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

16 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

List of figures

List of tables

Chapter 1

Introduction

TODO

At some point explain the general idea, including a diagram

1.1. Motivation

TODO

1.2. Goals

TODO

- Use the Elixir macro system to implement a verification system for Elixir itself.
- To integrate SMT solvers in Elixir and offer a Domain Specific Language (DSL) to specify restriction problems.
- Develop a verification Intermediate Representation (IR) to express Erlang terms and its dynamically typed nature.
- Translate the developed IR into the previous DSL.
- Design a mechanism to translate a subset of the Elixir programming language into the verification IR.

1.3. Non-goals

TODO

- Concurrency.
- Termination (i.e. only partial verification for the moment).

1.4. Work plan

TODO

Describe the work plan to achieve the proposed goals.

Chapter 2

State of the Art

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 [reference].

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. Its verified code can be translated into other programming languages, mainly 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 of translate verified code into an executable version.

2.1.2. Intermediate representations for verification

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 be executable at native speed in web browsers [reference].

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 input for a theorem prover. This last one will try to provide a proof for the verification.

TODO: discutir algunas

- Boogie
- Why3 / WhyML
- Viper
- CAVI-ART
- CHC (Constrained Horn Clauses)
- Rule-based representation (COSTA group)

2.1.3. SMT solvers

Z3, CVC4, MATHSAT, Yices... SMT-LIB

2.2. Correctness in Elixir

2.2.1. Propery-based testing

Propery-based testing: Proper, QuickCheck (excheck)

2.2.2. Erlang Verification Tool

Erlang Verification Tool (Lars-Åke Freudlund)

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 Satisfiability Modulo Theories (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, and `escape/1` to escape AST data (e.g. it allows to introduce some given AST as data instead of as code in a macro definition):

```
iex> quote do
       hello(unquote_splicing([2, 3]))
     end
{:hello, [], [2, 3]}
iex> ast = quote(do: 2 + 2)
iex> quote do
       unquote(ast)
     end
{:+, [context: Elixir, import: Kernel], [2, 2]}
iex> quote do
       unquote(Macro.escape(ast))
     end
{:{}, [], [:+, [context: Elixir, import: Kernel], [2, 2]]}
```

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 Application Programming Interface (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 (Clark Barrett and Tinelli, 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.

TODO: reference and comment a theory with smt decidable and some efficient algorithm.

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.

TODO: general description, many sorted and references. Show the subset of commands and responses that we are going to use.

$$
\begin{array}{rcl}
\langle\, \mathit{command} \,\rangle & ::= & \texttt{(assert } \langle\, \mathit{term} \,\rangle \texttt{)} \\
& | & \texttt{(check-sat)} \\
& | & \texttt{(pop } \langle\, \mathit{numeral} \,\rangle \texttt{)} \\
& | & \texttt{(push } \langle\, \mathit{numeral} \,\rangle \texttt{)} \\
& | & \texttt{(declare-sort } \langle\, \mathit{symbol} \,\rangle \langle\, \mathit{numeral} \,\rangle \texttt{)} \\
& | & \texttt{(declare-const } \langle\, \mathit{symbol} \,\rangle \langle\, \mathit{sort} \,\rangle \texttt{)} \\
& | & \texttt{(declare-fun } \langle\, \mathit{symbol} \,\rangle \texttt{ (} \langle\, \mathit{symbol} \,\rangle^{*} \texttt{) } \langle\, \mathit{sort} \,\rangle \texttt{)}
\end{array}
$$

$$
\begin{array}{rcl}
\langle\, \mathit{general\_response} \,\rangle & ::= & \texttt{success} \\
& | & \texttt{unsupported} \\
& | & \texttt{(error } \langle\, \mathit{string} \,\rangle \texttt{)} \\
& | & \langle\, \mathit{specific\_success\_response} \,\rangle
\end{array}
$$

TODO: the example proposed by Manuel

3.2.2. Z3

One of the SMT solvers that implements the SMT-LIB standard is the Z3 theorem prover from Microsoft Research.

TODO: reference, show its usage with -in and say why we have chosen it and why we only communicate with it by means of SMT-LIB.

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 Clark Barrett and Tinelli (2017):

TODO: clarify why this affects us

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

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.

Then, we will introduce a simple formal language whose semantics are 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, 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 at the end provided to be available for the Elixir community.

4.1.1. Overview

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

```
import SmtLib

run do
  declare_const x: Int,
                y: Int

  assert !( (:x + 3 <= :y + 3 ) ~> ( :x <= :y ))
  check_sat
end
```

which evaluates to `[:unsat]`, proving that

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

We provide a `run` macro that evaluates the given DSL block by communicating with a solver. This communication can be reused, customized and configured, but if none is provided as in the example, it uses a default fresh one that communicates with Z3 through Elixir ports.

On the one hand, the expressions corresponding to logical formulas include support for variables, uninterpreted function application, quantifiers, and built-in operators and logic connectives such as `+`, `!` and `&&`.

On the other hand, it 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.

4.1.2. Implementation

As explained in 3.1.1.5, we will use Elixir type specifications as a guide to explain our implementation in a simplified form.

First, we have defined `types` to represent SMT-LIB commands and responses from the subset that we have shown in 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
```

On the one hand, 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
```

It handles compositionally the `command_t` type with cases like the following:

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

On the other hand, 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}
```

It 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()
  ])
```

Finally, to interact with the solver, we have defined an Elixir low level protocol to send SMT-LIB commands 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 to communicate with Z3 through ports and allow 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 can be used as in 4.1.1.

4.2. The L0 language

This section exposes a formal language that we have called L0 and will represent the lowest level of our verification system.

It is intended to be implemented as Elixir expressions that send SMT-LIB statements to an SMT solver and 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:

$$\begin{array}{rcll}
\mathbf{Exp}^0 \ni \epsilon & ::= & \mathbf{skip} & \{\text{do nothing}\} \\
& \mid & \mathbf{fail} & \{\text{fail signal}\} \\
& \mid & \epsilon_1; \epsilon_2 & \{\text{sequential evaluation}\} \\
& \mid & \mathbf{local}\ \epsilon & \{\text{local scoped proof state}\} \\
& \mid & \mathbf{add}\ \varphi & \{\text{add a logic formula } \varphi \in \mathbb{F} \text{ to the state}\} \\
& \mid & \mathbf{declare\text{-}const}\ x & \{\text{declare constant of type } \mathit{Term}\} \\
& \mid & \mathbf{when\text{-}unsat}\ \epsilon_1\ \mathbf{do}\ \epsilon_2\ \mathbf{else}\ \epsilon_3 & \{\text{unsatisfiability conditional}\}
\end{array}$$

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 $\epsilon_{i_1}; \ldots; \epsilon_{i_n}$.

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 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 following rules:

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

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

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

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

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

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

The absence of rules for the **fail** expression is intentional, because we want any reachable **fail** to prevent the whole expression for evaluating. We have also required this to happen if the same variable is declared twice or if a formula with undeclared variables is being added.

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 solve the problem.

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} = run(conn, push)
    eval conn, unquote(e)
    {_, :ok} = run(conn, pop)
  end
end
```

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

```
defmacro eval(conn, {:add, _, [f]}) do
  quote do
    conn = unquote(conn)
    {_, :ok} = run(conn, assert(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} =
      run(conn, declare_const([{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} = run(conn, push)
    eval conn, unquote(e1)
    {_, {:ok, result}} = run(conn, check_sat)
    {_, :ok} = run(conn, pop)

    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:

```
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 and has the `Term` sort already defined, this would be a simple example of its usage:

```
eval conn do
  declare_const :x

  # Replacing '!=' by '==' leads to
  # a verification exception
  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

TODO: general explanation

5.1. Syntax

We denote by $\Sigma^1 = \{\texttt{===}, \texttt{<=}, \texttt{>=}, \texttt{+}, \texttt{-}, \ldots\}$ 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 $\mathit{Term} \times \overset{n}{\ldots} \times \mathit{Term} \to \mathit{Term}$.

Let us define the syntax of L1 expressions and statements:

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

5.2. Semantics

TODO: general explanation

5.2.1. Built-in declarations

TODO: Organize the section about the required SMT-LIB preamble for the translation.

During the translation of L1 expressions we require some defined sorts, constants and functions. In our implementation, the SMT-LIB commands with that purpose are the following:

```
(declare-sort Term 0)
(declare-sort Type 0)
(declare-fun type (Term) Type)
(declare-fun term_size (Term) Int)
(declare-fun integer_val (Term) Int)
(declare-fun boolean_val (Term) Bool)
(declare-fun integer_lit (Int) Term)
(declare-fun boolean_lit (Bool) Term)
(declare-fun tuple_size (Term) Int)
(declare-fun elem (Term Int) Term)
(declare-fun nil () Term)
(declare-fun cons (Term Term) Term)
(declare-fun hd (Term) Term)
(declare-fun tl (Term) Term)
(declare-const int Type)
(declare-const bool Type)
(declare-const tuple Type)
(declare-const nonempty_list Type)
(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))
(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)))
```

Also, tuple constructors for any size n must be declared with sort $\mathit{Term} \times \overset{n}{\ldots} \times \mathit{Term} \rightarrow \mathit{Term}$, but in our implementation we declare each one the first time that it is required.

5.2.2. Built-in specifications

TODO: organize the section about the built-in specifications and SMT-LIB code to emulate the Elixir semantics.

We shall 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:

$$\{\textit{is-integer}(x) \wedge \textit{is-integer}(y)\}$$
$$x \texttt{ === } y$$
$$\{\textit{boolean-value}(\widehat{\texttt{===}}(x,y)) \Leftrightarrow \textit{integer-value}(x) = \textit{integer-value}(y)\}$$

$$\{\textit{is-boolean}(x) \wedge \textit{is-boolean}(y)\}$$
$$x \texttt{ === } y$$
$$\{\textit{boolean-value}(\widehat{\texttt{===}}(x,y)) \Leftrightarrow \textit{boolean-value}(x) = \textit{boolean-value}(y)\}$$

$$\vdots$$

$$\{\textit{true}\}$$
$$x \texttt{ === } y$$
$$\{\textit{is-boolean}(\widehat{\texttt{===}}(x,y)) \wedge \textit{boolean-value}(\widehat{\texttt{===}}(x,y)) \Leftrightarrow (x = y)\}$$

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.

TODO: organize the section

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

For integer arithmethic, the specification of `+` can be as

$$\{\textit{is-integer}(x) \wedge \textit{is-integer}(y)\}$$
$$x \texttt{ + } y$$
$$\{\textit{is-integer}(\widehat{+}(x,y)) \wedge \textit{integer-value}(\widehat{+}(x,y)) = \textit{integer-value}(x) + \textit{integer-value}(y)\}$$

and similar for `-` and `*`. The unary version of `-` can be specified as follows:

$$\{\textit{is-integer}(x)\}$$
$$\texttt{- } x$$
$$\{\textit{is-integer}(\widehat{-}(x)) \wedge \textit{integer-value}(\widehat{-}(x)) = -\textit{integer-value}(x)\}$$

It is similar to the Elixir boolean negation:

$$\{\textit{is-boolean}(x)\}$$
$$\textit{not}(x)$$
$$\{\textit{is-boolean}(\widehat{\textit{not}}(x)) \wedge \textit{boolean-value}(\widehat{\textit{not}}(x)) \Leftrightarrow \neg\textit{boolean-value}(x)\}$$

We have only provided the comparison for integer terms as

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

and it is similar for `>`, `<=` and `>=`. An improvement would be to extend this for any term, including lists and tuples. Term equality can be specified as

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

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

$$\begin{array}{l}\{\textit{is-list}(x) \wedge \textit{is-list}(y)\}\\ x \texttt{ === } y\\ \{\textit{boolean-value}(\widehat{===}(x,y)) \Leftrightarrow (x = \textit{nil} \wedge y = \textit{nil}) \vee (\textit{hd}(x) = \textit{hd}(y) \wedge \textit{tl}(x) = \textit{tl}(y))\}\end{array}$$

$$\begin{array}{l}\{\textit{is-tuple}(x) \wedge \textit{is-tuple}(y) \wedge \textit{tuple-size}(x) = \textit{tuple-size}(y)\}\\ x \texttt{ === } y\\ \{\textit{boolean-value}(\widehat{===}(x,y)) \Leftrightarrow (\forall i. i >= 0 \wedge i < \textit{tuple-size}(x) \Rightarrow \textit{elem}(x,i) = \textit{elem}(y,i))\}\end{array}$$

$$\begin{array}{l}\{\textit{is-tuple}(x) \wedge \textit{is-tuple}(y) \wedge \textit{tuple-size}(x) \neq \textit{tuple-size}(y)\}\\ x \texttt{ === } y\\ \{\neg \textit{boolean-value}(\widehat{===}(x,y))\}\end{array}$$

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

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:

$$\begin{array}{l}\{\textit{is-tuple}(x)\}\\ \textit{tuple-size}(x)\\ \{\textit{is-integer}(\widehat{\textit{tuple-size}}(x)) \wedge \textit{integer-value}(\widehat{\textit{tuple-size}}(x)) = \textit{tuple-size}(x)\}\end{array}$$

$$\begin{array}{l}\{\textit{is-tuple}(x) \wedge \textit{is-integer}(i) \wedge \textit{integer-value}(i) >= 0 \wedge \textit{integer-value}(i) < \textit{tuple-size}(x)\}\\ \textit{elem}(x,i)\\ \{\widehat{\textit{elem}}(x,i) = \textit{elem}(x, \textit{integer-value}(i))\}\end{array}$$

The same can be applied to the *hd* function

$$\begin{array}{l}\{\textit{is-nonempty-list}(x)\}\\ \textit{hd}(x)\\ \{\widehat{\textit{hd}}(x) = \textit{hd}(x)\}\end{array}$$

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:

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

and it is similar for the remaining types.

TODO: explain non covered things (comparison between types) and that we could not model 'and' and 'or' in this way because of short-circuit

5.2.3. Translation into L0

TODO: general explanation

We shall define two functions:

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

Given an L1 expression e, the application $\textit{trExp}\ \gamma\ [\![e]\!]$ 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:

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

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

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

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:

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

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:

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

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:

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

The most complex case is that of function application:

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

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 and fail if none of the preconditions hold.

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

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

In the translation for an **and** expression, we firstly check if the term to the left is a 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:

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

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

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

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

$$\mathit{trStm}\ [\![\textbf{havoc}\ x]\!] \equiv \textbf{declare-const}\ \widehat{x}$$

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

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*:

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

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*:

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

Finally, the **unfold** statement relies on having a user-provided definition for the involved function body as a parameterized L1 expression, and also user-provided pre/post-condition pair specifications in terms of L1 expressions:

$$\begin{array}{l} \mathit{trStm}\ [\![\textbf{unfold}\ f(e_1, \ldots, e_n)]\!] \equiv \epsilon; \overline{\epsilon_\sigma}^{\sigma \in \mathit{UserSpec}(f)} \\ \quad \textbf{where}\ \epsilon = \mathit{trStm}\ [\![\textbf{assume}\ f(e_1, \ldots, e_n) \texttt{ === } \mathit{Body}(f)(e_1, \ldots, e_n)]\!] \\ \qquad\quad \forall \sigma \in \mathit{UserSpec}(f) \text{ such that } \sigma = (p_\sigma(e_1 \ldots, e_n), q_\sigma(e_1, \ldots, e_n)). \\ \qquad\qquad (\epsilon_p, t_p) = \mathit{trExp}\ [\![p_\sigma(e_1 \ldots, e_n)]\!] \\ \qquad\qquad \epsilon_\sigma = \left[\begin{array}{l} \epsilon_p; \\ \textbf{when-unsat add}\ \neg \mathit{is\text{-}boolean}(t_p)\ \textbf{do} \\ \quad \textbf{when-unsat add}\ \neg \mathit{boolean\text{-}value}(t_p)\ \textbf{do} \\ \qquad \mathit{trStm}\ [\![\textbf{assume}\ p_\sigma(e_1 \ldots, e_n)]\!]; \\ \qquad \mathit{trStm}\ [\![\textbf{assume}\ q_\sigma(e_1 \ldots, e_n)]\!] \\ \quad \textbf{else skip} \\ \textbf{else skip} \end{array}\right] \end{array}$$

5.2.4. Term size modelling

TODO: intended to be used for reasoning about termination.

We also provide the following axioms in order to reason about term sizes:

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

They are based on the types and will be useful for reasoning about termination.

5.3. Implementation

TODO: show examples in a separate section before this one if possible.

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.

The tuple constructor state has to be managed carefully, because an SMT-LIB `pop` command can remove definitions from the solver, created after the previous `push`, and this must be always reflected in the state to be synchronized.

Then, we also implement a function to translate Elixir AST corresponding to L2 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
```

For the `unfold` case, it relies on an environment of user defined specifications and body definitions in terms of L1 but, as previously, we omit such details in order to simplify the explanation.

Finally, this allows to implement a public API for the package which defines the corresponding macros from the example [reference]. They translate the L1 DSL into L0 and evaluate it as in 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 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,
    fn -> Msg.evaluate_stm_context(s) end,
    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

TODO

6.1. The L2 verification language

TODO

6.1.1. Syntax

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

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

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

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

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

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 $x_1, \dots, x_n$ we use the notation $[x_1, \dots, x_n]$ 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: $\mathit{trEXP}\ [\![\_]\!] : \mathbf{Exp}^2 \to [\mathbf{Stm} \times \mathbf{Exp}^1]$ 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.

We need an auxiliary function $\mathit{trMatch}\ [\![\_]\!]\ [\![\_]\!] : \mathbf{Exp}^1 \times \mathbf{Pat} \to \mathbf{Exp}^1$ 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:

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

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

L2 expressions that are contained within the syntax of L1 are translated as they are, but we generate a dummy assertion to check if the singleton tuple with this expression is a tuple. Otherwise, bad formed expressions (e.g. `2 + true`) at the top level or in a sequence may be ignored by the verification process:

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

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.

$$\begin{aligned}
\mathit{trEXP}\ [\![P = E]\!] &= [(S_1; S_1', e_1), \ldots, (S_n; S_n', e_n)] \\
\mathbf{where}\ \ & [(S_1, e_1), \ldots, (S_n, e_n)] = \mathit{trEXP}\ [\![E]\!] \\
& \{y_1, \ldots, y_m\} = \mathit{vars}(P) \\
& \forall i \in \{1..n\} : S_i' = \left(\begin{array}{l}
\mathbf{assert}\ \mathit{trMatch}\ [\![e_i]\!]\ [\![P]\!]; \\
\mathbf{havoc}\ y_1; \\
\vdots \\
\mathbf{havoc}\ y_m; \\
\mathbf{assume}\ e_i \mathtt{===} P
\end{array}\right)
\end{aligned}$$

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:

$$\begin{aligned}
\mathit{trEXP}\ [\![E_1; E_2]\!] &= [(S_i; S_j', e_j') \mid i \leftarrow [1..n], j \leftarrow [1..m]] \\
\mathbf{where}\ \ & [(S_1, e_1), \ldots, (S_n, e_n)] = \mathit{trEXP}\ [\![E_1]\!] \\
& [(S_1', e_1'), \ldots, (S_m', e_m')] = \mathit{trEXP}\ [\![E_2]\!]
\end{aligned}$$

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

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

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. Check that at least one pattern and guard of holds.
2. Assume that no previous pattern and guard holds, and that the current one does.
3. Declare the involved variables for the pattern matching.
4. Assume that the pattern does match.

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.

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

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

TODO: problem, this changes the program meaning for verification, because the result of a sequence that ends in a ghost will evaluate to the empty list, which will not exist in the executable program.

6.1.3. Verification process

TODO
Verification process, and also the Elixir code generation (remove ghosts).

6.1.4. Extended user-defined functions

TODO
Allowing L2 expressions in user defined functions.

6.1.5. Termination

TODO
Show our current ideas about termination using term sizes.

6.2. Implementation

TODO: show examples in a separate section before this one if possible. Introduce this section.

For the translation process, we implement 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, as 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
```

TODO: unfold if possible, and the verification process.

By using the L1 verification function from 5.3, 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(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.

TODO: paths with a common subpath yield to repeated error reports.

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.

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

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

Lattner, C. and Adve, V. The llvm instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS*, 2002.

Leino, K. R. M. This is boogie 2. *manuscript KRML*, Vol. 178(131), 9, 2008.

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

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

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. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 427–440. 2012.

Appendix A

Title of the Appendix A

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

Appendix B

Title of the Appendix B

TODO

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

Acronyms

API Application Programming Interface. 11, 15, 29

AST Abstract Syntax Tree. 10, 11, 18, 27, 28, 34

DSL Domain Specific Language. 1, 10, 13, 14, 16, 27, 29, 34

I/O Input/Output. 11

IR Intermediate Representation. 1, 4, 15

NIF Native Implemented Function. 11

OOP Object-oriented programming. 3

REPL Read-Eval-Print-Loop. 5

SMT Satisfiability Modulo Theories. 1, 5, 11–13, 15, 16, 18

"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