# 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**


**12 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

TODO

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: Thomas (2018)
- With citep: (Thomas, 2018)
- With citet: Thomas (2018)

Multiple cites at the same time (Thomas, 2018; Albuquerque and Caixinha, 2018; Clark Barrett and Tinelli, 2017)

- 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 [referenciar]. 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 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 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 into 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 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 deepest cons cell) are also allowed (Eli, 2022):

```
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 provides 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 branch 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 a `module` 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 (Thomas, 2018).

### 3.1.2. Macros

Because of its metaprogramming capabilities based on macros, Elixir is a suitable language for implementing DSLs [referenciar]. This will allow us to extend it without requiring to leave the language itself.

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 unevaluated Elixir Abstract Syntax Tree (AST), and its return value must also be valid Elixir AST that will replace the macro invocation at compile-time. The resulting code may also contain other macros that will be expanded recursively.

TODO: explain the Elixir AST and the usage of quote, unquote, unquotesplicing, Macro.escape, etc

TODO: examples

### 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). 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 them is to use Native Implemented Function (NIF)s, which allow 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 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 other 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 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: reference of 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 both 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

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

TODO: show an example like the following

```
; 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: reference, show its usage with -in and say why we have chosen it and why we

only communicate by means of smtlib.

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):

```
$ 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 a solver.

## 4.1. SMT-LIB interpreter binding

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, and may be at the end provided to be available for the Elixir community.

### 4.1.1. Example

TODO: overview of the resulting tool, its DSL and an example.

### 4.1.2. 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 `types` to represent SMT-LIB commands and responses 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}

@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 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]
```

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

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

TODO: explain the DSL and 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 [referenciar]. 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()
  ])
```

We expose then the public Application Programming Interface (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 $\varphi$, $\psi$, etc. to denote elements from this set.

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

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

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

### 4.2.2. Syntax

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

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

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:

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

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

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

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

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

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

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, `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 `eval` 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__, [], es}
) when is_list(es) do
  quote do
    conn = unquote(conn)
    eval conn, unquote(es)
  end
end

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

```
    eval conn, unquote(e)
  end
end

defmacro eval(_, []) do
  nil
end

defmacro eval(conn, [e | es]) do
  quote do
    conn = unquote(conn)
    eval conn, unquote(e)
    eval conn, unquote(es)
  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, a `conn` variable that represents a fresh connection with an SMT solver which has the `Term` sort already defined, this would be a simple example of 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 $\Sigma^1 = \{\texttt{===}, \texttt{<=}, \texttt{>=}, \texttt{+}, \texttt{-}, \dots\}$ the set of operators and functions allowed in L1. We assume that for every element $f \in \Sigma^1$ there is an uninterpreted function symbol in $\Sigma^0$, which will be denoted by $\widehat{f}$. If $f$ has arity $n$, its corresponding function symbol $\widehat{f}$ will have sort $\mathit{Term} \times \overset{n}{\dots} \times \mathit{Term} \to \mathit{Term}$.

Let us define the syntax of L1 expressions and statements:

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

## 5.2. Semantics

TODO: general explanation

### 5.2.1. Built-in declarations

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

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 $Term \times \overset{n}{\ldots} \times Term \to Term$, 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

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 $\Sigma^0$ 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

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

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

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

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

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

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

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

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

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

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

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

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

The same can be applied to the *hd* function

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

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

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

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

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:

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

Given an L1 expression $e$, the application $\mathit{trExp}\ \gamma\ [\![e]\!]$ returns a tuple $(\epsilon, t)$, in which $\epsilon$ 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 $\gamma$ 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 $\gamma$ parameter when it models no knowledge:

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

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

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

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

In the case of variables, we get:

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

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

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

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

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

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

The most complex case is that of function application:

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

Firstly, we generate the L0 expression $\epsilon_i$ 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:

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

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:

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

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

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

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

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

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

In the case of **assume**, we generate the expression $\epsilon$ 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*:

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

In the case of **assert**, we also generate the expression $\epsilon$ 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*:

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

Finally, the **unfold** statement relies on having an 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:

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

### 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:

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

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

## 5.3. Implementation

TODO: draft

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

Mention relevant facts of the development process and implementation details:

By using the L0 language implementation

function to translate L1 AST expressions into (t, L0)

it relies on an env of function specifications, where tuple constructors are defined lazily into the system.

function to translate L1 AST statements into L0, using the previous one

for the unfold, it relies on an env of user defined specifications and body (in L1)

now we can translate AST of L1 and evaluate it as if it were L0.

also, public API provided in a top level module

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

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

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

Albuquerque, A. and Caixinha, D. *Mastering Elixir*. Packt Publishing, 2018.

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

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

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. 17

**AST** Abstract Syntax Tree. 9, 20

**DSL** Domain Specific Language. 1, 9, 15, 16, 18

**I/O** Input/Output. 10

**IR** Intermediate Representation. 1, 3, 17

**NIF** Native Implemented Function. 10

**REPL** Read-Eval-Print-Loop. 5

**SMT** Satisfiability Modulo Theories. 1, 3, 5, 10, 12, 15, 17, 18, 20

*"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