - 4.1.1. Overview . . . 17
  - 4.1.2. Implementation . . . 18
- 4.2. The L0 language . . . 20
  - 4.2.1. Notation . . . 20
  - 4.2.2. Syntax . . . 20
  - 4.2.3. Semantics . . . 21

4.2.4. Implementation 22

5. The L1 Intermediate Representation 25
5.1. Syntax 25
5.2. Semantics 26
5.2.1. Built-in declarations 26
5.2.2. Translation into L0 28
5.2.3. Built-in specifications 32
5.2.4. Term size modelling 34
5.3. Implementation 34
5.3.1. Overview 34
5.3.2. Details 35

6. Elixir Code Verification 39
6.1. The L2 verification language 39
6.1.1. Syntax 39
6.1.2. Translation into L1 40
6.1.3. Verifying L2 expressions 42
6.1.4. Verifying user-defined functions 43
6.1.5. Unfolding user-defined functions 44
6.1.6. Termination 45
6.2. Implementation 46
6.2.1. Overview 46
6.2.2. Translation 46
6.2.3. Verification 47
6.2.4. User-defined functions verification 48
6.2.5. User-defined functions unfolding 49

7. Conclusions and Future Work 51

Bibliography 53

A. DSL example: Hamiltonian path problem 55

B. Title of the Appendix B 59

Acronyms 61

List of figures

4.1. Big step operational semantics of the L0 language 21

List of tables

Chapter 1

Introduction

TODO: introduction. At some point explain the general idea, including a diagram, and include a guide referencing the chapters.

1.1. Motivation

TODO

1.2. Goals

The main goal of this project is to use the Elixir metaprogramming capabilities through macros to implement a code verification system for the Elixir programming language itself, without requiring us to modify its compiler or to implement a parser.

Our system will rely on a verification Intermediate Representation (IR) and the use of SMT solvers for its verification.

1.2.1. Sub-goals

In order to achieve the main goal, we have proposed several possible sub-goals. One of them is to integrate SMT solvers in Elixir with a Domain Specific Language (DSL) of macros like in the following draft:

Another one is to develop a verification IR to express Elixir terms and its dynamically typed nature:

Then, we must define a translation from this IR into the SMT solver DSL for its verification and, finally, we must also provide a mechanism to translate a subset of the Elixir programming language into the verification IR.

1.2.2. Non-goals

We have left some points as possible future work. The first one is that we are going to deal only with sequential Elixir programs and not with concurrent ones. Even for the sequential Elixir part we are going to support only a small subset to start with.

Also, we are going to deal only with partial verification for the moment and not to verify or reason about termination. Nevertheless, we will discuss some ideas regarding it

when introducing user-defined verification functions and how the system allows to unfold their invocations.

1.3. Work plan

As a work plan, we will follow the sub-goals in the same order in which we have mentioned them, preceded by a training period for acquiring the required knowledge and practice with the Elixir programming language.

Chapter 2

State of the Art

"Don't you want to be a person who inspires others the way you were inspired by something?"
— Fredrik Thordendal

Our main goal is to provide a static verification mechanism to allow Elixir programmers to write formal specifications and prove the conformance of their code in regard to these specifications.

In this section, we will discuss the current approaches and tools for that purpose in general, and then which are the current approaches to prove or disprove correctness in Elixir.

2.1. Program verification

A usual approach for program verification is to transform code and specifications into an intermediate representation, such as Boogie, and then a theorem prover tries to prove its verification conditions, where their validity implies the correctness under consideration (Leino, 2008).

In general, a theorem prover may not be able to reach a required proof, although it may exist, and human intervention can be necessary in the form of interacting with an interface, transforming the code to be verified or adding information to help the prover.

2.1.1. Dafny

Dafny is a programming language that provides features for program verification and covers several programming paradigms, such as imperative, functional and Object-oriented programming (OOP) (Ford and Leino, 2017). It was created at Microsoft Research and is currently being developed with the support of Amazon.

The following is an example to specify and verify the implementation of a function to return the maximum of three integer numbers:

```
method max3(x: int, y: int, z: int) returns (m: int)
  ensures m == x || m == y || m == z
  ensures m >= x && m >= y && m >= y
{
  if x > y && x > z { return x; }
```

```
  if y > z { return y; }
  return z;
}
```

Although the above example may seem simple, Dafny can also handle more advanced topics such as recursion and loops by means of induction and loop invariants respectively. Once verified, the code can be translated, erasing everything that is related only to its verification, into other programming languages such as C# to be executed.

Our project is greatly inspired by this system, although our aim is to embed program verification features into an existing programming language instead.

2.1.2. Intermediate representations

IRs tend to arise in compiler technologies, with goals like perform analyses, optimizations or portability (Zhao et al., 2012). A richer language can be translated into a simpler or more focused one, the IR, which can also be the translation target for other languages.

A known technology which provides a platform-independent IR intended to be executed in different platforms, and a toolchain to work with it, is LLVM (Lattner and Adve, 2002).

Programming languages such as Java and Erlang also have as a compilation target a bytecode IR corresponding to their virtual machines, JVM and BEAM respectively. These are also compilation targets for other programming languages such as Kotlin and Scala for the JVM and Elixir for the BEAM virtual machine. Also, WebAssembly is an IR intended to be executable at native speed in web browsers.

2.1.3. Intermediate representations for verification

For building verification tools, we are interested on IRs that are focused in capturing the intended verification notions and are suitable to be transformed into an input for a theorem prover. This last one will try to provide a proof for the verification.

Apart from Boogie2, which is the verification IR used in Dafny and offers features to model a wide range of programming paradigms, there are other ones as Verification Infrastructure for Permission-based Reasoning (Viper), that is a suite of tools which provides an intermediate verification language with the same name and allows reasoning about the program state using *permissions* or *ownership*. It allows implementing verification techniques for sequential and concurrent programs with mutable state (Müller et al., 2016).

Also, Why3 is a platform for deductive program verification that provides a language for specification and programming, WhyML, that can be used as an IR for the verification of C, Java, Ada, or to obtain an automated correct-by-construction OCaml program (Bobot et al., 2022).

Other approaches try to use logic programs as an intermediate representation throughout its regular compilation process, as in Gange et al. (2015), and other ones put an effort into being a suitable target for modelling faithfully the semantics of different programming languages, as in Montenegro et al. (2015).

2.1.4. SMT solvers

At the bottom of the verification process, a theorem prover will try to obtain a proof for some verification conditions. SMT solvers in particular are gaining popularity for this task, and some current options are Z3 from Microsoft, Z3, CVC4, MathSAT and Yices.

There is also an international initiative, called SMT-LIB, aimed at facilitating research and development in Satisfiability Modulo Theories (SMT) (Barrett et al., 2017). We will discuss SMT, SMT-LIB and Z3 in the Chapter 3 as these are tools that are going to be used in our project.

2.2. Correctness in Elixir

The current approaches for proving or disproving the correctness of Elixir programs are in general inherited from Erlang, and they succeed more in disproving than in proving correctness.

2.2.1. Dynamic

Erlang provides several libraries for property-based testing such as Term Reductive Invariant Questant (Triq), PROPerty-based testing tool for ERlang (PropEr) and Erlang QuickCheck. Many of them are offered for Elixir through wrapper packages as `ExCheck` for Triq and `PropCheck` for PropEr.

There are also packages implemented completely in Elixir, as `StreamData` and `TypeCheck`. This last one tries to take profit of the type specification mechanism of Elixir in order to automatically provide data generators.

In this case, their goal is to disprove the correctness of the program by finding counterexamples for specified properties.

2.2.2. Static

Elixir allows to annotate the intended types for function parameters and return values, and a tool called DIscrepancy AnaLYZer for ERlang programs (Dialyzer) (Lindahl, 2012) can perform static analysis on them. A problem detected by this tool can disprove correctness, but it does not prove correctness, which is our aim in this project.

There have been other attempts in offering tools that provide correctness proofs for Erlang programs, as in Fredlund et al. (2003).

Chapter 3

Preliminaries

This chapter introduces some required topics and tools that are a basis to our project.

On the one hand, Elixir is the programming language that is the verification subject of this document and, at the same time, the one in which our implementation has been coded.

On the other hand, our verification system relies on the SMT problem and its encoding in SMT-LIB, a standard language and interface to interact with theorem provers such as Z3.

3.1. Elixir

Elixir is a general-purpose programming language that runs on the Erlang Virtual Machine, also called BEAM, where also programs written in the Erlang language run. Both of them share some features, like their actor-based concurrency model, and have a native capability to interoperate between them. Although Elixir is younger than Erlang, this has allowed the former to be part of an ecosystem which has been developed across more than three decades.

We have chosen such a programming language for this research because, first, it is a modern programming language ready to be used in the industry. Second, it has the unusual property in formal verification to be dynamically typed, but its functional programming principles will make it easier to reason about. Finally, its metaprogramming capabilities will allow us to extend it according to our needs without requiring us to modify its compiler.

3.1.1. General description

In this section, we introduce the basic concepts and constructs of sequential programming in Elixir. Our aim is to show only the behavior of the language subset that is studied later in this document for its verification, and also its metaprogramming mechanism based on macros, on top of which our proposed verification system has been implemented.

The following examples will be shown in the Elixir Read-Eval-Print-Loop (REPL), called `iex`, where `iex>` represents its default prompt and an introduced expression is followed by the result such that it evaluates:

```
iex> "Hello world"
"Hello world"
```

3.1.1.1. Value types

As usual, one of the core value types in Elixir is `integer`, for which arithmetic operators behave as expected:

```
iex> (2 + 2) * 5
20
iex> -1
-1
iex> 1 / 0
** (ArithmeticError)
```

The `boolean` value type is also at its core, but it is worth mentioning the semantics of its operators when involving non-`boolean` types, and also with respect to short-circuit evaluation:

```
iex> true and 2 # Evaluates to the second argument
2
iex> 2 and true # Requires the first one to be a boolean
** (BadBooleanError)
iex> false and 1 / 0 # Does not evaluate the second
   argument
false
```

Some built-in Elixir functions allow checking if a given value is of a given type by returning a `boolean` result:

```
iex> is_boolean(true)
true
iex> is_boolean(2)
false
iex> is_integer(2)
true
```

Equality and comparison operators also evaluate to `boolean` values and allow mixing types:

```
iex> 2 === 2
true
iex> 2 === true
false
iex> 2 !== true
true
iex> 2 > 1
true
iex> 2 < true
true
```

The `===`, `!==`, `and` and `or` operators are the so-called *strict* version of their respective counterparts `==`, `!=`, `&&` and `||`, but we are not going to deal with them for the moment.

Also, `boolean` values are in fact a special case for `atom` values, but we are not going to deal with that value type for the moment in this project.

3.1.1.2. Collection types

One of the simplest built-in collection types in Elixir is the inductive `list`, which consists of nested cons cells (i.e. pairs) and can be written in different ways:

```
iex> [] # The empty list
[]
iex> [3 | []] # A cons cell
[3]
iex> [1 | [2 | [3 | []]]] # Nested cons cells
[1, 2, 3]
iex> [1, 2, 3] # Syntactic sugar
[1, 2, 3]
iex> [1, 2 | [3]] # Mixing sugared and desugared syntax
[1, 2, 3]
```

It is not required for the `list` elements to be of the same type (i.e. heterogeneous lists are allowed), and improper lists (i.e. those that do not have an empty list as the second element in the deepest cons cell) are also allowed (Eli, 2022):

```
iex> [1, 2, false] # An heterogeneous list
[1, 2, false]
iex> [1 | [2 | 3]] # An improper list
[1, 2 | 3]
```

Functions in Elixir are commonly referred by its name and arity. The `hd/1` and `tl/1` built-in functions for lists allow us to respectively obtain the first and second components of a cons cell:

```
iex> hd([1, 2, false])
1
iex> tl([1, 2, false])
[2, false]
iex> hd([])
** (ArgumentError)
iex> tl([])
** (ArgumentError)
```

There is also a function for checking the `list` type membership. Consider the following code to apply the `is_list/1` function to several provided lists and return the conjunction of its results:

```
iex> Enum.all?(                  # Are all
  [[], [1, 2], [1, 2 | false]], # of these
  &is_list/1                     # lists?
)
true                             # Yes
```

Another core collection type in Elixir is the `tuple`, which also does not restrict its elements to be of the same type:

```
iex> {} # The empty tuple
{}
iex> {1, false, {3, 4}, []}
{1, false, {3, 4}, []}
```

Tuples have a size, which can be retrieved with the `tuple_size/1` function, and each `tuple` component can also be retrieved with the `elem/2` function by specifying its position with a zero-based index:

```
iex> tuple_size({1, 2, 3})
3
iex> elem({1, 2, 3}, 0)
1
iex> elem({1, 2, 3}, 2)
3
iex> elem({1, 2, 3}, 3)
** (ArgumentError)
```

In this case, the `tuple` type membership checking function is `is_tuple/1`. Usually, the components of a collection such as a `list` or a `tuple` are obtained by means of pattern matching, that is explained in the following section.

3.1.1.3. Blocks, pattern matching and control flow

Elixir expressions can be evaluated sequentially by gathering them inside a `block`, delimited by a semicolon or a line break:

```
iex> 2 + 1;5 === 5;false
false # Evaluates to the result of its last expressions
iex> (
        2 + 1
        5 === 5
        false
     )
false
```

The expressions inside a `block` that are not the last one tend to perform side effects, such as binding values to variable names with the `match` operator `=`. Note that these bindings are not locally scoped inside `block`s and, in contrast to Erlang, variable bindings can be overridden:

```
iex> (z = 2, 4)
4
iex> z
2
iex> z = 3
3
```

This operator also allows performing pattern matching, which destructures expressions according to patterns in order to check for a given shape and bind subexpressions to variable names. They are particularly useful for dealing with collection value types:

```
iex> {x, 3} = {2, 3}
{2, 3}
iex> x
2
iex> {x, 3} = {2, 4}
** (MatchError)
iex> [h | t = [_, 3]] = [1, 2, 3] # A nested match
```

```
[1, 2, 3]
iex> h
1
iex> t
[2, 3]
```

Regarding control flow, although Elixir provides usual constructs such as `if`, one of the most general ones is `case`. It is evaluated to the first branch that matches the pattern and is compliant with a guard expression if specified, and this is the only branch in the `case` that is evaluated:

```
iex> case {1, 2, 3} do
       {}                              -> 1 / 0
       {1, x, 3} when is_integer(x) -> x + 1
       {1, 2, 3}                       -> false
     end
3
iex> x
** (CompileError) # The case bindings are local
iex> case 2 do
       false -> 3
     end
** (CaseClauseError) # No pattern matches the expression
```

Guards have a restricted syntax, allowing for example comparison, boolean negation, conjunction and disjunction, and type checking for values.

3.1.1.4. Function definitions

A named function, identified by its name and arity, can be defined inside a `module` with different body definitions and different patterns and guards for its arguments:

```
defmodule Example do
  def fact(0) do
    1
  end

  def fact(n) when is_integer(n) and n > 0 do
    n * fact(n - 1) # Recursion is allowed
  end
end
```

The rules that determine which clause is applied is are the same as in `case` expressions, so function definitions can also express control flow (Thomas, 2018).

3.1.1.5. Type specifications

Although Elixir is dynamically typed, it has a system to annotate the intended types for functions and a tool to perform a static analysis on them, which is called Dialyzer (Lindahl, 2012). We will use these specifications together with function identifiers to outline the ideas behind our implementations along this document.

A function type specification can be defined as follows:

```
@spec function_name(type_1, type_2, ... type_n) ::
   return_type
```

Types can be defined by means of composing other types with constructs such as the `|` operator, which denotes the union of types:

```
@type tuple_or_nat :: tuple | non_neg_integer
```

3.1.2. Macros

Because of its metaprogramming capabilities based on macros, Elixir is a suitable language for implementing DSLs (McCord, 2015). This will allow us to extend it without requiring us to modify the Elixir compiler or implement a parser.

The main construct for this purpose is `defmacro` which, as a curiosity, is declared in the `Kernel` module of Elixir in terms of itself due to a bootstrapping process:

```
defmacro defmacro(call, expr) do
  ...
```

The argument values for a macro are Elixir Abstract Syntax Tree (AST)s, and its return value must also be a valid Elixir AST that will replace the macro invocation at compile-time. The resulting code may also contain other macro calls that will be expanded recursively.

By using type specifications, the AST type for Elixir expressions is defined in the `Macro` module as

```
@type ast ::
  atom
  | number
  | [ast]
  | {ast, ast}
  | ast_expr

@type ast_expr :: {ast_expr | atom, metadata, atom | [ast]}
```

where `ast_expr` represents a function invocation when the first component is the function name, and the third one its arguments. We can obtain the AST corresponding to an Elixir expression with the `quote/1` macro:

```
iex> quote do
       1 + 2
     end
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

`quote/1` is the main construct to transform the input AST into new AST when defining a macro, together with `unquote/1` to interpolate expressions inside a quoted one:

```
defmacro sum_into_product({:+, _, [x, y]}) do
  quote do
    unquote(x) * unquote(y)
  end
end
```

Elixir also offers several advanced constructs to deal with macros, such as `unquote_splicing/1` to interpolate an AST list as the arguments of a function invocation:

```
iex> quote do
       hello(unquote_splicing([2, 3]))
     end
{:hello, [], [2, 3]}
```

In this project, we have preferred to implement regular Elixir functions, some of them that transform Elixir AST, and to use these to implement macros only at the top level `module` of our packages in a simple and controlled manner due to the following reasons:

- In general, it is harder to reason about macros than about regular functions.
- These regular functions can be easily reused to provide different Application Programming Interface (API)s built on top of them, either by defining macros or not.
- In this way, the code generated by our macros is smaller because their expansion reuses regular function invocations at run-time.
- If a macro expands to other macros, the user has to import also that macros.

3.1.3. Interoperability

Elixir offers several ways to interoperate with processes or libraries that are external to the Erlang Virtual Machine, apart from conventional Input/Output (I/O) based mechanisms. We are interested in these features due to the integration of an SMT solver in Elixir, which will surely be an external process.

One of these ways is Native Implemented Function (NIF)s, which allows loading and calling libraries implemented in other programming languages such as C. When using this system, it is important to know that a crash in a NIF brings the Erlang Virtual Machine down too (Erl, 2022).

A safer approach is to launch an external process managed by the Erlang Virtual Machine and communicate with it by means of message passing, which in Elixir is provided by a mechanism called *ports*:

```
port = Port.open({:spawn, "cat"}, [:binary])
iex> send(port, {self(), {:command, "hello"}})
iex> flush()
# Received from the process
{#Port<0.1444>, {:data, "hello"}}
send(port, {self(), :close})
```

The underlying implementation makes the communication through `stdin` and `stdout`, but this is abstracted under the message passing API.

A known drawback of this mechanism is that, if the Erlang Virtual Machine crashes after having launched a long-running process, then its `stdin` and `stdout` channels will be closed, but it won't be automatically terminated. This depends on how the specific process behaves when its communication channels are closed (Eli, 2022).

3.2. Satisfiability Modulo Theories

The SMT problem consists in checking whether a given logical formula is satisfiable within a specific theory (Barrett et al., 2017). This allows a theorem prover to define theories in which the SMT problem is decidable and, moreover, to design efficient algorithms specialized in solving this problem for a theory or a set of them.

An example of a theory is that of linear integer arithmetic, which restricts the allowed functions, predicates and constants to be from the signature $\{=, +, \leq, 0, 1\}$ with the usual axioms for equality, order and addition.

The SMT problem is decidable for quantifier free fragments of an arbitrary theory $\mathcal{T}$ by an algorithm called DPLL($\mathcal{T}$) that combines a decision procedure for that specific theory with an algorithm to solve the Boolean SATisfiability problem (SAT) problem, such as CDCL.

3.2.1. SMT-LIB

SMT-LIB is an initiative which tries to provide a common interface to interact with SMT solvers. It defines a solver-agnostic standard language with a Lisp-like syntax to configure a solver, manage it, encode an SMT problem instance and query for solutions.

The terms and formulas in this language correspond to a sorted(i.e. typed) version of first-order logic (Barrett et al., 2017), and this is a subset of the syntax of allowed commands that we are going to use:

⟨ *command* ⟩ ::= `(` `assert` ⟨ *term* ⟩ `)`
| `(` `pop` ⟨ *numeral* ⟩ `)`
| `(` `push` ⟨ *numeral* ⟩ `)`
| `(` `check-sat` `)`
| `(` `declare-sort` ⟨ *symbol* ⟩ ⟨ *numeral* ⟩ `)`
| `(` `declare-const` ⟨ *symbol* ⟩ ⟨ *sort* ⟩ `)`
| `(` `declare-fun` ⟨ *symbol* ⟩ `(` ⟨ *symbol* ⟩* `)` ⟨ *sort* ⟩ `)`

As a brief description for these commands, `assert` adds a well-sorted formula to the current assertion level, `pop` and `push` respectively add and remove an assertion level from the stack. `check-sat` asks the solver whether the formulas from the stack are satisfiable or not. `declare-sort`, `declare-const` and `declare-fun` allow declaring new sorts, functions or constants (i.e. nullary functions).

The command responses are defined by the following syntax:

⟨ *general_response* ⟩ ::= `success`
| `unsupported`
| `(` `error` ⟨ *string* ⟩ `)`
| ⟨ *specific_success_response* ⟩

where some commands have a specific success response, like `sat` | `unsat` | `unknown` for `check-sat`.

This would be an example for checking the validity of the linear integer arithmetic formula $x + 3 \leq y + 3 \Rightarrow x \leq y$:

```
(declare-const x Int)
(declare-const y Int)

; We add the negated formula
(assert (not (=>
  (<= (+ x 3) (+ y 3))
  (<= x y)
)))
```

```
; and check if it is unsatisfiable
(check-sat)
```

We are going to check it in the following section.

3.2.2. Z3

One of the SMT solvers that implements the SMT-LIB standard is the Z3 theorem prover from Microsoft Research. It can be started with its `stdin` and `stdout` as communication channels by launching the `z3` executable with the `-in` argument:

```
z3 -in
# Copy and paste the previous SMT-LIB example...
unsat
```

Note that there may exist subtle non-compliances when a solver implements the SMT-LIB standard. For example, we have found that Z3 does not include the surrounding double-quotes when it prints back the provided string literal, which is the specified behavior in the standard.

This may lead to confusion because the `echo` command is the only one whose response is a string literal and, as this is not the case for Z3, there are corner cases in which a command response can be confused with a printed string intended to delimit command responses, which is one of the proposed usages for `echo` in Barrett et al. (2017):

```
$ z3 -in <<<‘(check-sat) (echo "sat")’
sat
sat
```

We are aware of this because we have used such technique to delimit the command responses in order to parse them.

Chapter 4

SMT Solver Integration in Elixir

In order to implement our system, we will require to be able to interact with an SMT solver from Elixir. We have decided to use the Z3 theorem prover, which implements SMT-LIB, and to communicate with it precisely by using this standard. This makes possible for our system to allow integrating other SMT solvers or to provide different communication implementations without requiring so much effort.

Then, we will introduce a simple formal language whose semantics is defined in terms of the SMT problem, and an example of its implementation in Elixir as a result of the previous integration with the solver.

4.1. SMT-LIB interpreter binding

As we did not find any existing Elixir package that met our requirements explained above, except for one that seemed to be unmaintained and not ready for a general-purpose usage, we addressed the implementation of an SMT-LIB interpreter binding as an opportunity to get started with Elixir in practice, and also with its macro system. This has given place to a side project which consists of an Elixir DSL to communicate with SMT-LIB interpreters, and may be eventually provided as a general-purpose library to be available for the Elixir community.

4.1.1. Overview

By using our solver DSL, the SMT-LIB example shown in Section 3.2.1 can be written in Elixir as follows:

```
import SmtLib

with_local_conn do
  declare_const x: Int,
                y: Int

  assert !((:x + 3 <= :y + 3) ~> (:x <= :y))
  case check_sat do
    {:ok, :unsat} -> IO.puts("Verified!")
    _             -> IO.puts("Not verified")
  end
```

```
end
```

which prints `"Verified!"`, proving that

$$x + 3 \leq y + 3 \Rightarrow x \leq y$$

We provide a `with_local_conn/1` macro that creates a default fresh connection with Z3 through ports, injects it as the first argument to its inner SMT-LIB command DSL macros (e.g. `declare_const/2` and `assert/2`) and closes it once all of them have been evaluated. It is a convenient wrapper around another macro, namely `with_conn/2`, which allows to provide a custom or reused connection and does not close it automatically.

Regarding the contents of a `with_conn/2` block, our DSL currently supports a subset of SMT-LIB commands that is shown in the following section, but it is almost trivial to add support for new ones, being the biggest deal to parse its response if it has a specific one. The expressions corresponding to logical formulas include support for variables, uninterpreted function applications, quantifiers, and built-in operators and logic connectives such as `+`, `!` and `&&`.

A longer example of using this binding to solve a Constraint Satisfaction Problem (CSP) is shown in Appendix A.

4.1.2. Implementation

As explained in Section 3.1.1.5, we will use Elixir type specifications as a guide to explain our implementation in a simplified form. Also, we will prefer to implement regular Elixir functions and then define macros only in the top level module by making use of this regular functions.

First, we have defined `type`s to represent SMT-LIB commands and responses from the subset that we have shown in Section 3.2.1:

```
@type command_t ::
  {:assert, term_t}
  | :check_sat
  | {:push, numeral_t}
  | {:pop, numeral_t}
  | {:declare_const, symbol_t, sort_t}
  | {:declare_sort, symbol_t, numeral_t}
  | {:declare_fun, symbol_t, [sort_t], sort_t}

@type general_response_t ::
  :success
  | :unsupported
  | {:error, string_t}
  | {:specific_success_response,
     specific_success_response_t}
```

where other involved types like `numeral_t` and `sort_t` are defined similarly, many of them as an alias to built-in Elixir value types.

Then, we have implemented a function that, given a subset of the Elixir AST (i.e. our DSL), transforms it into a list of SMT-LIB commands:

```
@spec ast_to_commands(ast) :: [command_t]
```

Its implementation defines cases for each possible term and subterms, like the following for the `declare-const` command:

```
@spec ast_to_command(SmtLib.ast) :: command_t
def ast_to_command({:declare_const, _, [{v, s}]}) do
  {:declare_const, symbol(v), sort(s)}
end
```

In fact, our implementation is a bit more complicated because it allows using run-time values bound to Elixir variables in some places, such as constants and identifiers, thus in our case it translates Elixir AST corresponding to our DSL into other Elixir AST that evaluates to SMT-LIB commands.

Once we were able to transform the DSL into SMT-LIB commands, we required a function to render each command into a `string` that an SMT-LIB interpreter understands:

```
@spec command_to_string(command_t) :: String.t
```

This function handles compositionally the `command_t` type with cases like the following:

```
{:declare_const, s1, s2} ->
  "(declare-const #{symbol(s1)} #{sort(s2)})"
```

Besides this, in order to understand the solver responses, we have also implemented a function that parses a received `string`:

```
@spec general_response_from_string(String.t) ::
  {:ok, general_response_t}
  | {:error, term}
```

This function has been implemented using `NimbleParsec`, an Elixir package of parser combinators, in order to delegate this task and get the reliability of a well tested tool (Nim, 2022). Its top level parser definition is as follows:

```
defparsec :general_response,
  skip_blanks_and_comments()
  |> choice([
    token(success()) |> eos(),
    token(unsupported()) |> eos(),
    token(error()) |> eos(),
    token(specific_success_response()) |> eos()
  ])
```

where, for example, `eos/1` is a combinator that denotes the end of the stream of data being parsed, `success/1` parses the success response of SMT-LIB commands that do not have a specific one, and `specific_success_response/1` parses specific responses such as the one for `check_sat`.

Finally, to interact with the solver, we have defined an Elixir low level protocol to send SMT-LIB commands to a solver and receive SMT-LIB responses in a synchronous way:

```
defprotocol Connection do
  @spec send_command(t, command_t) ::
    :ok | {:error, term}
  def send_command(connection, command)

  @spec receive_response(t)
```

```
      :: {:ok, general_response_t} | {:error, term}
    def receive_response(connection)
  end
```

We provide a default implementation of this protocol to communicate with Z3 through ports and allow the user to configure some of its parameters like the timeout, but other implementations involving different solvers and communication mechanisms should also be possible.

All of this makes possible to implement and provide the public API of the package under a top level module that defines the corresponding DSL macros and can be used as in Section 4.1.1.

4.2. The L0 language

This section introduces a formal language that we have named L0. The name stands for 'Level 0', since it is the lowest level language of our verification stack.

L0 is intended to be implemented as Elixir expressions that send SMT-LIB commands to an SMT solver. This will allow us to define a verification IR on top of it.

4.2.1. Notation

We assume that $\mathbb{F}$ is the set of many-sorted logic formulae involving equality, uninterpreted function symbols and arithmetic. We use φ, ψ, etc. to denote elements from this set.

Also, we assume a set Σ^0 of uninterpreted function symbols and a set $\mathbb{T}$ of terms in many-sorted logic, generated by the following grammar:

$$\mathbb{T} \ni t ::= n \mid x \mid f(t_1, \ldots, t_m)$$

where n is a number, x is a variable, and $f \in \Sigma^0$ is a function symbol of arity m.

4.2.2. Syntax

The syntax of L0 expressions is given by the following grammar:

$\mathbf{Exp}^0 \ni \epsilon$	::=	**skip**	{do nothing}
	|	**fail**	{fail signal}
	|	$\epsilon_1; \epsilon_2$	{sequential evaluation}
	|	**local** ϵ	{local scoped proof state}
	|	**add** φ	{add a logic formula $\varphi \in \mathbb{F}$ to the state}
	|	**declare** x	{declare a variable of type *Term*}
	|	**when-unsat** ϵ_1 **do** ϵ_2 **else** ϵ_3	{unsatisfiability conditional}

where *Term* is a sort that must be previously defined in the SMT solver.

If $I = [i_1, \ldots, i_n]$ is a sequence of elements, we use the notation $\overline{\epsilon_i}^{i \in I}$ to denote the sequential composition $\epsilon_{i_1}; \ldots; \epsilon_{i_n}$.

As we will reflect in the language semantics, the **fail** expression will allow us to trigger a validation failure, the **declare** x and **add** φ ones will allow us to respectively declare a new variable and to add a formula to the state, the **local** one to evaluate an expression without incorporating its changes in the final state, and **when-unsat** will be a control flow

$$\frac{}{\langle \textbf{skip},\ X,\ \Phi\rangle \Downarrow (X, \Phi)}$$

$$\frac{\varphi \in \mathbb{F}(X)}{\langle \textbf{add}\ \varphi,\ X,\ \Phi\rangle \Downarrow (X, \Phi \cup \{\varphi\})} \qquad \frac{x \notin X}{\langle \textbf{declare}\ x,\ X,\ \Phi\rangle \Downarrow (X \cup \{x\}, \Phi)}$$

$$\frac{\langle \epsilon_1,\ X,\ \Phi\rangle \Downarrow (X', \Phi') \qquad \langle \epsilon_2,\ X',\ \Phi'\rangle \Downarrow (X'', \Phi'')}{\langle \epsilon_1; \epsilon_2,\ X,\ \Phi\rangle \Downarrow (X'', \Phi'')}$$

$$\frac{\langle \epsilon,\ X,\ \Phi\rangle \Downarrow (X', \Phi')}{\langle \textbf{local}\ \epsilon,\ X,\ \Phi\rangle \Downarrow (X, \Phi)}$$

$$\frac{\langle \epsilon_1,\ X,\ \Phi\rangle \Downarrow (X', \Phi') \qquad \mathit{unsat}(\Phi') \qquad \langle \epsilon_2,\ X,\ \Phi\rangle \Downarrow (X'', \Phi'')}{\langle \textbf{when-unsat}\ \epsilon_1\ \textbf{do}\ \epsilon_2\ \textbf{else}\ \epsilon_3,\ X,\ \Phi\rangle \Downarrow (X'', \Phi'')}$$

$$\frac{\langle \epsilon_1,\ X,\ \Phi\rangle \Downarrow (X', \Phi') \qquad \neg\mathit{unsat}(\Phi') \qquad \langle \epsilon_3,\ X,\ \Phi\rangle \Downarrow (X'', \Phi'')}{\langle \textbf{when-unsat}\ \epsilon_1\ \textbf{do}\ \epsilon_2\ \textbf{else}\ \epsilon_3,\ X,\ \Phi\rangle \Downarrow (X'', \Phi'')}$$

Figure 4.1: Big step operational semantics of the L0 language

construct that depends on the unsatisfiability of a given expression. Sequential evaluation will combine expressions to be evaluated one after the other, and **skip** will be useful for example if some **when-unsat** branch requires doing nothing.

4.2.3. Semantics

Let V be a set of variable names, $\mathbb{F}(V)$ the subset of $\mathbb{F}$ with free variables in V, and a predicate *unsat* which, given a set of formulas Φ from $\mathbb{F}$, determines whether they are unsatisfiable or not. We define the big step operational semantics of L0 expressions as the smallest relation $\langle \epsilon,\ X,\ \Phi\rangle \Downarrow (X', \Phi')$ between $\mathbf{Exp}^0 \times \mathcal{P}(V) \times \mathcal{P}(\mathbb{F}(V))$ and $\mathcal{P}(V) \times \mathcal{P}(\mathbb{F}(V))$ that satisfies the rules from Figure 4.1.

A pair (X, Φ) denotes the state of the SMT solver, where X is the set of variable names defined at the moment, and Φ the set of formulas that have been added also at that moment. A judgement $\langle \epsilon,\ X,\ \Phi\rangle \Downarrow (X', \Phi')$ means that the expression ϵ transforms the solver state (X, Φ) into the state (X', Φ').

The absence of rules for the **fail** expression is intentional, because we want any reachable **fail** to prevent evaluation of the expression in which it is contained. We have also required this to happen if the same variable is declared twice or if a formula with undeclared variables is being added to the solver's state.

Note also that a **local** expression discards the variable definitions and formulas added by its expression, restoring the initial state, and the same happens for the expression under the unsatisfiability test in a **when-unsat** expression.

4.2.4. Implementation

It is difficult to justify the compliance of an implementation of L0 with its formal semantics due to the undecidability of the SMT problem in the general case. In practice, this task will be delegated to an SMT solver as if it were a black box that can determine whether a set of first-order formulas is unsatisfiable.

We can implement a simple Elixir DSL for the L0 language in terms of our SMT-LIB binding for Elixir. The `fail` expression raises an exception:

```
defmacro eval(_, {:fail, _, _}) do
  quote do
    raise "Verification failed"
  end
end
```

The `local` expression surrounds the evaluation in between `pop` and `push` SMT-LIB commands:

```
defmacro eval(conn, {:local, _, [e]}) do
  quote do
    conn = unquote(conn)
    :ok = push conn
    eval conn, unquote(e)
    :ok = pop conn
  end
end
```

The `add` expression corresponds to an `assert` in SMT-LIB:

```
defmacro eval(conn, {:add, _, [f]}) do
  quote do
    conn = unquote(conn)
    :ok = assert conn, unquote(f)
  end
end
```

Similarly, the `declare_const` expression corresponds to a `declare-const` in SMT-LIB:

```
defmacro eval(conn, {:declare_const, _, [x]}) do
  quote do
    conn = unquote(conn)
    :ok = declare_const conn, [{unquote(x), Term}]
  end
end
```

The `when-unsat` expression implementation is slightly longer:

```
defmacro eval(
  conn,
  {:when_unsat, _, [e1, [do: e2, else: e3]]}
) do
  quote do
    conn = unquote(conn)
    :ok = push conn
    eval conn, unquote(e1)
```

```
    {:ok, result} = check_sat conn
    :ok = pop conn

    case result do
      :unsat -> eval conn, unquote(e2)
      _ -> eval conn, unquote(e3)
    end
  end
end
```

Finally, instead of implementing a `seq` expression, we can reuse Elixir blocks by handling several cases, and allowing to provide them with `do` syntax (i.e. to write an Elixir block argument within a trailing `do` and `end` delimiters in a macro invocation):

```
defmacro eval(
  conn,
  do: {:__block__, [], []}
) when is_list(es) do
  nil
end

defmacro eval(
  conn,
  do: {:__block__, [], [e | es]}
) when is_list(es) do
  quote do
    conn = unquote(conn)
    eval conn, unquote(e)
    eval conn, unquote({:__block__, [], [es]})
  end
end

defmacro eval(conn, do: e) do
  quote do
    conn = unquote(conn)
    eval conn, unquote(e)
  end
end
```

We can also include a general case that raises an exception if the provided Elixir AST does not correspond to our language:

```
defmacro eval(_, other) do
  raise "Unknown expression #{Macro.to_string(other)}"
end
```

Assuming the defined macros to be in scope, and a `conn` variable that represents a fresh connection with an SMT solver which has the `Term` sort already defined, this would be a simple example of the usage of the `eval/1` macro:

```
eval conn do
  declare_const :x
```

```
    # Replacing '!=' by '==' leads to a verification
    # exception, since 'fail' is executed
    when_unsat add :x != :x do
      skip
    else
      fail
    end
  end
```

Chapter 5

The L1 Intermediate Representation

"This process of using tools you built yesterday to help build bigger tools today is called abstraction, and it is the most powerful force I know of in the universe"
— Sandy Maguire

In this chapter, we develop an IR for verification that is intended to be particularly suitable for languages like Elixir.

We start by providing its formal syntax, its translation into L0 expressions and the built-in prelude that we have defined for modelling the Elixir semantics. Then, we show some examples with our current implementation and give an overview of its details.

5.1. Syntax

We denote by $\Sigma^1 = \{\texttt{===}, \texttt{<=}, \texttt{>=}, \texttt{+}, \texttt{-}, \dots\}$ the set of operators and functions allowed in L1. We assume that for every element $f \in \Sigma^1$ there is an uninterpreted function symbol in Σ^0, which will be denoted by $\widehat{f}$. If f has arity n, its corresponding function symbol $\widehat{f}$ will have sort $Term \times \overset{n}{\dots} \times Term \to Term$.

Let us define the syntax of L1 expressions and statements:

$\mathbf{Exp}^1 \ni e$	::=	c	{literal}
	|	x	{variable}
	|	e_1 **and** e_2	{conjunction}
	|	e_1 **or** e_2	{disjunction}
	|	[]	{empty list}
	|	$[e_1 \mid e_2]$	{list cons cell}
	|	$\{e_1, \dots, e_n\}$	{tuple}
	|	$f(e_1, \dots, e_n)$	{function or operator application}
$\mathbf{Stm} \ni S$	::=	**skip**	{do nothing}
	|	**block** S	{local scoped evaluation}
	|	**havoc** x	{variable declaration}
	|	$S_1; S_2$	{sequential evaluation}
	|	**assume** e	{assume a formula}
	|	**assert** e	{assert a formula}

We also assume that for every function symbol $f \in \Sigma^1$ of arity n there is an overloaded specification expressed in terms of L0 formulae. Here the word *overloaded* means that there could be many pre/post-condition pairs for each function. For example, equality can be specified as follows:

$$
\begin{array}{l}
\{\textit{is-integer}(x) \wedge \textit{is-integer}(y)\} \\
x \texttt{ === } y \\
\{\textit{boolean-value}(\widehat{\texttt{===}}(x,y)) \Leftrightarrow \textit{integer-value}(x) = \textit{integer-value}(y)\} \\
\\
\{\textit{is-boolean}(x) \wedge \textit{is-boolean}(y)\} \\
x \texttt{ === } y \\
\{\textit{boolean-value}(\widehat{\texttt{===}}(x,y)) \Leftrightarrow \textit{boolean-value}(x) = \textit{boolean-value}(y)\} \\
\\
\vdots \\
\\
\{\textit{true}\} \\
x \texttt{ === } y \\
\{\textit{is-boolean}(\widehat{\texttt{===}}(x,y)) \wedge \textit{boolean-value}(\widehat{\texttt{===}}(x,y)) \Leftrightarrow (x = y)\}
\end{array}
$$

Here $\widehat{\texttt{===}}$ is the uninterpreted symbol in Σ^0 corresponding to Elixir's strict equality operator $\texttt{===} \in \Sigma^1$. We write the former in prefix form in order to highlight the fact that it is an uninterpreted function symbol in the logic. On the contrary, the $=$, $\Leftrightarrow$, $\wedge$ in the specification above are actual connectives and operators of the underlying logic.

We denote by $\sigma_1, \ldots, \sigma_m$ the specifications of a function $f \in \Sigma^1$. Each one is a pair $(\varphi(x_1, \ldots, x_n), \psi(x_1, \ldots, x_n))$, where the x_i variables denote the parameters of the function. We also denote by $\textit{Spec}(f)$ the set of specifications of f. They will be built-in into the system in order to model the Elixir semantics.

5.2. Semantics

In this section, we show the translation process from L1 statements and expressions into L0 expressions. Also, we show how the most relevant function specifications to model the Elixir semantics are built-in into the system, but we allow the user to introduce its own function definitions and specifications in terms of L1 expressions.

5.2.1. Built-in declarations

During the translation of L1 statements and expressions into L0 expressions, we require some defined sorts, constants and functions in SMT-LIB. Every representation of an L1 expression in the underlying logic has sort *Term*:

```
; The sort Term
(declare-sort Term 0)
```

Terms can be of a given type:

```
; The sort Type
(declare-sort Type 0)
```

```
; Modelled types
(declare-const int Type)
(declare-const bool Type)
(declare-const tuple Type)
(declare-const nonempty_list Type)

; All of them are different
(assert (distinct int bool))
(assert (distinct int tuple))
(assert (distinct int nonempty_list))
(assert (distinct bool tuple))
(assert (distinct bool nonempty_list))
(assert (distinct tuple nonempty_list))

; Type membership checking predicates
(declare-fun type (Term) Type)
(define-fun is_integer ((x Term)) Bool (= (type x) int))
(define-fun is_boolean ((x Term)) Bool (= (type x) bool))
(define-fun is_tuple ((x Term)) Bool (= (type x) tuple))

(define-fun is_nonempty_list ((x Term)) Bool
  (= (type x) nonempty_list)
)

(define-fun is_list ((x Term)) Bool
  (or (= x nil) (= (type x) nonempty_list))
)
```

We also need a way to introduce and eliminate *Term*s:

```
; Introduce literal values as corresponding Term
(declare-fun integer_lit (Int) Term)
(declare-fun boolean_lit (Bool) Term)

; In the case of boolean connectives, its value constructors
(declare-fun term_and (Term, Term) Term)
(declare-fun term_or (Term, Term) Term)

; In the case of lists, its value constructors
(declare-fun nil () Term)
(declare-fun cons (Term Term) Term)

; In the case of tuples, they are declared dynamically as
; (declare-fun tuple_n (Term, ..., Term) Term)

; The value of a Term of a given type in the underlying logic
(declare-fun integer_val (Term) Int)
(declare-fun boolean_val (Term) Bool)

; In the case of lists and tuples, its decomposition
```

```
(declare-fun hd (Term) Term)
(declare-fun tl (Term) Term)
(declare-fun tuple_size (Term) Int)
(declare-fun elem (Term Int) Term)
```

These SMT-LIB commands will be executed at the beginning of our verification process to initialize the solver.

5.2.2. Translation into L0

When it comes to assign a meaning to L1 statements and expressions, as this is an IR, we translate them into L0 expressions to verify them. For this, we shall define two functions:

$$\begin{array}{ll} \mathit{trExp}\ \_\ [\![\_]\!]: & \mathbf{Exp}^0 \times \mathbf{Exp}^1 \to \mathbf{Exp}^0 \times \mathbb{T} \\ \mathit{trStm}\ [\![\_]\!]: & \mathbf{Stm} \to \mathbf{Exp}^0 \end{array}$$

Given an L1 expression e, the application $\mathit{trExp}\ \gamma\ [\![e]\!]$ returns a tuple (ϵ, t), in which ϵ is an L0 expression that models the semantics of e, and t is the term in the underlying logic that will be used to refer to the result of e. The γ models those facts that are known by the time e is evaluated and is needed to handle the short circuit-based semantics of **and** and **or**. We are going to omit this γ parameter when it models no knowledge:

$$\mathit{trExp}\ [\![e]\!] \equiv \mathit{trExp}\ \mathbf{skip}\ [\![e]\!]$$

Let us define $\mathit{trExp}\ \_\ [\![\_]\!]$ case by case. In the case of literals, we get:

$$\mathit{trExp}\ \_\ [\![c]\!] \equiv (\mathbf{add}\ \mathit{is}\text{-}\tau(\tau\text{-}\mathit{lit}(\hat{c})); \mathbf{add}\ \tau\text{-}\mathit{value}(\tau\text{-}\mathit{lit}(\hat{c})) = \hat{c}, \tau\text{-}\mathit{lit}(\hat{c}))$$

where τ is the type of the literal, which can be determined at compile time since it is a literal, and $\hat{c}$ is the constant in the underlying logic represented by that literal. For example, the Elixir term **2** corresponds to the actual number $2 \in \mathbb{Z}$, so $\hat{\mathbf{2}} = 2$.

In the case of variables, we get:

$$\mathit{trExp}\ \_\ [\![x]\!] \equiv (\mathbf{skip}, \hat{x})$$

It returns the logic variable $\hat{x}$ corresponding to the L1 variable x. No L0 expression is generated.

The L0 expressions generated by a tuple correspond to the ones generated by each component, the projection function for each one and its tuple size function. Its translated term is a specific tuple constructor for its size n applied to its translated term components:

$$\begin{array}{l} \mathit{trExp}\ \gamma\ [\![\{e_1, \ldots, e_n\}]\!] \equiv (\epsilon_1; \ldots; \epsilon_n; \epsilon; \epsilon'_1; \ldots; \epsilon'_n, t) \\ \quad \mathbf{where}\ \forall i \in \{1..n\}.(\epsilon_i, t_i) = \mathit{trExp}\ \gamma\ [\![e_i]\!] \\ \qquad\qquad t = n\text{-}\mathit{tuple}(t_1, \ldots, t_n) \\ \qquad\qquad \epsilon = \mathbf{add}\ \mathit{is}\text{-}\mathit{tuple}(t); \mathbf{add}\ \mathit{tuple}\text{-}\mathit{size}(t) = n \\ \qquad\qquad \forall i \in \{1..n\}.\epsilon'_i = \mathbf{add}\ \mathit{elem}(t, i) = t_i \end{array}$$

The translation for lists is defined recursively, with the empty list as the base case. The generated L0 expressions set the corresponding heads and tails for the generated list terms, and it does not require the second argument for the list constructor to be a list:

$$
\begin{array}{l}
\mathit{trExp}\ \_\ [\![[]]\!] \equiv (\mathbf{skip}, \mathit{nil}) \\
\mathit{trExp}\ \gamma\ [\![[e_1 \mid e_2]]\!] \equiv (\epsilon_1; \epsilon_2; \epsilon, t) \\
\quad \mathbf{where}\ (\epsilon_1, t_1) = \mathit{trExp}\ \gamma\ [\![e_1]\!] \\
\qquad\qquad (\epsilon_2, t_2) = \mathit{trExp}\ \gamma\ [\![e_2]\!] \\
\qquad\qquad t = \mathit{cons}(t_1, t_2) \\
\qquad\qquad \epsilon = \left[\begin{array}{l} \mathbf{add}\ \mathit{is\text{-}nonempty\text{-}list}(t); \\ \mathbf{add}\ \mathit{hd}(t) = t_1; \\ \mathbf{add}\ \mathit{tl}(t) = t_2 \end{array}\right]
\end{array}
$$

A more complex case is that of function application:

$$
\begin{array}{l}
\mathit{trExp}\ \gamma\ [\![f(e_1, \ldots, e_n)]\!] \equiv (\epsilon_1; \ldots; \epsilon_n; \epsilon; \overline{\epsilon_\sigma}^{\sigma \in \mathit{Spec}(f)}, \widehat{f}(t_1, \ldots, t_n)) \\
\quad \mathbf{where}\ \forall i \in \{1..n\}.(\epsilon_i, t_i) = \mathit{trExp}\ \gamma\ [\![e_i]\!] \\
\qquad\qquad \epsilon = \left[\begin{array}{l} \mathbf{when\text{-}unsat}\ \gamma; \mathbf{add}\ \neg \bigvee_{\sigma \in \mathit{Spec}(f)} \mathit{Pre}(\sigma)(t_1 \ldots, t_n) \\ \quad \mathbf{do\ skip} \\ \quad \mathbf{else\ fail} \end{array}\right] \\
\qquad\qquad \forall \sigma \in \mathit{Spec}(f) \text{ such that } \sigma = (\varphi_\sigma(x_1 \ldots, x_n), \psi_\sigma(x_1, \ldots, x_n)). \\
\qquad\qquad\qquad \epsilon_\sigma = \left[\begin{array}{l} \mathbf{when\text{-}unsat}\ \gamma; \mathbf{add}\ \neg\varphi_\sigma(t_1 \ldots, t_n)\ \mathbf{do} \\ \quad \mathbf{add}\ \varphi_\sigma(t_1 \ldots, t_n); \\ \quad \mathbf{add}\ \psi_\sigma(t_1, \ldots, t_n) \\ \mathbf{else\ skip} \end{array}\right]
\end{array}
$$

Firstly, we generate the L0 expression ϵ_i corresponding to each argument e_i, and its corresponding uninterpreted term t_i. Then, for each pre/post-condition pair of the specification of the function being applied, we generate code that checks whether the precondition holds and, in case it does, we assert both the precondition and postcondition. Finally, we also check that at least one preconditions holds.

We distinguish the cases of logical connectives from function application because of their specific short-circuit semantics in Elixir:

$$
\begin{array}{l}
\mathit{trExp}\ \gamma\ [\![e_1\ \textbf{and}\ e_2]\!] \equiv (\epsilon, t) \\
\quad \textbf{where}\ (\epsilon_1, t_1) = \mathit{trExp}\ \gamma\ [\![e_1]\!] \\
\qquad\qquad (\epsilon_2, t_2) = \mathit{trExp}\ \gamma'\ [\![e_2]\!] \\
\qquad\qquad\qquad \gamma' = \gamma; \textbf{add}\ \mathit{boolean\text{-}value}(t_1) \\
\qquad\qquad\qquad t = \widehat{\textbf{and}}(t_1, t_2) \\
\qquad\qquad\qquad\qquad \epsilon = \left[\begin{array}{l}
\epsilon_1; \\
\textbf{when-unsat}\ \gamma; \textbf{add}\ \neg \mathit{is\text{-}boolean}(t_1)\ \textbf{do} \\
\quad \textbf{when-unsat}\ \gamma; \textbf{add}\ \mathit{boolean\text{-}value}(t_1)\ \textbf{do} \\
\quad\quad \textbf{add}\ \mathit{is\text{-}boolean}(t); \\
\quad\quad \textbf{add}\ \neg \mathit{boolean\text{-}value}(t); \\
\quad\quad \textbf{add}\ \neg \mathit{boolean\text{-}value}(t_1) \\
\quad \textbf{else} \\
\quad\quad \epsilon_2; \\
\quad\quad \textbf{when-unsat}\ \gamma; \textbf{add}\ \neg \mathit{boolean\text{-}value}(t_1)\ \textbf{do} \\
\quad\quad\quad \textbf{add}\ \mathit{boolean\text{-}value}(t_1) \\
\quad\quad\quad \textbf{add}\ t = t_2 \\
\quad\quad \textbf{else when-unsat}\ \gamma'; \textbf{add}\ \neg \mathit{is\text{-}boolean}(t_2)\ \textbf{do} \\
\quad\quad\quad \textbf{add}\ \mathit{is\text{-}boolean}(t); \\
\quad\quad\quad \textbf{add}\ \mathit{boolean\text{-}value}(t) = \\
\quad\quad\quad\quad (\mathit{boolean\text{-}value}(t_1) \wedge \mathit{boolean\text{-}value}(t_2)) \\
\quad\quad \textbf{else fail} \\
\textbf{else fail}
\end{array}\right]
\end{array}
$$

In the translation for an **and** expression, we firstly check if the term to the left is boolean. Then, on the one hand, if it is known to be always *false*, the resulting term is *false*. On the other hand, if it is known to be always *true*, the resulting term is the right one regardless of its type. Note that this right term has been translated with the knowledge that the left one is *true*. If the value of the left term is not exactly known at this point, we check if the right term is a boolean, again with the knowledge that the left one is *true*, and translate the whole expression into the underlying logical conjunction.

The translation corresponding to **or** is analogous:

$$
\begin{array}{l}
\mathit{trExp}\ \gamma\ [\![e_1\ \mathbf{or}\ e_2]\!] \equiv (\epsilon, t) \\
\quad \mathbf{where}\ (\epsilon_1, t_1) = \mathit{trExp}\ \gamma\ [\![e_1]\!] \\
\qquad\quad (\epsilon_2, t_2) = \mathit{trExp}\ \gamma'\ [\![e_2]\!] \\
\qquad\qquad \gamma' = \gamma; \mathbf{add}\ \neg \mathit{boolean\text{-}value}(t_1) \\
\qquad\qquad t = \widehat{\mathbf{or}}(t_1, t_2) \\
\qquad\qquad \epsilon = \left[\begin{array}{l}
\epsilon_1; \\
\mathbf{when\text{-}unsat}\ \gamma; \mathbf{add}\ \neg \mathit{is\text{-}boolean}(t_1)\ \mathbf{do} \\
\quad \mathbf{when\text{-}unsat}\ \gamma; \mathbf{add}\ \neg \mathit{boolean\text{-}value}(t_1)\ \mathbf{do} \\
\quad\quad \mathbf{add}\ \mathit{is\text{-}boolean}(t); \\
\quad\quad \mathbf{add}\ \mathit{boolean\text{-}value}(t); \\
\quad\quad \mathbf{add}\ \mathit{boolean\text{-}value}(t_1) \\
\quad \mathbf{else} \\
\quad\quad \epsilon_2; \\
\quad\quad \mathbf{when\text{-}unsat}\ \gamma; \mathbf{add}\ \mathit{boolean\text{-}value}(t_1)\ \mathbf{do} \\
\quad\quad\quad \mathbf{add}\ \neg \mathit{boolean\text{-}value}(t_1) \\
\quad\quad\quad \mathbf{add}\ t = t_2 \\
\quad\quad \mathbf{else\ when\text{-}unsat}\ \gamma'; \mathbf{add}\ \neg \mathit{is\text{-}boolean}(t_2)\ \mathbf{do} \\
\quad\quad\quad \mathbf{add}\ \mathit{is\text{-}boolean}(t); \\
\quad\quad\quad \mathbf{add}\ \mathit{boolean\text{-}value}(t) = \\
\quad\quad\quad\quad (\mathit{boolean\text{-}value}(t_1) \vee \mathit{boolean\text{-}value}(t_2)) \\
\quad\quad \mathbf{else\ fail} \\
\mathbf{else\ fail}
\end{array}\right]
\end{array}
$$

Now we move on to L1 statements. The following ones are translated in a quite straightforward way:

$$\mathit{trStm}\ [\![\mathbf{skip}]\!] \equiv \mathbf{skip}$$

$$\mathit{trStm}\ [\![\mathbf{block}\ S]\!] \equiv \mathbf{local}\ \mathit{trStm}\ [\![S]\!]$$

$$\mathit{trStm}\ [\![\mathbf{havoc}\ x]\!] \equiv \mathbf{declare}\ \widehat{x}$$

$$
\begin{array}{l}
\mathit{trStm}\ [\![S_1; S_2]\!] \equiv \mathit{trStm}\ [\![S_1]\!]; \mathit{trStm}\ [\![S_2]\!] \\
\quad \mathbf{where}\ (\epsilon_1, t_1) = \mathit{trExp}\ [\![e_1]\!] \\
\qquad\quad (\epsilon_2, t_2) = \mathit{trExp}\ [\![e_2]\!]
\end{array}
$$

In the case of **assume**, we generate the expression ϵ that corresponds to the expression being assumed and its uninterpreted term t. We ensure that the term t actually denotes a boolean value and, in this case, we assert that this boolean value is *true*:

$$
\mathit{trStm}\ [\![\mathbf{assume}\ e]\!] \equiv \left[\begin{array}{l}
\epsilon; \\
\mathbf{when\text{-}unsat\ add}\ \neg \mathit{is\text{-}boolean}(t) \\
\quad \mathbf{do\ add}\ \mathit{boolean\text{-}value}(t) \\
\quad \mathbf{else\ fail}
\end{array}\right] \qquad \mathbf{where}\ (\epsilon, t) = \mathit{trExp}\ [\![e]\!]
$$

In the case of **assert**, we also generate the expression ϵ that corresponds to the expression being assumed and its uninterpreted term t. We ensure that the term t actually denotes a boolean value and also that its boolean value is *true*:

$$trStm\ [\![\textbf{assert}\ e]\!] \equiv \left[\begin{array}{l} \epsilon; \\ \textbf{when-unsat add}\ \neg \mathit{is\text{-}boolean}(t) \\ \quad \textbf{do skip} \\ \quad \textbf{else fail}; \\ \textbf{when-unsat add}\ \neg \mathit{boolean\text{-}value}(t) \\ \quad \textbf{do add}\ \mathit{boolean\text{-}value}(t) \\ \quad \textbf{else fail} \end{array}\right] \qquad \textbf{where}\ (\epsilon, t) = trExp\ [\![e]\!]$$

5.2.3. Built-in specifications

In order to allow L1 expressions to model the semantics of built-in Elixir functions and operators, the corresponding uninterpreted functions must be declared in SMT-LIB with sort $Term \times \overset{n}{\ldots} \times Term \rightarrow Term$, and our system must provide its corresponding built-in specifications. We have explored some of them that are explained in this section.

For integer arithmetic, the specification of `+` can be defined as

$$\begin{array}{l} \{\mathit{is\text{-}integer}(x) \wedge \mathit{is\text{-}integer}(y)\} \\ x\ \texttt{+}\ y \\ \{\mathit{is\text{-}integer}(\widehat{+}(x,y)) \wedge \mathit{integer\text{-}value}(\widehat{+}(x,y)) = \mathit{integer\text{-}value}(x) + \mathit{integer\text{-}value}(y)\} \end{array}$$

and it will be extended if we model other numeric types such as `float`. It is similar for `-` and `*`. The unary version of `-` can be specified as follows:

$$\begin{array}{l} \{\mathit{is\text{-}integer}(x)\} \\ \texttt{-}\ x \\ \{\mathit{is\text{-}integer}(\widehat{-}(x)) \wedge \mathit{integer\text{-}value}(\widehat{-}(x)) = -\mathit{integer\text{-}value}(x)\} \end{array}$$

Similarly, the Elixir boolean negation can be specified as:

$$\begin{array}{l} \{\mathit{is\text{-}boolean}(x)\} \\ \mathit{not}(x) \\ \{\mathit{is\text{-}boolean}(\widehat{\mathit{not}}(x)) \wedge \mathit{boolean\text{-}value}(\widehat{\mathit{not}}(x)) \Leftrightarrow \neg \mathit{boolean\text{-}value}(x)\} \end{array}$$

We have only provided the comparison for integer terms as

$$\begin{array}{l} \{\mathit{is\text{-}integer}(x) \wedge \mathit{is\text{-}integer}(y)\} \\ x\ \texttt{<}\ y \\ \{\mathit{is\text{-}boolean}(\widehat{<}(x,y)) \wedge \mathit{boolean\text{-}value}(\widehat{<}(x,y)) \Leftrightarrow \mathit{integer\text{-}value}(x) < \mathit{integer\text{-}value}(y)\} \end{array}$$

and it is in the same way for `>`, `<=` and `>=`. An improvement would be to extend this for any term, including lists and tuples.

Term equality can be specified as

$\{\mathit{is\text{-}integer}(x) \wedge \mathit{is\text{-}integer}(y)\}$
x `===` y
$\{\mathit{boolean\text{-}value}(\widehat{===}(x, y)) \Leftrightarrow \mathit{integer\text{-}value}(x) = \mathit{integer\text{-}value}(y)\}$

$\{\mathit{is\text{-}boolean}(x) \wedge \mathit{is\text{-}boolean}(y)\}$
x `===` y
$\{\mathit{boolean\text{-}value}(\widehat{===}(x, y)) \Leftrightarrow \mathit{boolean\text{-}value}(x) = \mathit{boolean\text{-}value}(y)\}$

$\{\mathit{is\text{-}list}(x) \wedge \mathit{is\text{-}list}(y)\}$
x `===` y
$\{\mathit{boolean\text{-}value}(\widehat{===}(x, y)) \Leftrightarrow (x = \mathit{nil} \wedge y = \mathit{nil}) \vee (\mathit{hd}(x) = \mathit{hd}(y) \wedge \mathit{tl}(x) = \mathit{tl}(y))\}$

$\{\mathit{is\text{-}tuple}(x) \wedge \mathit{is\text{-}tuple}(y) \wedge \mathit{tuple\text{-}size}(x) = \mathit{tuple\text{-}size}(y)\}$
x `===` y
$\{\mathit{boolean\text{-}value}(\widehat{===}(x, y)) \Leftrightarrow (\forall i.i >= 0 \wedge i < \mathit{tuple\text{-}size}(x) \Rightarrow \mathit{elem}(x, i) = \mathit{elem}(y, i))\}$

$\{\mathit{is\text{-}tuple}(x) \wedge \mathit{is\text{-}tuple}(y) \wedge \mathit{tuple\text{-}size}(x) \neq \mathit{tuple\text{-}size}(y)\}$
x `===` y
$\{\neg \mathit{boolean\text{-}value}(\widehat{===}(x, y))\}$

$\{\mathit{true}\}$
x `===` y
$\{\mathit{is\text{-}boolean}(\widehat{===}(x, y)) \wedge \mathit{boolean\text{-}value}(\widehat{===}(x, y)) \Leftrightarrow (x = y)\}$

and it is also similar for `!==`.

The *tuple-size* and *elem* functions can be specified directly in terms of the built-in declarations used during the translation:

$\{\mathit{is\text{-}tuple}(x)\}$
$\mathit{tuple\text{-}size}(x)$
$\{\mathit{is\text{-}integer}(\widehat{\mathit{tuple\text{-}size}}(x)) \wedge \mathit{integer\text{-}value}(\widehat{\mathit{tuple\text{-}size}}(x)) = \mathit{tuple\text{-}size}(x)\}$

$\{\mathit{is\text{-}tuple}(x) \wedge \mathit{is\text{-}integer}(i) \wedge \mathit{integer\text{-}value}(i) >= 0 \wedge \mathit{integer\text{-}value}(i) < \mathit{tuple\text{-}size}(x)\}$
$\mathit{elem}(x, i)$
$\{\widehat{\mathit{elem}}(x, i) = \mathit{elem}(x, \mathit{integer\text{-}value}(i))\}$

The same can be applied to the *hd* function

$\{\mathit{is\text{-}nonempty\text{-}list}(x)\}$
$\mathit{hd}(x)$
$\{\widehat{\mathit{hd}}(x) = \mathit{hd}(x)\}$

and it is similar for *tl*. Note that, in these last examples, the L1 function is not the same as the one mentioned in the postcondition, which is a built-in L0 function, although we have used the same name.

The functions to mention the term types can also be specified directly with the built-in declared L0 functions:

$\{true\}$
$is\text{-}integer(x)$
$\{is\text{-}boolean(\widehat{is\text{-}integer}(x)) \wedge boolean\text{-}value(\widehat{is\text{-}integer}(x)) \Leftrightarrow is\text{-}integer(x)\}$

and it is similar for the remaining types.

If some Elixir operator cannot be modelled with function specifications like the ones shown in this section, they can be defined as L1 expressions with its own translation into L0, as we did to model the short-circuit semantics of the `and` and `or` operators.

5.2.4. Term size modelling

At some point, it will be required to reason about termination, mainly when verifying recursive Elixir function definitions. With that purpose, we have considered an uninterpreted function to assign an integer value to *Term*s

```
(declare-fun term_size (Term) Int)
```

and a set of axioms based on their types

$term\text{-}size(nil) = 1$
$\forall x.is\text{-}integer(x) \Rightarrow term\text{-}size(x) = 1$
$\forall x.is\text{-}boolean(x) \Rightarrow term\text{-}size(x) = 1$
$\forall x.is\text{-}nonempty\text{-}list(x) \Rightarrow term\text{-}size(x) = 1 + term\text{-}size(hd(x)) + term\text{-}size(tl(x))$
$\forall x.is\text{-}tuple(x) \Rightarrow \forall i.i >= 0 \wedge i < tuple\text{-}size(x) \Rightarrow term\text{-}size(elem(x, i)) < term\text{-}size(x)$

Note that this is a proposal. We have not worked on this for the moment, as termination reasoning is not a goal for this project and is intended to be addressed in future work.

5.3. Implementation

In this section, we present our implementation for the L1 verification IR in Elixir.

5.3.1. Overview

We have implemented a DSL in Elixir to write and verify L1 programs. The macro `with_local_env/1` provides a default environment with a fresh Z3 connection that is injected as the first argument of its inner DSL macros (i.e. `assert/2`, `havoc/2`, etc.) and is closed once all of them have been evaluated. As in our SMT binding, `with_local_env/1` is a convenient wrapper around `with_env/2`, which allows to provide a custom or reused connection and does not close it automatically.

Here are some examples that succeed to verify according to the Elixir behavior explained in Section 3.1.1:

```
import Boogiex

with_local_env do
  assert 4 - 2 === 6 - 4
  assert (false or 2) === 2
```

```
  assert 3 > 2 and 1 <= 1
  assert elem({1, 2, 3}, 0) === 1
  assert [1 | [2 | [3 | []]]] === [1, 2, 3]
  assert true or true + true

  havoc x
  assert x === x
  assert not (x !== x)

  block do
    assume x === 2
    assert is_integer(x), "This should not fail"
  end

  assert is_integer(x), "This should fail"

  havoc a
  havoc b
  havoc c

  assume is_integer(a) and is_integer(b)
  assume is_integer(a) and is_integer(b) and is_integer(c)

  assume a === b
  assume b === c
  assert a === c

  assert false, "This should fail"
end
```

The package also exposes the corresponding translation functions between DSLs for implementing other tools on top of them.

5.3.2. Details

We are going to explain our implementation in a schematic way in order to give its main idea.

First, we implement a function to translate Elixir AST corresponding to L1 expressions (i.e. its DSL), together with an assumption in terms of L0, into the term that it represents and L0 code (i.e. also its DSL) that models its semantics:

```
@spec translate_l1_exp(L0Exp.ast, L1Exp.ast)
  :: {L0Exp.ast, L0Exp.ast}
```

Its definition syntax matches pretty closely the formal version, as in this case for nonempty lists:

```
def translate_l1_exp(assumption, [{:|, _, [h, t]}]) do
  {head, head_sem} = translate_l1_exp(assumption, h)
  {tail, tail_sem} = translate_l1_exp(assumption, t)
  term = quote(do: :cons.(unquote(head), unquote(tail)))
```

```
    {
      term,
      quote do
        unquote(head_sem)
        unquote(tail_sem)
        add :is_nonempty_list.(unquote(term))
        add :hd.(unquote(term)) == unquote(head)
        add :tl.(unquote(term)) == unquote(tail)
      end
    }
  end
```

This translation relies on a state of tuple constructors that are declared on demand, and we also provide a mechanism to indicate the context of the program in order to report helpful error messages, but we have omitted such details in order to simplify.

Then, we also implement a function to translate Elixir AST corresponding to L1 statements into L0 code:

```
@spec translate_l1_stm(L1Stm.ast) :: L0Exp.ast
```

It also matches closely the formal version, as in this case for the `assert` statement:

```
def translate_l1_stm([{:assert, _, [f]}]) do
  {term, term_sem} = translate_l1_exp(nil, f)

  quote do
    when_unsat add !:is_boolean.(unquote(term)) do
    else
      fail
    end

    when_unsat add !:boolean_val.(unquote(term)) do
      add :boolean_val.(unquote(term))
    else
      fail
    end
  end
end
```

Finally, this allows us to implement a public API for the package which defines the corresponding macros from the example at Section 5.3.1. They translate the L1 DSL into L0 and evaluate it as in Section 4.2.4. A verification function for L1 statements can be implemented in terms of it as follows but, in contrast to the one presented in Section 4.2.4, our current implementation returns verification error reports instead of raising an exception and stopping the whole process:

```
@spec verify_l1(Env.t(), L1Stm.ast()) :: [term()]
def verify_l1(env, s) do
  L0Exp.eval(
    env,
    translate_l1_stm(s)
  )
end
```

Also, we have introduced some strategies to improve its performance. For example, instead of translating the whole L1 code into L0 and then evaluate it, we can translate and evaluate a sequence of L1 statements one at a time.

Chapter 6

Elixir Code Verification

"Do not fear mistakes - there are none"
— Miles Davis

This chapter shows a system to verify code of the Elixir programming language. First, we define a formal language to model a subset of sequential Elixir code which also allows introducing ghost verification conditions. Then we show how to verify it by means of the IR defined in Chapter 5.

We also show an overview of our implementation, written in Elixir itself, together with some of its implementation details.

6.1. The L2 verification language

In this section, we define the syntax of the L2 verification language, together with some procedures and definitions with the aim to verify sequential Elixir code.

6.1.1. Syntax

Let us define the set $\mathbf{Exp}^2$ of sequential Elixir expressions given by the following grammar:

$$
\begin{array}{lcll}
\mathbf{Exp}^2 \ni E & ::= & e & \{\text{L1 expression}\} \\
& | & P = E & \{\text{pattern matching}\} \\
& | & \textbf{empty} & \{\text{empty sequence}\} \\
& | & E_1; E_2 & \{\text{sequence}\} \\
& | & \textbf{case } E \textbf{ do} & \{\text{case distinction}\} \\
& & \quad P_1 \textbf{ when } f_1 \to E_1 & \\
& & \quad \vdots & \\
& & \quad P_n \textbf{ when } f_n \to E_n & \\
& & \textbf{end} & \\
& | & \textbf{ghost do } S \textbf{ end} & \{\text{L1 ghost statement}\}
\end{array}
$$

Here P denotes a pattern from a set **Pat** of patterns, defined by the following grammar:

$$\mathbf{Pat} \ni P ::= c \mid x \mid [\,] \mid [P_1 \mid P_2] \mid \{P_1, \ldots, P_n\}$$

Note that the guard expressions $f_1, \ldots, f_n$ correspond to L1 expressions, due to their restricted nature.

This language models a subset of the Elixir programming language with L1 expressions as their direct counterparts in Elixir, the **empty** sequence and sequences as Elixir blocks, **case** expressions, and a simplified version of its pattern matching capabilities. It also allows adding verification statements such as **assert** and **assume** within **ghost** blocks.

6.1.2. Translation into L1

In the following, given a set A, we use the notation $[A]$ to denote the set of sequences of elements in A. If $x_1, \ldots, x_n$ we use the notation $[x_1, \ldots, x_n]$ to denote such a sequence. We also use a list comprehension notation that is similar to the one in Haskell language. For example, $[(i, j) \mid i \leftarrow [1, 2], j \leftarrow [3, 4, 5]]$.

Let us define a function: $trEXP\ [\![\_]\!] : \mathbf{Exp}^2 \rightarrow [\mathbf{Stm} \times \mathbf{Exp}^1]$ that, given an expression E in the source language, generates a sequence of pairs (S, e) where S is the L1 statement that models the semantics of E, and e is a L1 expression that represents the result to which E is evaluated. The reason behind translating an L2 expression into a sequence is that control flow constructs such as **case** may yield to different possible execution paths.

We need an auxiliary function $trMatch\ [\![\_]\!]\ [\![\_]\!] : \mathbf{Exp}^1 \times \mathbf{Pat} \rightarrow \mathbf{Exp}^1$ that, given an L1 expression e and a pattern P, returns another L1 expression that is a *boolean* term and is evaluated to *true* if and only if e matches P. Its definition is as follows:

$$
\begin{aligned}
&trMatch\ [\![e]\!]\ [\![c]\!] = e\ \texttt{===}\ c \\
&trMatch\ [\![e]\!]\ [\![[\,]]\!] = e\ \texttt{===}\ [\,] \\
&trMatch\ [\![e]\!]\ [\![x]\!] = true \\
&trMatch\ [\![e]\!]\ [\![\{P_1, \ldots, P_n\}]\!] \\
&\quad = \textit{is-tuple}(e)\ \mathbf{and}\ \textit{tuple-size}(e)\ \texttt{===}\ n\ \mathbf{and}\ (\mathbf{and}_{i=1}^{n}\ trMatch\ [\![\textit{elem}(e, i)]\!]\ [\![P_i]\!]) \\
&trMatch\ [\![e]\!]\ [\![[P_1 \mid P_2]]\!] \\
&\quad = \textit{is-nelist}(e)\ \mathbf{and}\ trMatch\ [\![\textit{hd}(e)]\!]\ [\![P_1]\!]\ \mathbf{and}\ trMatch\ [\![\textit{tl}(e)]\!]\ [\![P_2]\!]
\end{aligned}
$$

Also, $vars(P)$ is a function to denote the $L1$ variable expressions that appear in a pattern P.

The **empty** expression is translated into an empty list:

$$trEXP\ [\![\mathbf{empty}]\!] = [(\mathbf{skip}, [])]$$

This would be properly modeled as the atom `nil` in Elixir, as it is the result of an empty `block` but, as we have not modelled atoms for the moment, we set it as the empty list.

L2 expressions that are contained within the syntax of L1 are translated as they are, but we generate an assertion to check if the singleton tuple which contains that expression is a tuple. This will force to introduce the expression in the generated L1 statements to be verified. Otherwise, bad formed expressions at the top level whose translation yields to verification failures may be ignored during the translation (e.g. `true + 2`):

$$trEXP\ [\![e]\!] = [(\mathbf{assert}\ \textit{is-tuple}(\{e\}), e)]$$

Another approach would be to extend the L1 syntax with a new statement that just checks an L1 expression.

A **ghost** expression is translated into an arbitrary $L1$ expression, for example the empty list, and the provided $L1$ statement as its semantics:

$$trEXP\ [\![\textbf{ghost do}\ S\ \textbf{end}]\!] = [(S, [])]$$

Expressions of the form $P = E$ are translated into assertions that check whether the result of evaluating E matches the pattern P, and then assume the equality between P and E.

$$\begin{array}{rl} trEXP\ [\![P = E]\!] = [(S_1; S'_1, e_1), \ldots, (S_n; S'_n, e_n)] & \\ \textbf{where} & [(S_1, e_1), \ldots, (S_n, e_n)] = trEXP\ [\![E]\!] \\ & \{y_1, \ldots, y_m\} = vars(P) \\ & \forall i \in \{1..n\} : S'_i = \left(\begin{array}{l} \textbf{assert}\ trMatch\ [\![e_i]\!]\ [\![P]\!]; \\ \textbf{havoc}\ y_1; \\ \vdots \\ \textbf{havoc}\ y_m; \\ \textbf{assume}\ e_i\ \texttt{===}\ P \end{array} \right) \end{array}$$

In order to translate a sequence of expressions $E_1; E_2$ we have to append every statement generated from the translation of E_2 to every statement generated from the translation of E_1. We must deal carefully with **ghost** expressions because, as its translation also returns an L1 expression, it must be skipped in order to avoid altering the semantics of a block.

This will be its translation when the last expression is a **ghost** one, taking into account that **;** is associative:

$$\begin{array}{rl} trEXP\ [\![E; \textbf{ghost do}\ S\ \textbf{end}]\!] = [(S_i; S, e_i) \mid i \leftarrow [1..n]] & \\ \textbf{where} & [(S_1, e_1), \ldots, (S_n, e_n)] = trEXP\ [\![E]\!] \end{array}$$

And this will be the general case when the previous one does not apply (i.e. E_2 is not a sequence ending in a **ghost**):

$$\begin{array}{rl} trEXP\ [\![E_1; E_2]\!] = [(S_i; S'_j, e'_j) \mid i \leftarrow [1..n], j \leftarrow [1..m]] & \\ \textbf{where} & [(S_1, e_1), \ldots, (S_n, e_n)] = trEXP\ [\![E_1]\!] \\ & [(S'_1, e'_1), \ldots, (S'_m, e'_m)] = trEXP\ [\![E_2]\!] \end{array}$$

The translation of **case** expressions is more complex:

$$
\begin{array}{l}
trEXP\ [\![\textbf{case}\ E\ \textbf{do}\ \overline{P_i\ \textbf{when}\ f_i \rightarrow E_i}^{n}\ \textbf{end}]\!] \\
\quad = \left\lceil \begin{array}{l|l}
(S_j; & \\
\textbf{havoc}\ y_1; & \\
\vdots & \\
\textbf{havoc}\ y_t; & \\
\textbf{assert}\ (e_{1,j}\ \textbf{and}\ ((\textbf{not}\ e_j\ \texttt{===}\ P_1)\ \textbf{or}\ f_1))\textbf{or}\ \cdots & j \leftarrow [1..m], \\
\quad \textbf{or}\ (e_{n,j}\ \textbf{and}\ ((\textbf{not}\ e_j\ \texttt{===}\ P_n)\ \textbf{or}\ f_n)); & i \leftarrow [1..n], \\
\textbf{assume}\ (\textbf{not}\ (e_{1,j}\ \textbf{and}\ ((\textbf{not}\ e_j\ \texttt{===}\ P_1)\ \textbf{or}\ f_1)))\ \textbf{and}\ \cdots & k \leftarrow [1..s_i] \\
\quad \textbf{and}\ (\textbf{not}\ (e_{i-1,j}\ \textbf{and}\ ((\textbf{not}\ e_j\ \texttt{===}\ P_{i-1})\ \textbf{or}\ f_{i-1}))); & \\
\textbf{assume}\ e_{i,j}\ \textbf{and}\ ((\textbf{not}\ e_j\ \texttt{===}\ P_i)\ \textbf{or}\ f_i); & \\
\textbf{assume}\ e_j\ \texttt{===}\ P_j; & \\
S'_{i,k}, e'_{i,k}) &
\end{array} \right\rceil \\
\textbf{where}\ \ [(S_1, e_1), \ldots, (S_m, e_m)] = trEXP\ [\![E]\!] \\
\quad \forall i \in \{1..n\} : [(S'_{i,1}, e'_{i,1}), \ldots, (S'_{i,s_i}, e'_{i,s_i})] = trEXP\ [\![E_i]\!] \\
\quad \forall i \in \{1..n\}, j \in \{1..m\} : e_{i,j} = trMatch\ [\![e_j]\!]\ [\![P_i]\!] \\
\quad \{y_1, \ldots, y_t\} = \bigcup_{i=1}^{n} vars(P_i)
\end{array}
$$

It can be described as, for each translation of E, for each branch and for each translation of the resulting expression of that branch:

1. Declare the involved variables for every pattern matching. This is because guards f_i may refer to variables bound in its pattern.
2. Check that at least one pattern and guard holds. The guards f_i are checked under the assumption that its pattern variables have been bound, which is a disjunction because there is no implication connective in L1 expressions.
3. Assume that no previous pattern and guard holds, with the same considerations explained above, and that the current one does.
4. Assume that the branch pattern does match.
5. Include the generated L1 statements for the expression corresponding to that branch.

This models appropriately the short-circuit semantics of the `case` construct because, if some branch will never be evaluated, assuming its pattern and guard will make the hypothesis state inconsistent and everything would be *true* after that, so it will not yield to validation failures.

6.1.3. Verifying L2 expressions

We will verify an L2 expression by verifying its corresponding translation into or IR.

First, for the verification of an L2 expression, as we want to allow reusing variable names, but this would be problematic in the generated L1 counterpart, we consider a function *ssa* to transform an L2 expression into Static Single-Assignment (SSA) form as in the following example:

$$E = (x = 2; x = 3 + x; y = x * x)$$

$$ssa(E) = (x_1 = 2; x_2 = 3 + x_1; y_1 = x_2 * x_2)$$

Then, we can obtain a single L1 statement that considers all the generated statements by wrapping them inside **block** expressions to be verified independently one after the other:

$$\begin{aligned} &\mathit{verification}(E) = \textbf{block}\ S_i; \ldots; \textbf{block}\ S_n \\ &\quad \textbf{where} \quad [(S_1, e_1), \ldots, (S_n, e_n)] = \mathit{trEXP}\ [\![\mathit{ssa}(E)]\!] \end{aligned}$$

Finally, the resulting expression from our IR can be translated into our low level verification language L0, as shown in Section 5.2.2 with the *trStm* $[\![\_]\!]$ function, which has its semantics defined in terms of the SMT problem.

We also define a function $\mathit{elixir}(E)$ that turns an L2 expression E into its counterpart with all its **ghost** subexpressions removed from every sequence expression. If E itself is one of them, then the result is **empty**.

6.1.4. Verifying user-defined functions

We will also define formally a representation for a set of user-defined functions with its different overloads, which correspond to Elixir functions that a user can define in an Elixir `module`.

A single function definition of a function with arity n is stated as

$$\mathit{def} \equiv (\{p\} \quad (P_1, \ldots, P_n)\ B \quad \{q\})$$

where the $p \in \mathbf{Exp}^1$ and $q \in \mathbf{Exp}^1$ denote a specified precondition and a postcondition, $P_1, \ldots, P_n$ are the parameter patterns and $B \in \mathbf{Exp}^2$ is its defined body.

Given a function named f with arity n, we denote its overloaded definitions as $\mathit{Defs}(f/n)$ = $(\mathit{def}_1, \ldots, \mathit{def}_k)$, where its definition order matters. For each function, we are going to translate its definitions into a **case** L2 expression that models them:

$$
trDef\ [\![f/n]\!] = \left[\begin{array}{l}
\textbf{ghost do} \\
\quad \textbf{havoc}\ arg_1; \\
\quad \vdots \\
\quad \textbf{havoc}\ arg_n \\
\textbf{end}; \\
\textbf{case}\ \{arg_1, \ldots, arg_n\}\ \textbf{do} \\
\quad \{P_{1,1}, \ldots, P_{1,n}\}\ \textbf{when}\ p_1 \rightarrow \\
\quad\quad res = B_1; \\
\quad\quad \textbf{ghost do} \\
\quad\quad\quad \textbf{assume}\ res === f(arg_1, \ldots, arg_n); \\
\quad\quad\quad \textbf{assert}\ q_1 \\
\quad\quad \textbf{end} \\
\quad \vdots \\
\quad \{P_{k,1}, \ldots, P_{k,n}\}\ \textbf{when}\ p_k \rightarrow \\
\quad\quad res = B_k; \\
\quad\quad \textbf{ghost do} \\
\quad\quad\quad \textbf{assume}\ res === f(arg_1, \ldots, arg_n); \\
\quad\quad\quad \textbf{assert}\ q_k \\
\quad\quad \textbf{end} \\
\quad \{arg_1, \ldots, arg_n\} \rightarrow \\
\quad\quad true \\
\textbf{end}
\end{array}\right] \\
\textbf{where}\ (def_1, \ldots, def_k) = Defs(f/n) \\
\qquad def_i = (\{p_i\} \quad (P_{i,1}, \ldots, P_{i,n})\ B_i \quad \{q_i\})
$$

We have used the fact that the rules in Elixir to select which function definition takes effect are similar to those of the Elixir `case` expression, using the guards to assume the preconditions and its branch pattern matching to allow using pattern matching also in the function parameters.

Also, a branch at the end that always succeeds is required because, as the function arguments are unknown at this point, it is not possible to know in advance if no branch will match.

A set of function overloaded definitions can be verified by applying *trDef* $[\![\_]\!]$ to each one of them and also applying the verification process shown in Section 6.1.3.

6.1.5. Unfolding user-defined functions

To allow the user to define and use verification functions in a restricted way, because we have not dealt with termination concerns in this project, we extend the L2 language with a new verification expression:

$$
\begin{array}{lll}
\textbf{Exp}^2 \ni E & ::= & \\
& | & \vdots \\
& | & \textbf{unfold}\ f(e_1, \ldots, e_n) \quad \{\text{definition unfold}\}
\end{array}
$$

where $e_i \in \textbf{Exp}^1 \forall i.i \in \{1..n\}$.

It must be specially handled in the sequence expression translation, as in the case of trailing **ghost** expressions:

$$
\begin{array}{l}
\mathit{trEXP}\ [\![E; \textbf{unfold}\ f(e'_1, \ldots, e'_k)]\!] = [(S_i; S'_j, e_i) \mid i \leftarrow [1..n], j \leftarrow [1..m]] \\
\quad \textbf{where}\ \ [(S_1, e_1), \ldots, (S_n, e_n)] = \mathit{trEXP}\ [\![E]\!] \\
\quad \textbf{where}\ \ [(S'_1, \_), \ldots, (S'_m, \_)] = \mathit{trEXP}\ [\![\textbf{unfold}\ f(e'_1, \ldots, e'_k)]\!]
\end{array}
$$

Also, the *elixir* function should deal with **unfold** expressions similarly as it deals with **ghost** ones.

Its own translation also takes advantage of the **case** expression, as we did for user-defined function verification:

$$
\begin{array}{l}
\mathit{trEXP}\ [\![\textbf{unfold}\ f(e_1, \ldots, e_n)]\!] = \mathit{trEXP}\ [\![E]\!] \\
\textbf{where}\ (\mathit{def}_1, \ldots, \mathit{def}_k) = \mathit{Defs}(f/n) \\
\qquad \mathit{def}_i = (\{p_i\} \quad (P_{i,1}, \ldots, P_{i,n})\ B_i \quad \{q_i\}) \\
\qquad E = \left[\begin{array}{l}
\textbf{case}\ \{e_1, \ldots, e_n\}\ \textbf{do} \\
\quad \{P_{1,1}, \ldots, P_{1,n}\}\ \textbf{when}\ p_1 \rightarrow \\
\quad\quad \mathit{res} = \mathit{elixir}(B_1); \\
\quad\quad \textbf{ghost do} \\
\quad\quad\quad \textbf{assume}\ \mathit{res} === f(e_1, \ldots, e_n); \\
\quad\quad\quad \textbf{assume}\ q_1 \\
\quad\quad \textbf{end} \\
\quad \vdots \\
\quad \{P_{k,1}, \ldots, P_{k,n}\}\ \textbf{when}\ p_k \rightarrow \\
\quad\quad \mathit{res} = \mathit{elixir}(B_k); \\
\quad\quad \textbf{ghost do} \\
\quad\quad\quad \textbf{assume}\ \mathit{res} === f(e_1, \ldots, e_n); \\
\quad\quad\quad \textbf{assume}\ q_k \\
\quad\quad \textbf{end} \\
\textbf{end}
\end{array}\right]
\end{array}
$$

In this case, in contrast with the generated **case** for verifying function definitions, the postconditions are assumed, and we do not add an extra branch that always holds because the function arguments are known.

Also, the verification expressions are removed from the body expansions in order to avoid possible **unfold** expressions that lead to non-terminating recursion.

It would be possible to define it allowing *L2* expressions as arguments, but we have taken this approach because the resulting translation is simpler and **unfold** $f(E)$ with $E \in \mathbf{Exp}^2$ can be expressed as $(e = E; \textbf{unfold}\ f(e))$.

6.1.6. Termination

Regarding termination, we are not verifying whether a recursive function definition terminates or not, which could be addressed by means of a ranking function over its arguments.

For this purpose, we have proposed a set of axioms based on *Term* types in Section 5.2.4, which can help in proving that the destructuring of a term always leads to smaller ones.

6.2. Implementation

This section shows our early implementation of the system and the current API for its usage. It also explains some of its implementation details briefly.

6.2.1. Overview

The package that we provide, named `Verixir`, can be imported with the `use` macro in order to add the capabilities of our system.

The preconditions can be specified with `module` attributes as in the following example:

```
defmodule Example do
  use Verixir

  @verifier requires is_integer(x)
  defv dup(x) do
    x + x
  end

  @verifier ensures uses_dup(y) === 2 * y
  defv uses_dup(y) when is_integer(y) do
    unfold dup(y)
    dup(y)
  end
end
```

which verifies during compilation and leaves in there the function definitions without verification code to be executed.

If we change the literal `2` from the example by `3`, we get the following error:

```
[error] Verification: Assert failed uses_dup(y_1) === 3 * y_1
```

6.2.2. Translation

For the translation process, we have implemented a function that given Elixir AST code that corresponds to an L2 program (i.e. its DSL), yields a list of pairs with the expression in L1 that represents its resulting value, and an L1 statement that models its meaning:

```
@spec translate_l2_exp(L2Exp.ast)
  :: [{L1Exp.ast, L1Stm.ast}]
```

As in the previous DSL translations, its definition syntax matches closely the formal version, like in this example for assignment with pattern matching:

```
def translate_l2_exp({:=, _, [p, e]}) do
  for {t, sem} <- translate(e) do
    {
      t,
      quote do
        unquote(sem)
        assert unquote(translate_match(p, e))

        unquote_splicing(
```

```
            for var <- vars(p) do
              quote do
                havoc unquote(var)
              end
            end
          )

          assume unquote(t) === unquote(p)
        end
      }
    end
  end
```

where we also have implemented an auxiliary function to obtain the variables of a pattern, as L1 variable expressions

```
@spec vars(Pat.ast) :: MapSet.t(L1Exp.ast)
```

and a function to perform the pattern matching translation:

```
@spec translate_match(Pat.ast, L2Exp.ast) :: L1Exp.ast
```

For example, this is the case for lists:

```
def translate_match({:|, _, [p1, p2]}, e) do
  tr_1 = translate_match(p1, quote(do: hd(unquote(e))))
  tr_2 = translate_match(p2, quote(do: tl(unquote(e))))

  quote(
    do:
      is_list(unquote(e)) and unquote(e) !== [] and
        unquote(tr_1) and unquote(tr_2)
  )
end
```

6.2.3. Verification

As a helper function for verification, we have also implemented one to transform an L2 program into SSA form:

```
@spec l2_ssa(L2Exp.ast) :: L2Exp.ast
```

We could not use the built-in `Macro.traverse/4` for it, which would be as usual when transforming AST with state, because the pattern matching case was difficult to handle. It was easier in contrast by defining an explicit recursive function:

```
defp ssa_rec({:=, _, [p, e]}, state) do
  {e, state} = ssa_rec(e, state)
  state = new_version_for_vars(var_names(p), state)
  {p, state} = ssa_rec(p, state)

  {
    {:=, [], [p, e]},
    state
```

```
  }
end
```

By using first the function to transform L2 code into SSA and then the L1 verification function from Section 5.3.2, we can define a verification function for L2 code as follows:

```
@spec verify_l2(Env.t(), L2Exp.ast()) :: [term()]
def verify_l2(env, e) do
  for {_, sem} <- translate_l2_exp(l2_ssa(e)) do
    L1Stm.eval(
      env,
      quote do
        block do
          unquote(sem)
        end
      end
    )
  end
  |> List.flatten()
end
```

Note that each possible path of the translation must be verified in an independent proof context, so one option is to wrap each one into a `block` statement. Another one could be to use a fresh SMT solver connection.

6.2.4. User-defined functions verification

In order to provide an API for programmers to use this verification system in their own Elixir `modules`, we have defined a module that can be imported by means of the `use` built-in macro. This section deals with Elixir concepts that have not been explained in this document.

First, we have defined a macro `defv` to write functions involved in the verification process. Those defined with this macro are collected during the `module` compilation, where preconditions and postconditions are specified as `module attributes`, replaced by regular function definitions with no verification code, and then verified by translating them into **case** expressions as explained in Section 6.1.4.

The function to remove verification code has been implemented as follows:

```
@spec remove_verification(L2Exp.ast()) :: L2Exp.ast()
def remove_verification(ast) do
  Macro.prewalk(ast, fn
    {:__block__, meta, es} ->
      {:__block__, meta,
       Enum.reject(es, fn
         {:unfold, _, _} -> true
         {:ghost, _, _} -> true
         _ -> false
       end)}

    {:unfold, _, _} ->
      {:__block__, [], []}
```

```
      {:ghost, _, _} ->
        {:__block__, [], []}

      other ->
        other
    end)
  end
```

6.2.5. User-defined functions unfolding

To allow our expression for unfolding user-defined functions, we have added to the translation function and environment with all the verification function definitions of the `module`, and new a case for it:

```
  def translate_l2_exp(
    user_defined,
    {:unfold, _, [{f, _, args}]}
  ) do
    defs = user_defined[{f, length(args)}]

    translate(
      user_defined,
      {:case, [], [quote(do: {unquote_splicing(args)}), [do:
        List.flatten(
          for d <- defs do
            quote do
              {unquote_splicing(d.args)}
              when unquote(d.pre) ->
                res = unquote(
                    remove_verification(d.body)
                  )

                ghost do
                  assume res === unquote(f)(
                      unquote_splicing(args)
                    )

                  assume unquote(d.post)
                end
            end
          end
        )
      ]]}
    )
  end
```

Chapter 7

Conclusions and Future Work

"That's just how it is: when you get over one milestone, there's another, bigger one"
— Allan Holdsworth

TODO

Bibliography

Elixir documentation in Hex. `https://hexdocs.pm/elixir`, 2022.

Erlang documentation. `https://www.erlang.org/doc/`, 2022.

NimbleParsec documentation in Hex. `https://hexdocs.pm/nimble_parsec`, 2022.

Barrett, C., Fontaine, P. and Tinelli, C. *The SMT-LIB Standard*. Digital version, 2017.

Bobot, F., Filliâtre, J.-C., Marché, C., Melquiond, G. and Paske-vich, A. *Why3 Documentation*. Digital version, 2022.

Ford, R. L. and Leino, K. R. M. *Dafny Reference Manual*. Digital version, 2017.

Fredlund, L.-Å., Gurov, D., Noll, T., Dam, M., Arts, T. and Chugunov, G. A verification tool for erlang. 2003.

Gange, G., Navas, J. A., Schachte, P., Søndergaard, H. and Stuckey, P. J. Horn clauses as an intermediate representation for program analysis and transformation. 2015.

Lattner, C. and Adve, V. The llvm instruction set and compilation strategy. 2002.

Leino, K. R. M. This is boogie 2. 2008.

Lindahl, T. The dialyzer: a discrepancy analyzer for erlang programs. 2012.

McCord, C. *Metaprogramming Elixir*. The Pragmatic Programmers, 2015.

Montenegro, M., Peña, R. and Sánchez-Hernández, J. A generic intermediate representation for verification condition generation. 2015.

Müller, P., Schwerhoff, M. and Summers, A. J. Viper: A verification infrastructure for permission-based reasoning. 2016.

Thomas, D. *Programming Elixir*. The Pragmatic Programmers, 2018.

Zhao, J., Nagarakatte, S., Martin, M. M. and Zdancewic, S. Formalizing the llvm intermediate representation for verified program transformations. 2012.

Appendix A

DSL example: Hamiltonian path problem

This appendix shows a longer usage example of our developed DSL to communicate with an SMT solver from Elixir. It corresponds to the CSP of checking whether a given graph has a Hamiltonian path or not.

A Hamiltonian path is a path that visits each node exactly once, and it can be expressed as a SAT problem by specifying it as a CSP and encoding its constraints as propositional formulas.

We are going to write it as an Elixir script (i.e. code that is not provided under a `module` and can be directly evaluated by `iex`). It will involve some Elixir usage that has not been presented in this document (e.g. the `pin` operator, the `pipe` operator, comprehensions, etc.), but we hope that the comments will clarify the idea:

```
import SmtLib

# The problem input: a graph
nodes = 0..3
edges = MapSet.new([{0, 1}, {1, 2}, {2, 3}])

# A variable identifier for every node and path position
# to denote propositional variables meaning that node
# n is in position i
node_in_position =
  for n <- nodes, {_, i} <- Enum.with_index(nodes) do
    {{n, i}, String.to_atom("p_#{n}_#{i}")}
  end
  |> Map.new()

# We use a default and self-managed solver connection
with_local_conn do

  # Declare all the variables with sort Bool
  for {_, v} <- node_in_position do
    declare_const [{v, Bool}]
  end
```

```
  # This comprehension stands for pairs of
  # variables corresponding to different nodes
  # at the same position
  for {{m, i}, v1} <- node_in_position,
      {{n, ^i}, v2} <- node_in_position,
      n !== m do
      # Nodes do not collide in their positions
    assert !(v1 && v2)
  end

  # This comprehension stands for pairs of variables
  # corresponding to different nodes in adjacent path
  # positions that are not adjacent in the graph
  for {{m, i}, v1} <- node_in_position,
      j <- [i + 1],
      {{n, ^j}, v2} <- node_in_position,
      n !== m,
      {m, n} not in edges do
    # Non adjacent nodes cannot be in adjacent positions
    assert v1 ~> !v2
  end

  # Every node is at least in some position
  for n <- nodes do
    # This reduce generates a disjunction with
    # all the positions of the node
    assert unquote(
             Enum.reduce(
               Enum.with_index(nodes),
               quote(do: false),
               fn {_, i}, acc ->
                 quote do
                   unquote(acc) ||
                     unquote(node_in_position[{n, i}])
                 end
               end
             )
           )
  end

  check_sat
end

# Print the result of the block, that is,
# the result of check_sat
|> IO.inspect()
```

And, for the given problem input, its evaluation prints

```
{:ok, :sat}
```

meaning that there exists an assignment for the defined variables that satisfies the specified formulas, thus the graph has a Hamiltonian path.

Appendix B

Title of the Appendix B

TODO

Appendix content. Think if something should be converted into an appendix.

Acronyms

API Application Programming Interface. 13, 20, 36, 48

AST Abstract Syntax Tree. 12, 19, 23, 35, 36, 46, 47

CSP Constraint Satisfaction Problem. 18, 53

Dialyzer DIscrepancy AnaLYZer for ERlang programs. 5, 11

DSL Domain Specific Language. 1, 12, 17–20, 22, 34–36, 46, 53

I/O Input/Output. 13

IR Intermediate Representation. 1, 4, 20, 25, 28, 34, 39, 42, 43

NIF Native Implemented Function. 13

OOP Object-oriented programming. 3

PropEr PROPerty-based testing tool for ERlang. 5

REPL Read-Eval-Print-Loop. 7

SAT Boolean SATisfiability problem. 14, 53

SMT Satisfiability Modulo Theories. 1, 4, 5, 7, 13–15, 17, 20–23, 34, 43, 47, 53

SSA Static Single-Assignment. 42, 47

Triq Term Reductive Invariant Questant. 5

Viper Verification Infrastructure for Permission-based Reasoning. 4

"Computing without a computer," said the president impatiently,
"is a contradiction in terms."

"Computing," said the congressman,
"is only a system for handling data. A machine might do it, or the human brain might. Let me give you an example." And, using the skills he had learned, he worked out sums and products until the president, despite himself, grew interested.

"Does this always work?"
"Every time, Mr. President. It is foolproof."

Isaac Asimov
The Feeling of Power

TODO Illustration