A calculus for a LLVM-based software verification tool LAV

Milena Vujošević Janičić

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?

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

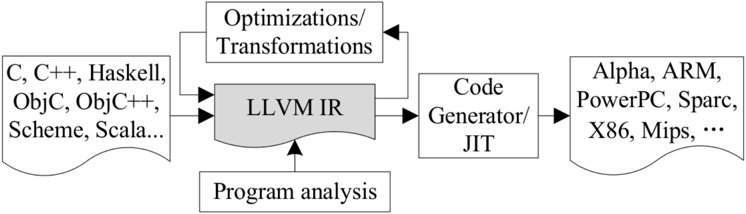
Start with a subset of LLVM IR and a subset of LAV

- 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 $\mathcal{F}_{\mathcal{R}}$ and programs consisting of such functions as $\mathcal{P}_{\mathcal{F}_{\mathcal{R}}}$
- 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

LLVM IR characteristics



- 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)

Modules	mod, P	::=	$\overline{layout}\ \overline{namedt}\ \overline{prod}$
Layouts	$layout$	::=	**bigendian** | **littleendian** | **ptr** $sz\ align_0\ align_1$ | **int** $sz\ align_0\ align_1$
		|	**float** $sz\ align_0\ align_1$ | **aggr** $sz\ align_0\ align_1$ | **stack** $sz\ align_0\ align_1$
Products	$prod$	::=	id = **global** $typ\ const\ align$ | **define** $typ\ id(\overline{arg})\{\overline{b}\}$ | **declare** $typ\ id(\overline{arg})$
Floats	fp	::=	**float** | **double**
Types	typ	::=	**i**sz | fp | **void** | $typ*$ | $[\,sz \times typ\,]$ | $\{\,\overline{typ_j}^{\,j}\,\}$ | $typ\ \overline{typ_j}^{\,j}$ | id
Values	val	::=	id | $cnst$
Binops	bop	::=	**add** | **sub** | **mul** | **udiv** | **sdiv** | **urem** | **srem** | **shl** | **lshr** | **ashr** | **and** | **or** | **xor**
Float ops	$fbop$	::=	**fadd** | **fsub** | **fmul** | **fdiv** | **frem**
Extension	eop	::=	**zext** | **sext** | **fpext**
Cast op	cop	::=	**fptoui** | **ptrtoint** | **inttoptr** | **bitcast**
Trunc op	$trop$	::=	$\mathbf{trunc}_{int}$ | $\mathbf{trunc}_{fp}$
Constants	$cnst$	::=	**i**$sz\ Int$ | $fp\ Float$ | $typ * id$ | $(typ*)$ **null** | typ **zeroinitializer** | $typ[\,\overline{cnst_j}^{\,j}\,]$ | $\{\,\overline{cnst_j}^{\,j}\,\}$
		|	typ **undef** | $bop\ cnst_1\ cnst_2$ | $fbop\ cnst_1\ cnst_2$ | $trop\ cnst$ **to** typ | $eop\ cnst$ **to** typ
		|	$cop\ cnst$ **to** typ | **getelementptr** $cnst\ \overline{cst_j}^{\,j}$ | **select** $cnst_0\ cnst_1\ cnst_2$ | **icmp** $cond\ cnst_1\ cnst_2$
		|	**fcmp** $fcond\ cnst_1\ cnst_2$
Blocks	b	::=	$l\ \overline{\phi}\ \overline{c}\ tmn$
ϕ nodes	ϕ	::=	id = **phi** $typ\ \overline{[val_j, l_j]}^{\,j}$
Tmns	tmn	::=	**br** $val\ l_1\ l_2$ | **br** l | **ret** $typ\ val$ | **ret void** | **unreachable**
Commands	c	::=	$id = bop($**int** $sz)val_1\ val_2$ | $id = fbop\ fp\ val_1\ val_2$ | id = **load** $(typ*)val_1\ align$
		|	**store** $typ\ val_1\ val_2\ align$ | id = **malloc** $typ\ val\ align$ | **free** $(\,typ *\,)\ val$
		|	id = **alloca** $typ\ val\ align$ | $id = trop\ typ_1\ val$ **to** typ_2 | $id = eop\ typ_1\ val$ **to** typ_2
		|	$id = cop\ typ_1\ val$ **to** typ_2 | id = **icmp** $cond\ typ\ val_1\ val_2$ | id = **select** $val_0\ typ\ val_1\ val_2$
		|	id = **fcmp** $fcond\ fp\ val_1\ val_2$ | $option\ id$ = **call** $typ_0\ val_0\ \overline{param}$
		|	id = **getelementptr** $(\,typ *\,)\ val\ \overline{val_j}^{\,j}$

LLVM IR example

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

```
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
}
```

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

```
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 definitons of LLVM IR semantics (1)

Application in cryptography

Corin and Manzano. Efficient Symbolic Execution for Analysing Cryptographic Protocol Implementations. ESSoS 2011.

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 definitons of LLVM IR semantics (2)

Application in program transformations and compilation

Zhao, Nagarakatte, Martin, and Zdancewic. Formalizing the LLVM IR for Verified Program Transformations. POPL '12. 427-440.

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 $\mathcal{M}_\mathcal{C}$
- The semantics is described in terms of a transition system with states $\langle f : b_n : c_k, \mathcal{M}_\mathcal{C} \rangle$ 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:

bop type op_1, op_2	error kind	error condition
add	overflow	$signed(type) \wedge op_1 > 0$ $\wedge op_2 > 0 \wedge op_1 + op_2 < 0$
sub		$signed(type) \wedge op_1 > 0$ $\wedge op_2 < 0 \wedge op_1 + op_2 < 0$
sdiv		$signed(type) \wedge op_1 = min\_value(type)$ $\wedge op_2 = -1$
mul		$signed(type) \wedge op_2 > 0$ $\wedge op_1 > max\_value(type)/op_2$
udiv, sdiv, urem, srem	division by zero	$op_2 = 0$

Some concrete semantics rules

Binary operations

$$\frac{c = (\mathsf{id} = \mathsf{bop\ t}\ op_1, op_2) \qquad \wedge_i \neg cond_i(err\_kind_i, bop(t, \mathcal{M}_\mathcal{C}(op_1), \mathcal{M}_\mathcal{C}(op_2)))}{\langle f : b_n : c_k, \mathcal{M}_\mathcal{C} \rangle \rightarrow_\mathcal{P} \langle f : b_n : c_{k+1}, \mathcal{M}_\mathcal{C}\{id \mapsto (t, \mathcal{M}_\mathcal{C}(op_1)\ bop\ \mathcal{M}_\mathcal{C}(op_2))\} \rangle} \text{ BOP}$$

$$\frac{c = (\mathsf{id} = \mathsf{bop\ t}\ op_1, op_2) \qquad cond_i(err\_kind_i, bop(t, \mathcal{M}_\mathcal{C}(op_1), \mathcal{M}_\mathcal{C}(op_2)))}{\langle f : b_n : c_k, \mathcal{M}_\mathcal{C} \rangle \rightarrow_\mathcal{P} \mathcal{ERR}} \text{ BOPerr}_i$$

Branching

$$\frac{\begin{array}{c} c = (\mathsf{br}\ val\ l_1\ l_2) \\ \mathcal{M}_\mathcal{C}(val) = \top \qquad block(l_1) = b_t \end{array}}{\langle f : b_n : c_k, \mathcal{M}_\mathcal{C} \rangle \rightarrow_\mathcal{P} \langle f : b_t : c_1, \mathcal{M}_\mathcal{C} \rangle} \text{ BRT} \qquad \frac{\begin{array}{c} c = (\mathsf{br}\ val\ l_1\ l_2) \\ \mathcal{M}_\mathcal{C}(val) = \bot \qquad block(l_2) = b_f \end{array}}{\langle f : b_n : c_k, \mathcal{M}_\mathcal{C} \rangle \rightarrow_\mathcal{P} \langle f : b_f : c_1, \mathcal{M}_\mathcal{C} \rangle} \text{ BRF}$$

Execution

Definition (Partial concrete execution)

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

$$s_1 = \langle f : b_1 : c_1, \mathcal{M}_{\mathcal{C}_{[a_1,\ldots a_i]}} \rangle \quad \overset{*}{\to}_{\mathcal{P}} \quad \langle f : b_n : c_k, \mathcal{M}_{\mathcal{C}}^{b_n:c_{k-1}} \rangle$$
$$\to_{\mathcal{P}} \quad \langle f : b : c, \mathcal{M}_{\mathcal{C}}^{b_n:c_k} \rangle = s_l$$

(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 $s_1 s_2 ... s_l$.

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

concrete block transition If there exists v such that $v \in \{1, \ldots, (l-1)\}$ and $s_v = \langle f : b_i : c_j, \mathcal{M_C} \rangle \rightarrow_{\mathcal{P}} \langle f : b_{i+1} : c_1, \mathcal{M}'_{\mathcal{C}} \rangle = s_{v+1}$ are in $\mathcal{CE}^\circ$, we say that there is transition from block b_i to block b_{i+1} and we write $\mathcal{CE}^\circ \blacktriangleright tr(b_i, b_{i+1})$.

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, ... 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 $\prec_b \subseteq \mathcal{B} \times \mathcal{B}$ 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 $\mathcal{B}$
- A sequence of blocks b_1, b_2, ... 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 $\langle \overline{F}\ (fdcl, \overline{B}\ (\overline{C}\ c\ C\ )B)\ F, \mathcal{M}_{\mathcal{S}}^{b}, \mathcal{C}_{\mathcal{S}}^{b} \rangle$

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

Some transition rules

Binary operations

$$\dfrac{\begin{array}{c} c = (\textsf{id} = \textsf{bop t}\ op_1, op_2) \qquad ec = \vee_i cond(err\_kind, bop(t, \mathcal{M}^b_{\mathcal{S}}(op_1), \mathcal{M}^b_{\mathcal{S}}(op_2))) \\ btr = \bigwedge_{id \in \mathcal{ID}} (final(b, id) = \mathcal{M}^b_{\mathcal{S}}(id)) \bigwedge_{cond_i \in \mathcal{C}^b_{\mathcal{S}}} cond_i \\ pftr = active(b_1) \bigwedge_{\forall b \in \overline{B}} ann(b) \bigwedge entry(b) \wedge active(b) \end{array}}{\begin{array}{c} \langle \overline{F}\ (fdcl, \overline{B}\ \left(\overline{C}\ cC\right)\ B\ F, \mathcal{M}^b_{\mathcal{S}}, \mathcal{C}^b_{\mathcal{S}} \rangle \rightsquigarrow_{\mathcal{P}} \\ \langle \overline{F}\ (fdcl, \overline{B}\ \left(\overline{C} c^{\langle ec, btr, pftr \rangle} C\right)\ B)\ F, \mathcal{M}^b_{\mathcal{S}}\{id \mapsto (t, \mathcal{M}^b_{\mathcal{S}}(op_1)\ sbop\ \mathcal{M}^b_{\mathcal{S}}(op_2))\}, \mathcal{C}^b_{\mathcal{S}} \rangle \end{array}} \text{ SBOP}$$

Branching instruction

$$\dfrac{\begin{array}{c} c = (\textsf{br}\ val\ l_1\ l_2) \qquad ec = \bot \\ btr = \bigwedge_{id \in \mathcal{ID}} (final(b, id) = \mathcal{M}^b_{\mathcal{S}}(id)) \bigwedge_{cond_i \in \mathcal{C}^b_{\mathcal{S}}} cond_i \\ pftr = active(b_1) \bigwedge_{\forall b \in \overline{B}} ann(b) \bigwedge entry(b) \wedge active(b) \\ desc = entry(b) \wedge btr \wedge exit(b, val, l_1, l_2) \end{array}}{\langle \overline{F}\ (fdcl, \overline{B}\ \left(\overline{C} c\right)\ B, \mathcal{M}^b_{\mathcal{S}}, \mathcal{C}^b_{\mathcal{S}} \rangle \rightsquigarrow_{\mathcal{P}} \langle \overline{F}\ (fdcl, \overline{B}\ \left(\overline{C} c^{\langle ec, btr, pftr \rangle}\right)^{\langle desc \rangle}\ B, \mathcal{M}^{\epsilon}_{\mathcal{S}}, \emptyset \rangle} \text{ SBR}$$

Modelling links between blocks

$$
\begin{aligned}
entry(b) &= activating(b) \wedge initialize(b)\\
exit(b, val, l_1, l_2) &= jump(b, \{(block(l_1), val =_s \top), (block(l_2), val =_s \bot)\})\\
&\quad \wedge leaving(b, \{block(l_1), block(l_2)\})
\end{aligned}
$$

$$
activating(b) = \begin{cases} \left(\bigvee_{pred \in preds(b)} transition(pred, b)\right) \Leftrightarrow active(b), & \text{if } |preds(b)| > 1\\ transition(pred, b) \Leftrightarrow active(b), & \text{if } preds(b) = \{pred\}\\ \top & \text{if } preds(b) = \emptyset \end{cases}
$$

$$
initialize(b) = \begin{cases} \bigwedge_{pred \in preds(b)} (transition(pred, b) \Rightarrow & \\ \bigwedge_{id \in \mathcal{ID}} final(pred, id) = init(b, id)), & \text{if } |preds(b)| > 1\\ transition(pred, b) \Rightarrow & \\ \bigwedge_{id \in \mathcal{ID}} final(pred, id) = init(b, id), & \text{if } preds(b) = \{pred\}\\ \top & \text{if } preds(b) = \emptyset \end{cases}
$$

$$
jump(b, \mathcal{S}) = \begin{cases} \bigwedge_{(succ, c) \in \mathcal{S}} ((active(b) \wedge c) \Leftrightarrow transition(b, succ)), & \text{if } |\mathcal{S}| > 1\\ active(b) \Leftrightarrow transition(b, succ), & \text{if } \mathcal{S} = \{succ, \top\}\\ \top, & \text{if } \mathcal{S} = \emptyset \end{cases}
$$

$$
leaving(b, \mathcal{S}) = \begin{cases} active(b) \Leftrightarrow \bigvee_{succ \in \mathcal{S}} transition(b, succ), & \text{if } |\mathcal{S}| > 1\\ active(b) \Leftrightarrow transition(b, succ), & \text{if } \mathcal{S} = \{succ\}\\ \top, & \text{if } \mathcal{S} = \emptyset \end{cases}
$$

Description

Definition (Partial function description)

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

$$
\begin{array}{ll}
 & \langle \overline{F}(fdcl, b_1 b_2 \dots b_n \dots b_m)F, \mathcal{M}_\mathcal{S}^\epsilon, \emptyset \rangle \\
\rightsquigarrow^*_\mathcal{P} & \langle \overline{F}(fdcl, b_1^{\langle desc_1 \rangle} b_2^{\langle desc_2 \rangle} \dots \overline{C} c_k C \dots b_m)F, \mathcal{M}_\mathcal{S}^{b_n : c_{k-1}}, \mathcal{C}_\mathcal{S}^{b^{n:(k-1)}} \rangle \\
\rightsquigarrow_\mathcal{P} & \langle \overline{F}(fdcl, b_1^{\langle desc_1 \rangle} b_2^{\langle desc_2 \rangle} \dots \overline{C} c_k^{\langle ec, btr, pftr \rangle} c C \dots b_m)F, \mathcal{M}_\mathcal{S}^{b_n : c_k}, \mathcal{C}_\mathcal{S}^{b^{n:k}} \rangle
\end{array}
$$

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

$$pftr \wedge btr$$

Correspondence

Definition (Correspondence $\bowtie$)

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

$$(\forall b \in (b_1 \dots b_n))(\forall id \in \mathcal{ID})$$
$$\left(\mathcal{M}_{\mathcal{C}}^{b:c_1}(id) = I_{M_{\mathcal{DE}^\circ}}\left(\mathcal{M}_{\mathcal{S}}^{b:c_1}(id)\right)\right) \wedge$$
$$\left(\mathcal{M}_{\mathcal{C}}^{b:c_{last}}(id) = I_{M_{\mathcal{DE}^\circ}}\left(\mathcal{M}_{\mathcal{S}}^{b:c_{last}}(id)\right)\right)$$

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 $M_{\mathcal{DE}^\circ} \models acitve(b)$, we say that a block b gets executed in the $M_{\mathcal{DE}^\circ}$ and we write $M_{\mathcal{DE}^\circ} \triangleright b$.

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 $M_{\mathcal{DE}^\circ} \triangleright tr(b_i, b_{i+1})$.

Concrete and model execution: correspondence

Lemma (Concrete and model execution: correspondence)

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

(a) $\mathcal{CE}^{\circ} \blacktriangleright b$ *iff* $M_{\mathcal{DE}^{\circ}} \triangleright b$.

(b) $\mathcal{CE}^{\circ} \blacktriangleright tr(b_i, b_{i+1})$ *iff* $M_{\mathcal{DE}^{\circ}} \triangleright tr(b_i, b_{i+1})$.

Concrete and model execution: existence

Lemma (Existence of model execution)

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

Lemma (Existence of concrete execution)

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

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 ϕ it holds: there is a model M of ϕ iff the SMT solver claims that ϕ is satisfiable and returns its model.

Properties of LAV

Theorem

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

Theorem

For a function $f \in \mathcal{F}_{\mathcal{R}}$, 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!