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

9 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 relevant 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 encodings 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. Both languages share some features, like their actor-based concurrency model, and have a native capability to interoperate between them. Although Elixir is younger than Erlang, this has allowed the former to be part of an ecosystem which has been developed across more than three decades.

We have chosen such a programming language for this research because, first, it is a modern programming language ready to be used in the industry [referenciar]. Second, it has the unusual property in formal verification to be dynamically typed, but its functional programming principles will make its reasoning easier. 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 Elixir programming. [El objetivo de esto es, a parte de presentar el lenguaje, el de exponer cómo se comporta el subconjunto del lenguaje que hemos estudiado por el momento para su verificación].

[mencionar iex para los snippets]

3.1.1.1. Value types

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

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

The `boolean` type is also present, but its operators have some worth to mention semantics when involving non-`boolean` types:

```
iex> true and 2
2
iex> 2 and true
** (BadBooleanError)
iex> false or 2
2
iex> 2 or false
** (BadBooleanError)
true
```

And also with respect to short-circuit evaluation:

```
iex> false and 1 / 0
false
iex> true or 1 / 0
true
```

Some built-in Elixir functions allow to check if a term 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 to mix types:

```
iex> 2 === true
false
iex> 2 === 2
true
iex> 2 > 1
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 can be .

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

```
iex> [1, 2 | [false]]
[1, 2, false]
iex> [1 | [2 | [false | []]]]
[1, 2, false]
iex> [1 | [2 | [3 | 4]]]
[1, 2, 3 | 4] # An improper list

iex> hd([1, 2, false])
1
iex> tl([1, 2, false])
[2, false]
iex> hd([])
** (ArgumentError)
iex> tl([])
** (ArgumentError)
iex> Enum.all?(                    # All
  [[], [1, 2], [1, 2 | false]], # of these
  &is_list/1                    # are lists
)
true
```

3.1.1.3. Function definitions

Comment what has to do with pattern matching
Multiple bodies, name+arity, and recursion

3.1.1.4. Blocks, assignments and control flow

Comment what has to do with pattern matching

3.1.2. Macros

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

3.1.3. Interoperability

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.

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 an 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

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

Poner un problema sencillo (el mismo de la sección anterior)

3.2.2. Z3

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

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 discrepancies when implementing the SMT-LIB standard. For example, while trying to use the `echo` command as a delimiter for the solver responses as suggested in the standard document, we have found that Z3 does not include the surrounding double-quotes when it prints back the provided string literal, which is specified in the standard. This behavior may add confusion because the `echo` command is the only one that responds with a string and, in Z3, the `( echo "sat")` response is exactly the same as the `( check-sat )` one when it returns `sat`, so a command response can be confused with the string used to delimit command responses.

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.

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

TODO

6.1. The L2 Elixir subset

TODO

6.2. Verification system

TODO

6.2.1. Extended verification functions

TODO
Allowing L2 expressions in user defined functions.

6.2.2. L2 verification

TODO
Translate L2 code into verification and Elixir code.

6.2.3. Termination

TODO
Show our current ideas about termination using term sizes.

6.3. Implementation

TODO
Show examples in a separate section before this one if possible.
Mention relevant facts of the development process and implementation details.

Chapter 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.

Appendix B

Title of the Appendix B

TODO
Appendix content.

Acronyms

DSL Domain Specific Language. 1, 9

IR Intermediate Representation. 1, 3, 9

NIF Native Implemented Function. 7

SMT Satisfiability Modulo Theories. 1, 3, 5, 7, 8

“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