Program Verification in Elixir

Master's Degree in Formal Methods and Computer Engineering

Adrián Enríquez Ballester
Supervisor: Manuel Montenegro Montes
July 3, 2022

Complutense University of Madrid

Table of Contents

Introduction

Motivation

Motivation

- Light-weight program verification systems:

Motivation

- Light-weight program verification systems:
  - Allow specifying and verifying code

Motivation

- Light-weight program verification systems:
  - Allow specifying and verifying code
  - Reduce the required human intervention

Motivation

- Light-weight program verification systems:
  - Allow specifying and verifying code
  - Reduce the required human intervention
- Dafny:

Motivation

- Light-weight program verification systems:
  - Allow specifying and verifying code
  - Reduce the required human intervention
- Dafny:
  - Specify code with pre/post conditions

Motivation

- Light-weight program verification systems:
  - Allow specifying and verifying code
  - Reduce the required human intervention
- Dafny:
  - Specify code with pre/post conditions
  - Compiled to Boogie, a verification IR

Motivation

- Light-weight program verification systems:
  - Allow specifying and verifying code
  - Reduce the required human intervention
- Dafny:
  - Specify code with pre/post conditions
  - Compiled to Boogie, a verification IR
  - Verification conditions discharged by the Z3 theorem prover

Motivation

- Light-weight program verification systems:
  - Allow specifying and verifying code
  - Reduce the required human intervention
- Dafny:
  - Specify code with pre/post conditions
  - Compiled to Boogie, a verification IR
  - Verification conditions discharged by the Z3 theorem prover
  - Compiled also to other programming languages to be executed

The Elixir programming language

The Elixir programming language

- A functional programming language that runs on the Erlang Virtual Machine

The Elixir programming language

- A functional programming language that runs on the Erlang Virtual Machine
- Dynamically typed

The Elixir programming language

- A functional programming language that runs on the Erlang Virtual Machine
- Dynamically typed
- Suitable for developing DSLs through macros

The Elixir programming language

- A functional programming language that runs on the Erlang Virtual Machine
- Dynamically typed
- Suitable for developing DSLs through macros
- Main current verification approaches:

The Elixir programming language

- A functional programming language that runs on the Erlang Virtual Machine
- Dynamically typed
- Suitable for developing DSLs through macros
- Main current verification approaches:
    - Dialyzer (static)

The Elixir programming language

- A functional programming language that runs on the Erlang Virtual Machine
- Dynamically typed
- Suitable for developing DSLs through macros
- Main current verification approaches:
  - Dialyzer (static)
  - Property-based testing (dynamic)

The Elixir programming language

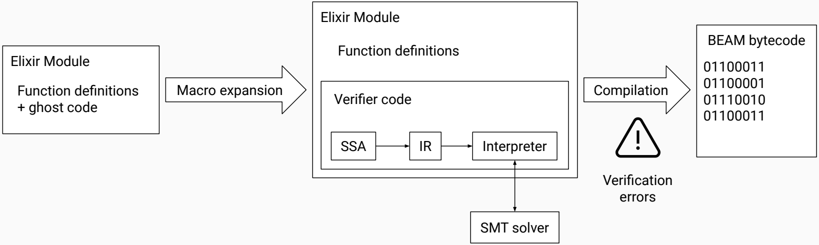
- A functional programming language that runs on the Erlang Virtual Machine
- Dynamically typed
- Suitable for developing DSLs through macros
- Main current verification approaches:
    - Dialyzer (static)
    - Property-based testing (dynamic)
    - Both of them show the presence of errors rather than their absence

Our aim

Provide a system similar to that of Dafny but specialized for Elixir and implemented in Elixir itself

Our aim

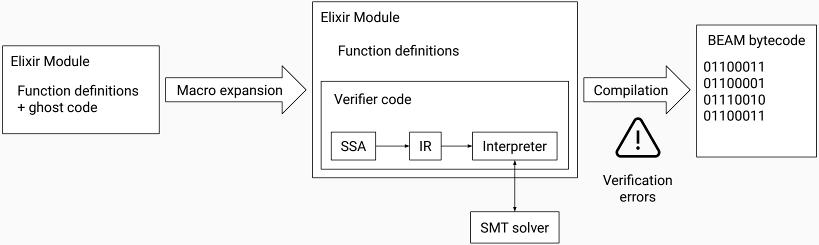
Provide a system similar to that of Dafny but specialized for Elixir and implemented in Elixir itself



https://github.com/adrianen-ucm/verixir-project

Our aim

Provide a system similar to that of Dafny but specialized for Elixir and implemented in Elixir itself



`https://github.com/adrianen-ucm/verixir-project`

Scope: only a subset of sequential Elixir for the moment, and partial verification (i.e. not verifying termination)

Plan

Plan

1. SMT solver integration in Elixir

Plan

1. SMT solver integration in Elixir
2. L0, a low level language close to the SMT solver

Plan

1. SMT solver integration in Elixir
2. L0, a low level language close to the SMT solver
3. L1, a verification IR for dynamically typed Elixir expressions

Plan

1. SMT solver integration in Elixir
2. L0, a low level language close to the SMT solver
3. L1, a verification IR for dynamically typed Elixir expressions
4. L2, a high level language that models Elixir + verification code

SMT Solver Integration in Elixir

Elixir SMT-LIB binding

We have developed an SMT-LIB binding for Elixir with the following features:

Elixir SMT-LIB binding

We have developed an SMT-LIB binding for Elixir with the following features:

- An SMT-LIB (subset) DSL

Elixir SMT-LIB binding

We have developed an SMT-LIB binding for Elixir with the following features:

- An SMT-LIB (subset) DSL
- Different SMT solvers can be easily integrated

Elixir SMT-LIB binding

We have developed an SMT-LIB binding for Elixir with the following features:

- An SMT-LIB (subset) DSL
- Different SMT solvers can be easily integrated
- Out-of-the-box support for Z3

Elixir SMT-LIB binding example

```
import SmtLib

with_local_conn do
  declare_const x: Int,
                y: Int

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

  check_sat
end
```

The L0 language

The L0 language

- The lowest level language of our verification stack

The L0 language

- The lowest level language of our verification stack
- Close to the SMT solver

The L0 language

- The lowest level language of our verification stack
- Close to the SMT solver
- Restricted SMT-LIB + control flow + failure

L0 expressions syntax

$$
\begin{array}{rcl}
\mathbf{Exp}^0 \ni \epsilon & ::= & \textbf{skip} \\
& \mid & \textbf{fail} \\
& \mid & \epsilon_1 ; \epsilon_2 \\
& \mid & \textbf{local } \epsilon \\
& \mid & \textbf{add } \varphi \\
& \mid & \textbf{declare } x \\
& \mid & \textbf{when-unsat } \epsilon_1 \textbf{ do } \epsilon_2 \textbf{ else } \epsilon_3
\end{array}
$$

L0 expressions syntax

$$
\begin{array}{rcl}
\mathbf{Exp}^0 \ni \epsilon & ::= & \mathbf{skip} \\
& | & \mathbf{fail} \\
& | & \epsilon_1; \epsilon_2 \\
& | & \mathbf{local}\ \epsilon \\
& | & \mathbf{add}\ \varphi \\
& | & \mathbf{declare}\ x \\
& | & \mathbf{when\text{-}unsat}\ \epsilon_1\ \mathbf{do}\ \epsilon_2\ \mathbf{else}\ \epsilon_3
\end{array}
$$

where $x \in V$ is a variable name and $\varphi \in \mathbb{F}$ is a formula with many-sorted terms $t \in \mathbb{T}$

L0 big-step operational semantics

Notation:

L0 big-step operational semantics

Notation:

- $X \subseteq V$ set of variable names

L0 big-step operational semantics

Notation:

- $X \subseteq V$ set of variable names
- $\Phi \subseteq \mathbb{F}$ set of formulas

L0 big-step operational semantics

Notation:

- $X \subseteq V$ set of variable names
- $\Phi \subseteq \mathbb{F}$ set of formulas
- $\mathbb{F}(X)$ subset of $\mathbb{F}$ with free variables in X

L0 big-step operational semantics

Notation:

- $X \subseteq V$ set of variable names
- $\Phi \subseteq \mathbb{F}$ set of formulas
- $\mathbb{F}(X)$ subset of $\mathbb{F}$ with free variables in X
- (X, Φ) SMT solver state

L0 big-step operational semantics

Notation:

- $X \subseteq V$ set of variable names
- $\Phi \subseteq \mathbb{F}$ set of formulas
- $\mathbb{F}(X)$ subset of $\mathbb{F}$ with free variables in X
- (X, Φ) SMT solver state
- $\langle \epsilon,\ X,\ \Phi \rangle \Downarrow (X', \Phi')$ judgement

L0 big-step operational semantics

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

L0 big-step operational semantics

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

$$\frac{\varphi \in \mathbb{F}(X)}{\langle \textbf{add}\ \varphi,\ X,\ \Phi\rangle \Downarrow (X, \Phi \cup \{\varphi\})}$$

L0 big-step operational semantics

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

$$\frac{\varphi \in \mathbb{F}(X)}{\langle \textbf{add}\ \varphi,\ X,\ \Phi\rangle \Downarrow (X, \Phi \cup \{\varphi\})}$$

$$\frac{x \notin X}{\langle \textbf{declare}\ x,\ X,\ \Phi\rangle \Downarrow (X \cup \{x\}, \Phi)}$$

L0 big-step operational semantics

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

$$\frac{\varphi \in \mathbb{F}(X)}{\langle \textbf{add}\ \varphi,\ X,\ \Phi\rangle \Downarrow (X, \Phi \cup \{\varphi\})}$$

$$\frac{x \notin X}{\langle \textbf{declare}\ x,\ X,\ \Phi\rangle \Downarrow (X \cup \{x\}, \Phi)}$$

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

L0 big-step operational semantics

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

L0 big-step operational semantics

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

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

L0 big-step operational semantics

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

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

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

L0 Elixir implementation

A simple implementation in Elixir is straightforward by using our SMT-LIB binding

L0 Elixir implementation

A simple implementation in Elixir is straightforward by using our SMT-LIB binding

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

L0 Elixir example

```
eval conn do
  declare_const :x

  when_unsat add :x != :x do
    skip # Does not reach fail
  else
    fail
  end
end
```

L0 Elixir example

```
eval conn do
  declare_const :x

  when_unsat add :x == :x do
    skip
  else
    fail # Reaches fail
  end
end
```

Verification Intermediate Representation

The L1 language

The L1 language

- Verification IR

The L1 language

- Verification IR
- It models Elixir expressions dynamically typed

The L1 language

- Verification IR
- It models Elixir expressions dynamically typed
- Statements for writing verification code

L1 expressions syntax

$$
\begin{array}{rcl}
\mathbf{Exp}^1 \ni e & ::= & c \\
& | & x \\
& | & e_1 \ \mathbf{and} \ e_2 \\
& | & e_1 \ \mathbf{or} \ e_2 \\
& | & [] \\
& | & [e_1 \mid e_2] \\
& | & \{e_1, \ldots, e_n\} \\
& | & f(e_1, \ldots, e_n)
\end{array}
$$

L1 expressions syntax

$$
\begin{array}{rcl}
\mathbf{Exp}^1 \ni e & ::= & c \\
& \mid & x \\
& \mid & e_1 \ \mathbf{and} \ e_2 \\
& \mid & e_1 \ \mathbf{or} \ e_2 \\
& \mid & [] \\
& \mid & [e_1 \mid e_2] \\
& \mid & \{e_1, \ldots, e_n\} \\
& \mid & f(e_1, \ldots, e_n)
\end{array}
$$

where c is a constant literal of a simple type, currently integer or boolean, and $f \in \Sigma^1$ a function name

L1 statements syntax

$$
\begin{array}{rcl}
\textbf{Stm} \ni S & ::= & \textbf{skip} \\
& \mid & \textbf{block}\ S \\
& \mid & \textbf{havoc}\ x \\
& \mid & S_1 ; S_2 \\
& \mid & \textbf{assume}\ e \\
& \mid & \textbf{assert}\ e \\
& \mid & \textbf{unfold}\ f(e_1, \ldots, e_n)
\end{array}
$$

Built-in SMT-LIB declarations

Foundation to represent L1 expressions in the underlying many-sorted logic

Built-in SMT-LIB declarations

Foundation to represent L1 expressions in the underlying many-sorted logic

```
(declare-sort Term 0)
(declare-sort Type 0)
...
(declare-const int Type)
(declare-const bool Type)
(assert (distinct int bool))
...
(declare-fun type (Term) Type)
(define-fun is_integer ((x Term)) Bool
  (= (type x) int)
)
...
```

Built-in L1 specifications

Built-in **sets** of pair/postconditions for functions to model their behavior in Elixir

Built-in L1 specifications

Built-in **sets** of pair/postconditions for functions to model their behavior in Elixir

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

Built-in L1 specifications

Built-in **sets** of pair/postconditions for functions to model their behavior in Elixir

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

There could be more for other types (e.g. float)

Translation from L1 into L0

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

Translation from L1 into L0

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

$\mathit{trExp}\ \gamma\ [\![e]\!]$ returns a tuple (ϵ, t) where

Translation from L1 into L0

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

$\mathit{trExp}\ \gamma\ [\![e]\!]$ returns a tuple (ϵ, t) where

- ϵ is an L0 expression that models the semantics of e

Translation from L1 into L0

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

$\mathit{trExp}\ \gamma\ [\![e]\!]$ returns a tuple (ϵ, t) where

- ϵ is an L0 expression that models the semantics of e
- t is a term in the underlying logic to refer to the result of e

Translation from L1 into L0

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

$\textit{trExp}\ \gamma\ [\![e]\!]$ returns a tuple (ϵ, t) where

- ϵ is an L0 expression that models the semantics of e
- t is a term in the underlying logic to refer to the result of e
- γ models known facts by the time e is evaluated

Translation of L1 lists

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

Translation of L1 lists example

$$
\begin{aligned}
&\mathit{trExp}\ \gamma\ [\![[2, x]]\!] \equiv (\epsilon, \mathit{cons}(2, \mathit{cons}(\hat{x}, \mathit{nil}))) \\
&\qquad \textbf{where } \epsilon = \left[\begin{array}{l}
\textbf{add } \mathit{is\text{-}integer}(\mathit{integer\text{-}lit}(2)); \\
\textbf{add } \mathit{integer\text{-}value}(\mathit{integer\text{-}lit}(2)) = 2; \\
\textbf{add } \mathit{is\text{-}nonempty\text{-}list}(\mathit{cons}(\hat{x}, \mathit{nil})); \\
\textbf{add } \mathit{hd}(\mathit{cons}(\hat{x}, \mathit{nil})) = \hat{x}; \\
\textbf{add } \mathit{tl}(\mathit{cons}(\hat{x}, \mathit{nil})) = \mathit{nil}; \\
\textbf{add } \mathit{is\text{-}nonempty\text{-}list}(\mathit{cons}(2, \mathit{cons}(\hat{x}, \mathit{nil}))); \\
\textbf{add } \mathit{hd}(\mathit{cons}(2, \mathit{cons}(\hat{x}, \mathit{nil}))) = 2; \\
\textbf{add } \mathit{tl}(\mathit{cons}(2, \mathit{cons}(\hat{x}, \mathit{nil}))) = \mathit{cons}(\hat{x}, \mathit{nil});
\end{array}\right]
\end{aligned}
$$

L1 Elixir implementation

Our implementation is quite direct from the formalization

L1 Elixir implementation

Our implementation is quite direct from the formalization

```
def tr_exp(_, [{:|, _, [h, t]}]) do
  {h, h_sem} = tr_exp(_, h)
  {y, t_sem} = tr_exp(_, t)
  t =
    quote(do: :cons.(unquote(h), unquote(t)))

  { t, quote do
    unquote(h_sem)
    unquote(t_sem)
    add :is_nonempty_list.(unquote(t))
    add :hd.(unquote(t)) == unquote(h)
    add :tl.(unquote(t)) == unquote(t)
  end }
end
```

L1 Elixir example

```
import Boogiex

with_local_env do
  assert (false or 2) === 2
  assert elem({1, 2, 3}, 0) === 1
  assert true or true + true

  havoc x
  assert x === x
  assert not (x !== x)
end
```

Elixir Code Verification

The L2 language

The L2 language

- The highest level language of our verification stack

The L2 language

- The highest level language of our verification stack
- Elixir (subset) + ghost verification code

L2 expressions syntax

$$
\begin{array}{rcl}
\textbf{Exp}^2 \ni E & ::= & e \\
& | & P = E \\
& | & \textbf{empty} \\
& | & E_1; E_2 \\
& | & \textbf{case } E \textbf{ do} \\
& & \quad P_1 \textbf{ when } f_1 \to E_1 \\
& & \quad \vdots \\
& & \quad P_n \textbf{ when } f_n \to E_n \\
& & \textbf{end} \\
& | & \textbf{ghost do } S \textbf{ end}
\end{array}
$$

L2 expressions syntax

$$
\begin{array}{rcl}
\mathbf{Exp}^2 \ni E & ::= & e \\
& | & P = E \\
& | & \mathbf{empty} \\
& | & E_1; E_2 \\
& | & \mathbf{case}\ E\ \mathbf{do} \\
& & \quad P_1\ \mathbf{when}\ f_1 \rightarrow E_1 \\
& & \quad \vdots \\
& & \quad P_n\ \mathbf{when}\ f_n \rightarrow E_n \\
& & \mathbf{end} \\
& | & \mathbf{ghost\ do}\ S\ \mathbf{end}
\end{array}
$$

where $P, P_1, \ldots P_n$ are patterns:

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

Translation from L2 into L1

$$trEXP\ [\![\_]\!] : \mathbf{Exp}^2 \rightarrow [\mathbf{Stm} \times \mathbf{Exp}^1]$$
$$trMatch\ [\![\_]\!]\ [\![\_]\!] : \mathbf{Exp}^1 \times \mathbf{Pat} \rightarrow \mathbf{Exp}^1$$

Translation from L2 into L1

$$trEXP\ [\![\_]\!] : \mathbf{Exp}^2 \rightarrow [\mathbf{Stm} \times \mathbf{Exp}^1]$$
$$trMatch\ [\![\_]\!]\ [\![\_]\!] : \mathbf{Exp}^1 \times \mathbf{Pat} \rightarrow \mathbf{Exp}^1$$

$trEXP\ [\![E]\!]$ generates a sequence of pairs (S, e) where

Translation from L2 into L1

$$\mathit{trEXP}\ [\![\_]\!] : \mathbf{Exp}^2 \rightarrow [\mathbf{Stm} \times \mathbf{Exp}^1]$$
$$\mathit{trMatch}\ [\![\_]\!]\ [\![\_]\!] : \mathbf{Exp}^1 \times \mathbf{Pat} \rightarrow \mathbf{Exp}^1$$

$\mathit{trEXP}\ [\![E]\!]$ generates a sequence of pairs (S, e) where

- S is an L1 statement that models the semantics of E

Translation from L2 into L1

$$trEXP\ [\![\_]\!] : \mathbf{Exp}^2 \rightarrow [\mathbf{Stm} \times \mathbf{Exp}^1]$$
$$trMatch\ [\![\_]\!]\ [\![\_]\!] : \mathbf{Exp}^1 \times \mathbf{Pat} \rightarrow \mathbf{Exp}^1$$

$trEXP\ [\![E]\!]$ generates a sequence of pairs (S, e) where

- S is an L1 statement that models the semantics of E
- e is an L1 expression that represents the result to which E is evaluated

Translation from L2 into L1

$$\begin{gathered} \mathit{trEXP}\ [\![\_]\!] : \mathbf{Exp}^2 \to [\mathbf{Stm} \times \mathbf{Exp}^1] \\ \mathit{trMatch}\ [\![\_]\!]\ [\![\_]\!] : \mathbf{Exp}^1 \times \mathbf{Pat} \to \mathbf{Exp}^1 \end{gathered}$$

$\mathit{trEXP}\ [\![E]\!]$ generates a sequence of pairs (S, e) where

- S is an L1 statement that models the semantics of E
- e is an L1 expression that represents the result to which E is evaluated
- Each pair corresponds to an execution path

Translation of L2 lists pattern matching

trMatch $[\![e]\!]$ $[\![P]\!]$ returns an L1 expression that is a *boolean* term and is evaluated to *true* if and only if e matches P

Translation of L2 lists pattern matching

trMatch $[\![e]\!]$ $[\![P]\!]$ returns an L1 expression that is a *boolean* term and is evaluated to *true* if and only if e matches P

$$\begin{aligned}&\mathit{trMatch}\ [\![e]\!]\ [\![[P_1 \mid P_2]]\!] =\\&\qquad \mathit{is\text{-}nelist}(e)\ \textbf{and}\\&\qquad \mathit{trMatch}\ [\![\mathit{hd}(e)]\!]\ [\![P_1]\!]\ \textbf{and}\\&\qquad \mathit{trMatch}\ [\![\mathit{tl}(e)]\!]\ [\![P_2]\!]\end{aligned}$$

Translation of L2 pattern matching expressions

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

Verifying user-defined functions

A single clause of a function with arity n:

$$def \equiv (\{p\} \quad (P_1, \ldots, P_n)\ B \quad \{q\})$$

Verifying user-defined functions

A single clause of a function with arity n:

$$def \equiv (\{p\} \quad (P_1, \ldots, P_n)\ B \quad \{q\})$$

where $p \in \mathbf{Exp}^1$ and $q \in \mathbf{Exp}^1$ denote a specified precondition and a postcondition, $P_1, \ldots, P_n$ are the parameter patterns and $B \in \mathbf{Exp}^2$ is its defined body

Verifying user-defined functions

A single clause of a function with arity n:

$$def \equiv (\{p\} \quad (P_1, \ldots, P_n)\ B \quad \{q\})$$

where $p \in \mathbf{Exp}^1$ and $q \in \mathbf{Exp}^1$ denote a specified precondition and a postcondition, $P_1, \ldots, P_n$ are the parameter patterns and $B \in \mathbf{Exp}^2$ is its defined body

Clauses of a function f with arity n:

$$Defs(f/n) = (def_1, \ldots, def_k)$$

Verifying user-defined functions

Verifying user-defined functions

1. Transform the function definition clauses into an L2 case expression with the parameter variables free and a trivial branch

Verifying user-defined functions

1. Transform the function definition clauses into an L2 case expression with the parameter variables free and a trivial branch
2. Apply Static Single-Assignment to allow rebinding variables

Verifying user-defined functions

1. Transform the function definition clauses into an L2 case expression with the parameter variables free and a trivial branch
2. Apply Static Single-Assignment to allow rebinding variables
3. Translate it into the corresponding execution paths in our IR

Verifying user-defined functions

1. Transform the function definition clauses into an L2 case expression with the parameter variables free and a trivial branch
2. Apply Static Single-Assignment to allow rebinding variables
3. Translate it into the corresponding execution paths in our IR
4. Translate each path into an L0 statement

Verifying user-defined functions

1. Transform the function definition clauses into an L2 case expression with the parameter variables free and a trivial branch
2. Apply Static Single-Assignment to allow rebinding variables
3. Translate it into the corresponding execution paths in our IR
4. Translate each path into an L0 statement
5. Verify them independently of each other

Verifying user-defined functions

1. Transform the function definition clauses into an L2 case expression with the parameter variables free and a trivial branch
2. Apply Static Single-Assignment to allow rebinding variables
3. Translate it into the corresponding execution paths in our IR
4. Translate each path into an L0 statement
5. Verify them independently of each other

Note: our formalization does not address currently the verification of user-defined function invocations (i.e. their specifications and body unfolding), but our implementation does it by automatically generating ghost code.

L2 Elixir implementation

Again, our implementation is quite direct from the formalization

L2 Elixir implementation

Again, our implementation is quite direct from the formalization

```
def tr_match({:|, _, [p1, p2]}, e) do
  tr_1 =
    tr_match(p1, quote(do: hd(unquote(e))))
  tr_2 =
    tr_match(p2, quote(do: tl(unquote(e))))

  quote(do:
    is_list(unquote(e)) and
    unquote(e) !== [] and
    unquote(tr_1) and unquote(tr_2)
  )
end
```

L2 Elixir example

Live demo

Conclusions

Conclusions

Conclusions

- We have developed a framework for Elixir code verification across several areas (i.e. SMT solver integration, verification IR and code verification)

Conclusions

- We have developed a framework for Elixir code verification across several areas (i.e. SMT solver integration, verification IR and code verification)
- Future work may address concurrent code and total verification

Conclusions

- We have developed a framework for Elixir code verification across several areas (i.e. SMT solver integration, verification IR and code verification)
- Future work may address concurrent code and total verification
- Also, we have left several improvements on the way:

Conclusions

- We have developed a framework for Elixir code verification across several areas (i.e. SMT solver integration, verification IR and code verification)
- Future work may address concurrent code and total verification
- Also, we have left several improvements on the way:
    - More of the SMT-LIB standard and SMT solvers support

Conclusions

- We have developed a framework for Elixir code verification across several areas (i.e. SMT solver integration, verification IR and code verification)
- Future work may address concurrent code and total verification
- Also, we have left several improvements on the way:
    - More of the SMT-LIB standard and SMT solvers support
    - Extend our IR to model more Elixir value types and built-in functions

Conclusions

- We have developed a framework for Elixir code verification across several areas (i.e. SMT solver integration, verification IR and code verification)
- Future work may address concurrent code and total verification
- Also, we have left several improvements on the way:
  - More of the SMT-LIB standard and SMT solvers support
  - Extend our IR to model more Elixir value types and built-in functions
  - Extend the Elixir subset to verify (e.g. pin operator and higher-order)

Conclusions

- We have developed a framework for Elixir code verification across several areas (i.e. SMT solver integration, verification IR and code verification)
- Future work may address concurrent code and total verification
- Also, we have left several improvements on the way:
    - More of the SMT-LIB standard and SMT solvers support
    - Extend our IR to model more Elixir value types and built-in functions
    - Extend the Elixir subset to verify (e.g. pin operator and higher-order)
    - The current implementation is in an early proof of concept stage

Program Verification in Elixir

Master's Degree in Formal Methods and Computer Engineering

Adrián Enríquez Ballester
Supervisor: Manuel Montenegro Montes

July 3, 2022

Complutense University of Madrid