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

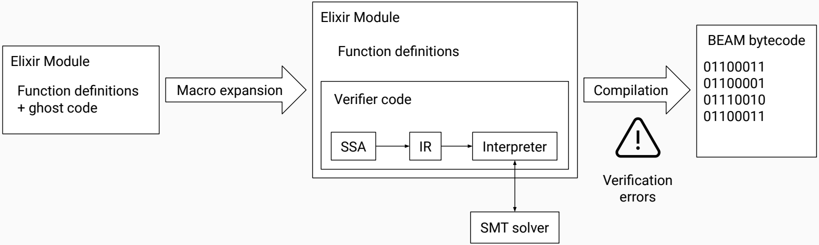
Light-weight program verification (trying to make it friendly)

Motivation

Dafny: overview, verification IR (Boogie), SMT solver (Z3)

Our aim

Provide a similar system but specialized for Elixir and implemented in Elixir



Comment Elixir: dynamically typed, functional, DSLs, current verif. approaches...

Only a subset of sequential Elixir for the moment, and partial verification (no concurrency and no termination)

Plan

SMT solver integration (binding)

L0 (closer to SMT solver)

L1 (verification IR)

L2 (Elixir + ghost code)

SMT Solver Integration

Elixir SMT-LIB binding

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

- An SMT-LIB (subset) Domain-Specific Language (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 lowest level language of our verification stack
- Closer to the SMT solver
- Restricted SMT-LIB + control flow + failure

L0 expressions syntax

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

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:

- $X \subseteq V$ 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

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

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

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

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

L0 big-step operational semantics

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

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

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

L0 Elixir implementation

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

```
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

- Verification Intermediate Representation (IR)
- It models Elixir expressions dynamically typed
- Statements for writing verification code

L1 expressions syntax

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

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

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

Built-in SMT-LIB declarations

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

```
(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

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

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

Translation from L1 into L0

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

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

- ϵ, an L0 expression that models the semantics of e
- t, 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

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

Translation of L1 lists example

$$
\mathit{trExp}\ \gamma\ [\![[2, x]]\!] \equiv (\epsilon, \mathit{cons}(2, \mathit{cons}(\hat{x}, \mathit{nil})))
$$

$$
\textbf{where } \epsilon = \begin{bmatrix}
\textbf{add}\ \mathit{is\text{-}integer}(\mathit{integer\text{-}lit}(2)); \\
\textbf{add}\ \mathit{integer\text{-}value}(\mathit{integer\text{-}lit}(2)) = 2; \\
\textbf{add}\ \mathit{is\text{-}nonempty\text{-}list}(\mathit{cons}(\hat{x}, \mathit{nil})); \\
\textbf{add}\ \mathit{hd}(\mathit{cons}(\hat{x}, \mathit{nil})) = \hat{x}; \\
\textbf{add}\ \mathit{tl}(\mathit{cons}(\hat{x}, \mathit{nil})) = \mathit{nil}; \\
\textbf{add}\ \mathit{is\text{-}nonempty\text{-}list}(\mathit{cons}(2, \mathit{cons}(\hat{x}, \mathit{nil}))); \\
\textbf{add}\ \mathit{hd}(\mathit{cons}(2, \mathit{cons}(\hat{x}, \mathit{nil}))) = 2; \\
\textbf{add}\ \mathit{tl}(\mathit{cons}(2, \mathit{cons}(\hat{x}, \mathit{nil}))) = \mathit{cons}(\hat{x}, \mathit{nil});
\end{bmatrix}
$$

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

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

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

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

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

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

Translation of L2 list patterns

TODO

Translation of L2 pattern matching expressions

TODO

Verifying user-defined functions

TODO

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

- We have developed a framework for Elixir code verification across several areas (SMT solver integration, a verification IR and Elixir code verification)
- Future work may address concurrency and termination
- Also, we have left several improvements on the way
  - More SMT-LIB 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 final tool 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