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

10 de junio de 2022

Dedication

TODO, this is optional

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

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: 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. This is a `string` literal:

```
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 and pattern matching

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 | []]]]
[1, 2, 3]
iex> [1, 2 | [3]]
[1, 2, 3]
iex> [1, 2, 3]
[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 extract 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, assignments and control flow

TODO

Comment what has to do with pattern matching (where to introduce it?)

3.1.1.4. Function definitions

TODO

Comment what has to do with pattern matching (where to introduce it?)

Multiple bodies, name+arity, and recursion

3.1.2. Macros

TODO

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

3.1.3. Interoperability

TODO

Different ways (ports, Native Implemented Function (NIF)s), for the SMT solver integration.

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

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

Poner un problema sencillo de ejemplo

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

TODO

4.1. SMT-LIB interpreter bindings

TODO
Explain why we have developed this in the way that we have done it.
Mention relevant facts of the development process and implementation details.
Present the resulting tool, the DSL and an example.

4.2. The L0 language

Show the formalization of L0, a simple language in top of where the IR is defined.
Present also an implementation draft in terms of our DSL.

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.

We assume a set Σ^0 of uninterpreted function symbols.

We assume a set $\mathbb{T}$ of terms in many-sorted logic, generated by the following grammar:

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

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

We assume the following syntax for the lowest level language, which corresponds to Elixir expressions that send SMT-LIB statements to the solver:

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

where $\varphi \in \mathbb{F}$ denotes a formula of arithmetic logic with equality and uninterpreted function symbols.

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:

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

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

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

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

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

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

The absence of rules for the **fail** expression is intentional, because we want any reachable **fail** to prevent the whole expression for evaluating. This also happens 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.

```
defmodule L0 do
  import SmtLib

  defmacro init(conn) do
    quote do
      run(unquote(conn)) do
        declare_sort Term
      end
    end
  end

  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
  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(_, other) do
    raise "Unknown L0 expression #{Macro.to_string(other)}"
  end
end
```

Chapter 5

The L1 Intermediate Representation

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. Lowering to 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

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, 11
IR	Intermediate Representation. 1, 3, 11
NIF	Native Implemented Function. 8
REPL	Read-Eval-Print-Loop. 5
SMT	Satisfiability Modulo Theories. 1, 3, 5, 8, 9

"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