A calculus for a LLVM-based software verification tool LAV

Milena Vujošević Janićić

Faculty of Mathematics
University of Belgrade
Serbia

EUTypes meeting, Nijmegen, Netherlands, January 22-24, 2018.
Verified software verification?

Correctness of software is critical in many domains

- Automated software verification tools are getting more and more accepted and involved in software development process
- But, are these tools themselves correct?
Motivation

LLVM IR

Properties of LAV

Conclusions and further work

Ongoing work

LAV

• Deals with code in widely used LLVM IR
• Uses SMT solvers for checking verification conditions

Goals

• Define a suitable LLVM IR semantics
• Model LAV’s correctness conditions construction
• Prove properties of LAV such as soundness and completeness (for some classes of programs)

Ultimate goal

Formalization within a proof assistant and extraction of a verified software verifier for LLVM IR
Restrictions and extensions

Start with a subset of LLVM IR and a subset of LAV

- Consider programs without loops and recursive function calls (a wider class of programs reduces to this one by unrolling)
- Cover integer manipulations, simple memory model (no pointers) and only functions with available definitions
- We will denote the set of functions that satisfy these restrictions as $\mathcal{F}_R$ and programs consisting of such functions as $\mathcal{P}_{\mathcal{F}_R}$
- Cover LAV without optimizations LAV$_R$

Spiral development instead of waterfall development

- Models and proofs should be easily extensible going from very restricted to the full power of both LLVM IR and LAV
An LLVM-based compiler is structured as a translation from a high-level source language to the LLVM IR.

It is SSA-based IR, originally developed as a research tool for studying optimizations and modern compilation techniques, but nowadays is much more than that.

It is a real world IR (not a toy language): big and complex, spanning a big number of possible language constructs.

It is challenging to formally reason about it.
LLVM IR syntax (picture taken from VeLLVM paper popl’12)

Motivation

LLVM IR

Properties of LAV

Conclusions and further work

Milena Vujošević Janičić
(University of Belgrade)

Propreties of software verification tool LAV

EUTypes

5 / 30
LLVM IR example

```c
int add(int x, int y) {
    return x + y;
}

int main() {
    int a = add(3, 5);
    return 0;
}
```

```llvm
define i32 @add(i32 %x, i32 %y) #0 {
entry:
    %x.addr = alloca i32, align 4
    %y.addr = alloca i32, align 4
    store i32 %x, i32* %x.addr, align 4
    store i32 %y, i32* %y.addr, align 4
    %0 = load i32* %x.addr, align 4
    %1 = load i32* %y.addr, align 4
    %add = add i32 %0, %1
    ret i32 %add
}

define i32 @main() #0 {
entry:
    %retval = alloca i32, align 4
    %a = alloca i32, align 4
    store i32 0, i32* %retval
    %call = call i32 @add(i32 3, i32 5)
    store i32 %call, i32* %a, align 4
    ret i32 0
}
```
Defining LLVM IR semantics

Different ways for modelling LLVM IR semantics

- Some recent projects give some concrete definitions of semantics
- It is almost necessary to ignore (or abstract) a number of details
Examples of definitions of LLVM IR semantics (1)

Application in cryptography


LLVM IR semantics for symbolic execution

- Concrete and symbolic semantics for LLVM IR
- Showed that their approach for analysing cryptographic protocol implementations is sound (by proving operational correspondence between the two semantics).
Examples of definitions of LLVM IR semantics (2)

Application in program transformations and compilation

Verified LLVM — Vellvm
- A framework that includes a formal semantics and associated tools for mechanized verification of LLVM IR code, IR to IR transformations, and analyses.
- It is built using the Coq interactive theorem prover.
- It includes multiple operational semantics and proves relations among them to facilitate different reasoning styles in the context of compiler’s transformations.
Our LLVM IR semantics

Transition system

- Instructions change memory $\mathcal{M}_C$
- The semantics is described in terms of a transition system with states $\langle f : b_n : c_k, \mathcal{M}_C \rangle$ and with transitions corresponding to executions of individual instructions
- Our semantic must cover runtime and other errors
Our LLVM IR semantics

Error conditions

- Runtime errors or unexpected program behaviour can be caused by invalid operands values
- Some error conditions:

<table>
<thead>
<tr>
<th>bop type $op_1, op_2$</th>
<th>error kind</th>
<th>error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>overflow</td>
<td>$\text{signed}(type) \land op_1 &gt; 0 \land op_2 &gt; 0 \land op_1 + op_2 &lt; 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{signed}(type) \land op_1 &gt; 0 \land op_2 &lt; 0 \land op_1 + op_2 &lt; 0$</td>
</tr>
<tr>
<td>sub</td>
<td></td>
<td>$\text{signed}(type) \land op_1 &gt; 0 \land op_2 &lt; 0 \land op_1 + op_2 &lt; 0$</td>
</tr>
<tr>
<td>sdiv</td>
<td></td>
<td>$\text{signed}(type) \land op_1 = \text{min_value}(type) \land op_2 = -1$</td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td>$\text{signed}(type) \land op_2 &gt; 0 \land op_1 &gt; \text{max_value}(type)/op_2$</td>
</tr>
<tr>
<td>udiv, sdiv,</td>
<td>division by zero</td>
<td>$op_2 = 0$</td>
</tr>
<tr>
<td>urem, srem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some concrete semantics rules

**Binary operations**

\[
c = (i = bop t \ op_1, \ op_2) \quad \land_i \neg cond_i(\text{err}_i \ \text{kind}_i, \ bop(t, \ M_C(\op_1), \ M_C(\op_2)))
\]

\[
\langle f : b_n : c_k, \ M_C \rangle \rightarrow P \langle f : b_n : c_{k+1}, \ M_C\{i \mapsto (t, \ M_C(\op_1) \ bop \ M_C(\op_2))\} \rangle
\]

**BOP**

\[
c = (i = bop t \ op_1, \ op_2) \quad \text{cond}_i(\text{err}_i \ \text{kind}_i, \ bop(t, \ M_C(\op_1), \ M_C(\op_2)))
\]

\[
\langle f : b_n : c_k, \ M_C \rangle \rightarrow P \ \mathcal{E}R\mathcal{R}
\]

**BOPerr**

**Branching**

\[
c = (\text{br} \ \text{val} \ l_1 \ l_2)
\]

\[
M_C(\text{val}) = \top \quad \text{block}(l_1) = b_t
\]

\[
\langle f : b_n : c_k, \ M_C \rangle \rightarrow P \langle f : b_t : c_1, \ M_C \rangle
\]

**BRT**

\[
c = (\text{br} \ \text{val} \ l_1 \ l_2)
\]

\[
M_C(\text{val}) = \bot \quad \text{block}(l_2) = b_f
\]

\[
\langle f : b_n : c_k, \ M_C \rangle \rightarrow P \langle f : b_f : c_1, \ M_C \rangle
\]

**BRF**
Execution

Definition (Partial concrete execution)

For a program $\mathcal{P}$, for a function $f = (f dcl, b_1 \ldots b_n \ldots b_m)$ and a command $c_k$ of a block $b_n$, a partial concrete execution $CE(\mathcal{P}, f: b_n: c_k)$ is a sequence of states $s_1 s_2 \ldots s_l$ such that

$$s_1 = \langle f : b_1 : c_1, \mathcal{M}_{C[a_1, \ldots a_i]} \rangle \xrightarrow{\mathcal{P}} \langle f : b_n : c_k, \mathcal{M}_{C}^{b_n: c_k-1} \rangle \xrightarrow{\mathcal{P}} \langle f : b : c, \mathcal{M}_{C}^{b_n: c_k} \rangle = s_l$$

(Intuitively, $c_k$ is the last executed instruction.)
Execution and transition — concrete

Definition (Concrete block execution and transition)

Let $\mathcal{CE}^\circ$ be a partial concrete execution $s_1 s_2 \ldots s_l$.

**Concrete block execution** If there exists $i$ such that

$i \in \{1, \ldots, (l - 1)\}$ and $s_i = \langle f : b : c, M_C \rangle$ is in $\mathcal{CE}^\circ$, we say that a block $b$ gets executed in $\mathcal{CE}^\circ$ and we write $\mathcal{CE}^\circ \triangleright b$.

**Concrete block transition** If there exists $v$ such that

$v \in \{1, \ldots, (l - 1)\}$ and 

$s_v = \langle f : b_i : c_j, M_C \rangle \rightarrow_P \langle f : b_{i+1} : c_1, M'_C \rangle = s_{v+1}$

are in $\mathcal{CE}^\circ$, we say that there is transition from block $b_i$ to block $b_{i+1}$ and we write $\mathcal{CE}^\circ \triangleright tr(b_i, b_{i+1})$. 
Introducing orderings

Execution paths

- Intuitively, LAV constructs a formula that describes all possible executions through a function (or through a part of a function)
- A model of such formula should correspond to some concrete (partial) execution
- We need to sort functions and blocks (instructions inside one block are naturally sorted)
Ordering functions

Partial ordering $\prec_f$

- For a recursion-free program $\mathcal{P}$ and its set of functions $\mathcal{F}$, we define relation $\prec_f : \prec_f \subseteq \mathcal{F} \times \mathcal{F}$ in the following way: $f_1 \prec_f f_2$ if $f_1$ is called within $f_2$ and $\prec_f$ is transitivity closed.
- $\prec_f$ is a strict partial ordering over $\mathcal{F}$
- A sequence of functions $f_1, f_2, \ldots, f_n$ is sorted if there are no indices $i$ and $j$ such that $i < j$ and $f_j \prec_f f_i$ (Intuitively, such an ordering of the functions corresponds to one of bottom-up traversals of the control-flow graph for $\mathcal{P}$.)
Ordering blocks

Partial ordering $\prec_b$

- For a loop free function $f$ and its blocks $b_i$, we define relation $\prec_b \subseteq B \times B$ in the following way: $b_1 \prec_b b_2$ if $b_1$ has $b_2$ as an immediate successor and $\prec_b$ is transitivity closed.
- $\prec_b$ is a strict partial ordering over $B$
- A sequence of blocks $b_1, b_2, \ldots, b_n$ sorted if there are no indices $i$ and $j$ such that $i < j$ and $b_j \prec_b b_i$ (Intuitively, such an ordering of the blocks corresponds to one of top-down traversals of the control-flow graph for the function $f$.)
Transition system

Annotating states $\langle F, M^b_s, C^b_s \rangle$

- Annotating instructions, blocks and functions ($\mathcal{F}_R$)
- $F (fdcl, B (C c C) B) F$ is a sequence of functions $f$.
- $f_i = (fdcl, B (C c C) B)$ is partly annotated.
- $M^b_s$ corresponds to a symbolic memory
- $C^b_s$ set of constraints which are necessary for modelling concepts like pointers and function calls
Some transition rules

### Binary operations

\[ c = (id = \text{bop } t \ op_1, op_2) \quad ec = \lor_i \text{cond}(\text{err\_kind}, \text{bop}(t, \mathcal{M}_S^b(op_1), \mathcal{M}_S^b(op_2))) \]
\[ btr = \land_{id \in \mathcal{ID}}(\text{final}(b, id) = \mathcal{M}_S^b(id)) \land_{\text{cond}_i \in \mathcal{C}_S^b} \text{cond}_i \]
\[ pftr = \text{active}(b_1) \land_{\forall b \in \mathcal{B}} \text{ann}(b) \land \text{entry}(b) \land \text{active}(b) \]

\[ \langle F (fdcl, \overline{B} (\overline{C} cC) B F, \mathcal{M}_S^b, \mathcal{C}_S^b) \rightarrow P \rangle \]
\[ \langle F (fdcl, \overline{B} (\overline{C} cC^{\langle ec, btr, pftr \rangle} C) B) F, \mathcal{M}_S^b \{ id \mapsto (t, \mathcal{M}_S^b(op_1) \text{ sbop } \mathcal{M}_S^b(op_2)) \}, \mathcal{C}_S^b \rangle \]

### Branching instruction

\[ c = (\text{br } val \ l_1 \ l_2) \quad ec = \perp \]
\[ btr = \land_{id \in \mathcal{ID}}(\text{final}(b, id) = \mathcal{M}_S^b(id)) \land_{\text{cond}_i \in \mathcal{C}_S^b} \text{cond}_i \]
\[ pftr = \text{active}(b_1) \land_{\forall b \in \mathcal{B}} \text{ann}(b) \land \text{entry}(b) \land \text{active}(b) \]
\[ \text{desc} = \text{entry}(b) \land btr \land \text{exit}(b, val, l_1, l_2) \]

\[ \langle F (fdcl, \overline{B} (\overline{C} c) B, \mathcal{M}_S^b, \mathcal{C}_S^b) \rightarrow P \rangle \langle F (fdcl, \overline{B} (\overline{C} c^{\langle ec, btr, pftr \rangle})^{\langle \text{desc} \rangle} B, \mathcal{M}_S^b, \emptyset) \rangle \]
Modelling links between blocks

\[
\begin{align*}
\text{entry}(b) & = \text{activating}(b) \land \text{initialize}(b) \\
\text{exit}(b, \text{val}, l_1, l_2) & = \text{jump}(b, \{(\text{block}(l_1), \text{val} = s \top), (\text{block}(l_2), \text{val} = s \bot)\}) \\
& \land \text{leaving}(b, \{\text{block}(l_1), \text{block}(l_2)\})
\end{align*}
\]

\[
\begin{align*}
\text{activating}(b) & = \begin{cases} \\
\left( \lor_{\text{pred} \in \text{preds}(b)} \text{transition}(\text{pred}, b) \right) \iff \text{active}(b), & \text{if } |\text{preds}(b)| > 1 \\
\text{transition}(\text{pred}, b) \iff \text{active}(b), & \text{if } \text{preds}(b) = \{\text{pred}\} \\
\top & \text{if } \text{preds}(b) = \emptyset \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{initialize}(b) & = \begin{cases} \\
\land_{\text{pred} \in \text{preds}(b)} (\text{transition}(\text{pred}, b) \Rightarrow \\
\land_{\text{id} \in \text{ID}} \text{final}(\text{pred}, \text{id}) = \text{init}(b, \text{id})), & \text{if } |\text{preds}(b)| > 1 \\
\text{transition}(\text{pred}, b) \Rightarrow \\
\land_{\text{id} \in \text{ID}} \text{final}(\text{pred}, \text{id}) = \text{init}(b, \text{id}), & \text{if } \text{preds}(b) = \{\text{pred}\} \\
\top & \text{if } \text{preds}(b) = \emptyset \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{jump}(b, S) & = \begin{cases} \\
\land_{(\text{succ}, c) \in S} ((\text{active}(b) \land c) \iff \text{transition}(b, \text{succ})), & \text{if } |S| > 1 \\
\text{active}(b) \iff \text{transition}(b, \text{succ}), & \text{if } S = \{\text{succ}, \top\} \\
\top, & \text{if } S = \emptyset \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{leaving}(b, S) & = \begin{cases} \\
\text{active}(b) \iff \lor_{\text{succ} \in S} \text{transition}(b, \text{succ}), & \text{if } |S| > 1 \\
\text{active}(b) \iff \text{transition}(b, \text{succ}), & \text{if } S = \{\text{succ}\} \\
\top, & \text{if } S = \emptyset \\
\end{cases}
\end{align*}
\]
**Description**

**Definition (Partial function description)**

For a program \( P \in \mathcal{P}_{\mathcal{F}_R} \), for a function \( f = (fdcl, b_1 b_2 \ldots b_n \ldots b_m) \) and a command \( c_k \) of a block \( b_n \), if it holds

\[
\langle F(fdcl, b_1 b_2 \ldots b_n \ldots b_m) F, M_S^c, \emptyset \rangle \xrightarrow{\ast}_P \langle F(fdcl, b_1^{\langle \text{desc}_1 \rangle} b_2^{\langle \text{desc}_2 \rangle} \ldots C c_k C \ldots b_m) F, M_S^{b_n : c_k - 1}, C_S^{b_n : (k-1)} \rangle
\]

\[
\langle F(fdcl, b_1^{\langle \text{desc}_1 \rangle} b_2^{\langle \text{desc}_2 \rangle} \ldots C c_k^{\langle ec, btr, pftr \rangle} c \ldots b_m) F, M_S^{b_n : c_k}, C_S^{b_n : k} \rangle
\]

a partial function description \( \mathcal{DE}(P, f : b_n : c_k) \) is defined as

\[
pftr \land btr
\]
Correspondence

Definition (Correspondence $\propto$)

We say that partial concrete execution $\mathcal{CE}^\circ$ corresponds to a model $M_{DE}^\circ$ of partial function description $DE^\circ$ and we write $\mathcal{CE}^\circ \propto M_{DE}^\circ$ if it holds

$$(\forall b \in (b_1 \ldots b_n))(\forall id \in ID)$$

$$\left( M^b_{C}^{'c_1}(id) = I_{M_{DE}^\circ} \left( M^b_{S}^{c_1}(id) \right) \right) \wedge$$

$$\left( M^b_{C}^{c_{last}}(id) = I_{M_{DE}^\circ} \left( M^b_{S}^{c_{last}}(id) \right) \right)$$
Execution and transition — by model

Definition (Model block execution and transition)

Let $\mathcal{DE}^\circ$ be a partial function description and $M_{\mathcal{DE}^\circ}$ be its model (in the standard BVA interpretation).

**model block execution** If $M_{\mathcal{DE}^\circ} \models active(b)$, we say that a block $b$ gets executed in the $M_{\mathcal{DE}^\circ}$ and we write $M_{\mathcal{DE}^\circ} \triangledown b$.

**model block transition** If $M_{\mathcal{DE}^\circ} \models transition(b_i, b_{i+1})$, we say that there is transition from block $b_i$ to block $b_{i+1}$ in $M_{\mathcal{DE}^\circ}$ and we write $M_{\mathcal{DE}^\circ} \triangledown tr(b_i, b_{i+1})$. 
Concrete and model execution: correspondence

Lemma (Concrete and model execution: correspondence)

Let $C\varepsilon^\circ$ be a partial concrete execution and $M_{DE^\circ}$ a model of partial function description $DE^\circ$. If it holds $C\varepsilon^\circ \triangleright M_{DE^\circ}$ then:

(a) $C\varepsilon^\circ \triangleright b$ iff $M_{DE^\circ} \triangleright b$.

(b) $C\varepsilon^\circ \triangleright tr(b_i, b_{i+1})$ iff $M_{DE^\circ} \triangleright tr(b_i, b_{i+1})$. 
Concrete and model execution: existence

Lemma (Existence of model execution)

For each partial concrete execution $CE^\circ$ there exists a model $M_{DE^\circ}$ of partial function description $DE^\circ$ such that $CE^\circ \bowtie M_{DE^\circ}$.

Lemma (Existence of concrete execution)

For each partial function description $DE^\circ$ and for its arbitrary model $M_{DE^\circ}$ there exists a concrete partial execution $CE^\circ$ such that $CE^\circ \bowtie M_{DE^\circ}$. 
SMT solving

Theory for bit-vector arithmetic (BVA) is decidable
There are several SMT solvers for BVA available: Boolector, Z3...

SMT solver for BVA is sound and complete
For any BVA formula $\phi$ it holds: there is a model $M$ of $\phi$ iff the SMT solver claims that $\phi$ is satisfiable and returns its model.
Properties of LAV

**Theorem**

For a function $f \in \mathcal{F}_R$, $LAV_R$ is sound and complete.

**Theorem**

For a function $f \in \mathcal{F}_R$, $LAV_R$ can reconstruct a concrete error trace for any erroneous command.
Conclusions

Ongoing work presented

- Modelling LLVM IR and the basic way LAV works
- Conjectures are given about soundness and completeness of LAV for a restricted class of programs

Currently working on ...

- Polishing models and proofs to be elegant — proofs are not surprising but involve many details
- Models and proofs should be easily extensible
Ongoing and further work

Further work

- Incremental/spiral development: going from very restricted to the full power of LLVM IR / LAV (a number of optimizations that should be formally justified, e.g. symbolic execution over several blocks, different levels of error conditions, parallelization)
- Ultimate goal: formalization within a proof assistant — it requires a huge amount of work for full, real world, LLVM IR / LAV
Thank you!