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

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

11 de junio de 2022

Dedication

Acknowledgements

TODO

This is optional

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.
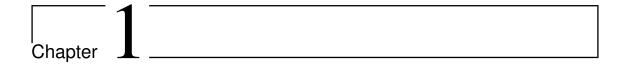
Keywords

10 keywords max., separated by commas.

Contents

List of figures

List of tables

Chapter 1

Introduction

> *"This is an interesting quote"*
> — Someone smart

TODO En algun punto explicar la idea general, incluyendo un diagrama

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

Discuss the state of the Art regarding the topics of this research project. You can cite the appearing references from the bibliography in different ways:

- With cite: Oetiker et al. (1996)

- With citep: (Oetiker et al., 1996)

- With citet: Oetiker et al. (1996)

Multiple cites at the same time (Mittelbach et al., 2004; Lamport, 1994; Knuth, 1986)

- Elixir

- Propery-based testing: Proper, QuickCheck (excheck)

- Erlang Verification Tool (Lars-Åke Freudlund)

- SMT solvers: Z3, CVC4, MATHSAT, Yices...

- Dafny

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

- Other IRs:
  - LLVM
  - BEAM
  - WebAssembly

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, where also the Erlang programming language runs [referenciar]. Both of them share some features, like their actor-based concurrency model, and have a native capability to interoperate. 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 [referenciar]. 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 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 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 exposed in the Elixir Read-Eval-Print-Loop (REPL), called `iex`, where `iex>` represents its default prompt:

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

3.1.1.1. Value types

As usual, one of the core value types in Elixir is the `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 its operators have some worth to mention semantics when involving non-`boolean` types, and also with respect to short-circuit evaluation:

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

Some built-in Elixir functions allow checking if a given value is of a given type in terms of 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
```

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] # Syntax sugar
[1, 2, 3]
iex> [1, 2 | [3]] # Syntax mix
[1, 2, 3]
```

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

```
iex> [1, 2, false]
[1, 2, false]
iex> [1 | [2 | 3]]
[1, 2 | 3]
```

Functions in Elixir use to be referred by its name and arity. The `hd/1` and `tl/1` built-in functions for lists allow 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?(                 # All
  [[], [1, 2], [1, 2 | false]], # of these
  &is_list/1                    # are lists
)
true
```

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}, []}
```

They 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`.

3.1.1.3. Blocks, pattern matching and control flow

Elixir statements 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 last statement value
iex> (
        2 + 1
        5 === 5
        false
     )
false
```

The `match` operator = allows binding values to variable names. Note that these bindings are not locally scoped inside blocks and, in contrast to Erlang, variable bindings can be overridden:

```
iex> x = 2
2
iex> y = 2
2
iex> (z = 2, 4)
4
iex> z = 2
2
```

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

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

3.1.1.4. Function definitions

A named function, identified by its name and arity, can be defined inside `modules` with different body definitions and different matching 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 for which one is applied are the same as in `case` expressions, so function definitions can also express control flow [referencia].

3.1.2. Macros

TODO

Describe, this will be our way to extend Elixir for code verification.

3.1.3. Interoperability

TODO for the SMT solver integration.

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).

One of them are Native Implemented Function (NIF)s, which allow loading and calling libraries implemented in other programming languages such as C. A crash in a NIF brings the Erlang Virtual Machine down too. [referencia docs erlang]

A more safe 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 as 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})
```

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

3.2.   Satisfiability Modulo Theories

The SMT problem consists of checking whether a given logical formula is satisfiable within a specific theory (Clark Barrett and Tinelli, 2017). This allows 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

Para esto último, poner un ejemplo referenciado de una teoría con SMT decidible y algún algoritmo eficiente.

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 both configure a solver, manage it, encode an SMT problem instance and query for solutions.

General description (many sorted, citado) and example, show the subset of commands that we are going to use

TODO

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

TODO Poner un problema sencillo de ejemplo. Separar a anexo o buscar otro mas corto?

```
; The propositional variable 'pi_j' means that the node i appears in
; the path position j, where nodes are labeled as natural numbers
; starting from 0.
(declare-const p0_0 Bool)
(declare-const p0_1 Bool)
(declare-const p0_2 Bool)
(declare-const p0_3 Bool)
(declare-const p1_0 Bool)
(declare-const p1_1 Bool)
(declare-const p1_2 Bool)
(declare-const p1_3 Bool)
```

```
(declare-const p2_0 Bool)
(declare-const p2_1 Bool)
(declare-const p2_2 Bool)
(declare-const p2_3 Bool)
(declare-const p3_0 Bool)
(declare-const p3_1 Bool)
(declare-const p3_2 Bool)
(declare-const p3_3 Bool)

; Every node should appear in at least one position.
(assert (or p0_0 p0_1 p0_2 p0_3))
(assert (or p1_0 p1_1 p1_2 p1_3))
(assert (or p2_0 p2_1 p2_2 p2_3))
(assert (or p3_0 p3_1 p3_2 p3_3))

; Two different nodes do not appear in the same path position.
(assert (not (and p0_0 p1_0)))
(assert (not (and p0_1 p1_1)))
(assert (not (and p0_2 p1_2)))
(assert (not (and p0_3 p1_3)))
(assert (not (and p0_0 p2_0)))
(assert (not (and p0_1 p2_1)))
(assert (not (and p0_2 p2_2)))
(assert (not (and p0_3 p2_3)))
(assert (not (and p0_0 p3_0)))
(assert (not (and p0_1 p3_1)))
(assert (not (and p0_2 p3_2)))
(assert (not (and p0_3 p3_3)))
(assert (not (and p1_0 p0_0)))
(assert (not (and p1_1 p0_1)))
(assert (not (and p1_2 p0_2)))
(assert (not (and p1_3 p0_3)))
(assert (not (and p1_0 p2_0)))
(assert (not (and p1_1 p2_1)))
(assert (not (and p1_2 p2_2)))
(assert (not (and p1_3 p2_3)))
(assert (not (and p1_0 p3_0)))
(assert (not (and p1_1 p3_1)))
(assert (not (and p1_2 p3_2)))
(assert (not (and p1_3 p3_3)))
(assert (not (and p2_0 p0_0)))
(assert (not (and p2_1 p0_1)))
(assert (not (and p2_2 p0_2)))
(assert (not (and p2_3 p0_3)))
(assert (not (and p2_0 p1_0)))
(assert (not (and p2_1 p1_1)))
(assert (not (and p2_2 p1_2)))
(assert (not (and p2_3 p1_3)))
(assert (not (and p2_0 p3_0)))
```

```
(assert (not (and p2_1 p3_1)))
(assert (not (and p2_2 p3_2)))
(assert (not (and p2_3 p3_3)))
(assert (not (and p3_0 p0_0)))
(assert (not (and p3_1 p0_1)))
(assert (not (and p3_2 p0_2)))
(assert (not (and p3_3 p0_3)))
(assert (not (and p3_0 p1_0)))
(assert (not (and p3_1 p1_1)))
(assert (not (and p3_2 p1_2)))
(assert (not (and p3_3 p1_3)))
(assert (not (and p3_0 p2_0)))
(assert (not (and p3_1 p2_1)))
(assert (not (and p3_2 p2_2)))
(assert (not (and p3_3 p2_3)))

; If two nodes are not adjacent, then they do not appear
; consecutively in the path.
(assert (=> p1_0 (not p3_1)))
(assert (=> p1_1 (not p3_2)))
(assert (=> p1_2 (not p3_3)))
(assert (=> p2_0 (not p3_1)))
(assert (=> p2_1 (not p3_2)))
(assert (=> p2_2 (not p3_3)))
(assert (=> p3_0 (not p1_1)))
(assert (=> p3_1 (not p1_2)))
(assert (=> p3_2 (not p1_3)))
(assert (=> p3_0 (not p2_1)))
(assert (=> p3_1 (not p2_2)))
(assert (=> p3_2 (not p2_3)))

(check-sat)
```

3.2.2.   Z3

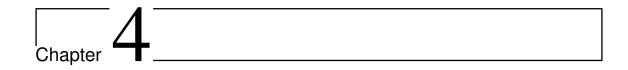One of the SMT solvers that implements the SMT-LIB standard is the Z3 theorem prover from Microsoft Research.

TODO

Poner alguna referencia, enseñar su uso con -i

Decir porque lo hemos elegido y que aun así hemos intentado utilizarlo como interprete SMT-LIB para no depender completamente de el.

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 add 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).

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 a solver.

4.1. SMT-LIB interpreter bindings

Initially, we found an Elixir package that provides integration with Z3 and is implemented using ports, but it seemed to be not so mature and maintained. It was also not published in Hex, the Elixir package manager, and we had no guarantees about if it would lack some functionality that we would require at some point.

To implement our own SMT-LIB interpreter binding was 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, which may be at short term useful to the Elixir community.

4.1.1. Elixir SmtLib example

Present the resulting tool, the DSL and an example.

4.1.2. SmtLib implementation

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. We will use these specifications together with function identifiers to outline the ideas behind our implementation.

First, we define a `type` to represent an SMT-LIB command from the subset that we have exposed 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}
```

where other involved types like `numeral_t` and `sort_t` are defined similarly.

Then, we implement a function `commands/1` that, given a subset of the Elixir AST (i.e. our DSL), transforms it into a list of SMT-LIB commands:

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

The implementation defines cases for each possible term and its possible subterms, like the following:

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

TODO explain the DSL and think about split this into subsections distinguish from function names by modules

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

```
@spec command(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)})"
```

Communication with ports, implementation for Z3 but it can be other

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

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

It has been implemented using `nimble`, an Elixir package of parser combinators, in order to delegate this task and get the reliability of a well tested tool [referencia]. 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()
  ])
```

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:

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

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 $\textit{unsat}\ [\![\_]\!]$ 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:

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

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

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

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

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

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

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

TODO
Simple implementation using the SMT-LIB bindings for Elixir. Show a simple example.

```
defmacro eval(_, {:skip, _, _}) do
  quote do
    nil
  end
end

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

defmacro eval(conn, {:seq, _, [e1, e2]}) do
  quote do
    conn = unquote(conn)
    eval(conn, unquote(e1))
    eval(conn, unquote(e2))
  end
end

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

```elixir
  end

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

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

  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

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

5.1. Syntax

TODO

5.2. Semantics

TODO

5.2.1. Built-in declarations

TODO

Required SMT-LIB preamble for the translation.

5.2.2. Built-in specifications

TODO

Built-in specifications and SMT-LIB code to emulate the Elixir semantics.

5.2.3. Translation into L0

TODO

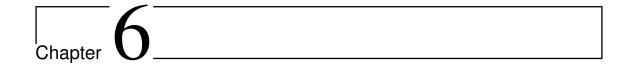The translation into L0

5.2.4. Term size modelling

TODO

Intended to be used for reasoning about termination.

5.3. Implementation

TODO

Show examples in a separate section before this one if possible.

Mention relevant facts of the development process and implementation details.

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

TODO

6.1.2. Extended verification functions

TODO

Allowing L2 expressions in user defined functions.

6.1.3. Translation and verification

TODO

Translate L2 code into L1 for verification and into Elixir code.

6.1.4. 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.
Mention relevant facts of the development process and implementation details.

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

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.
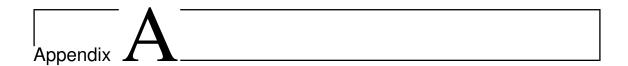
Miguel de Cervantes Saavedra

CLARK BARRETT, P. F. and TINELLI, C. *The SMT-LIB Standard*. Digital version, 2017.

KNUTH, D. E. *The TEX book*. Addison-Wesley Professional., 1986.

LAMPORT, L. *LATEX: A Document Preparation System, 2nd Edition*. Addison-Wesley Professional, 1994.

MITTELBACH, F., GOOSSENS, M., BRAAMS, J., CARLISLE, D. and ROWLEY, C. *The LATEX Companion*. Addison-Wesley Professional, segunda edn., 2004.

OETIKER, T., PARTL, H., HYNA, I. and SCHLEGL, E. *The Not So Short Introduction to LATEX 2ε*. Versión electrónica, 1996.

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

DSL Domain Specific Language. 1, 13, 14

I/O Input/Output. 9

IR Intermediate Representation. 1, 3, 15

NIF Native Implemented Function. 9

REPL Read-Eval-Print-Loop. 5

SMT Satisfiability Modulo Theories. 1, 3, 5, 9, 10, 12, 13, 15

"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