Value Set Analysis in LLVM

Julian Erhard, Jakob Gottfriedsen, Peter Munch, Alexander Roschlaub, Michael Schwarz

IN2053 - Program Optimization Lab 2018

June 21, 2018

Part 0:

Introduction

Value Set Analysis

Bounded Set Analysis:

$$Var \rightarrow \{a, b, c, ...\}_n$$

Interval Analysis:

$$Var \rightarrow [a : b]_n$$

Strided Interval Analysis:

$$Var \rightarrow s[a : b]_n$$

SSA: sufficient to store the abstract value for each variable once per basic block:[1]

$$\mathcal{D} : BB \rightarrow Var \rightarrow Val$$

[1] The need to store it not just once only arises to preserve information at conditional branches.

Passes in LLVM

LLVM's analysis and optimization framework opt:

Figure: stages of clang and LLVM

using existing passes (from command line):

```
opt -load -mem2reg -o hello-opt.bc < hello.bc
```

running user passes:

```
opt -load llvm/lib/llvm-vsa.so -vsapass -o hello.bc < hello.bc
```

Passes in LLVM (cont.)

Creating user passes:

- inherit from existing passes (module, function, block):

```
struct ThisPass : public ModulePass {}
```

- specify required passes, which have to be run in advance:

```
void ThisPass::getAnalysisUsage(AnalysisUsage &AU) override {
  AU.setPreservesAll();
  AU.addRequired<OtherPass>();
}
```

- perform analysis by using/accessing results of other passes:

```
bool ThisPass::runOnModule(Module &M) override {
  auto& other_result =
    getAnalysis<OtherPass>(function).getResult();
  /* ... perform analysis and fill result ... */

  // Return if the pass modified the bitcode (no)
  return false;
}
```

- make results available for other pass (optional):

```
ThisResult& ThisPass::getResult(){ return result; }
```

Content of the Lab

Tasks:

- implement abstract domains that suitably represent value sets
- develop a new analysis tool in LLVM to determine the value set of each variable, using visitor and fixpoint algorithm (worklist) $\triangleright$ VSAPass
- make results accessible via API: VSAResult and VSAResultValue
- compare results with LLVM's LazyValueInfo

Future work:

- widening and narrowing
- inter-procedural analysis
- memory access

unknowns (ops, return values and arguments of functions) treated as: $\top$

Part 1:

Abstract Domain

Background to LLVM Integer Types

- In LLVM the type of N-bit integers is the set $\mathtt{i}N := \{0,1\}^N$, for $N \in \{1, \ldots, 2^{23} - 1\}$.
- "$\mathtt{i}N \cong \mathbb{Z}/2^N$ "
- In the in-memory-representation, these types are represented by the LLVM class APInt.
- This type is used for both signed and unsigned integers.
- We use APInt in our implementation of abstract domains.

LLVM Integer Operations

Arithmetic Operations:

In LLVM there are separate `div` and `rem` operations for signed and unsigned integers. For `add`, `sub` and `mul`, there is no such distinction needed.

- `<result> = add [nuw] [nsw] <bitWidth> <op1> <op2>`
- `<result> = sub [nuw] [nsw] <bitWidth> <op1> <op2>`
- `<result> = mul [nuw] [nsw] <bitWidth> <op1> <op2>`
- `<result> = udiv [exact] <bitWidth> <op1> <op2>` [2]
- `<result> = sdiv [exact] <bitWidth> <op1> <op2>` [2]
- `<result> = urem <bitWidth> <op1> <op2>`
- `<result> = srem <bitWidth> <op1> <op2>`

`nuw` : "no unsigned wrap", `nsw` : "no signed wrap"

[2] exact-flag not used in our implementation.

LLVM Integer Operations, Continued

Bitwise Operations:

- `<result> = shl [nuw] [nsw] <bitWidth> <op1> <op2>`
- `<result> = lshr [exact] <bitWidth> <op1> <op2>` [3]
- `<result> = ashr [exact] <bitWidth> <op1> <op2>` [3]
- `<result> = and <bitWidth> <op1> <op2>`
- `<result> = or <bitWidth> <op1> <op2>`
- `<result> = xor <bitWidth> <op1> <op2>`

[3] exact-flag not used in our implementation.

Bounded Set

- A bounded set represents a set of values up to a given cardinality k, or $\top$: $\mathrm{BS}_N := \{M \in \mathcal{P}(\mathtt{i}N) \mid |M| \leq k\} \dot{\cup} \{\top\}$
- $\sqcup$ and $\sqsubseteq$ on bounded sets essentially reduce to $\cup$ and $\subseteq$ on sets.
- Any set with more elements than k is over-approximated by $\top$.
- $\gamma_{BS_N} : \mathrm{BS}_N \to \mathcal{P}(\mathtt{i}N), b \mapsto \begin{cases} \mathtt{i}N, & \text{if } b = \top \\ b, & \text{otherwise} \end{cases}$

Modular Strided Interval

- Intervals:
  - $\mathrm{I} := [a, b]$, for $a, b \in \mathbb{Z}$
- Strided Intervals:
  - $\mathrm{SI} := s[a, b]$, for $a, b \in \mathbb{Z}, s \in \mathbb{N}$
  - $\gamma_{\mathrm{SI}}, s[a, b] \mapsto \{k \in \mathbb{Z} \mid a \leq k \leq b, k \equiv a \mod s\}$
- Modular Strided Intervals:
  - $\mathrm{MSI}_N := \{\bar{s}[\bar{a}, \bar{b}]_N \mid \bar{a}, \bar{b}, \bar{s} \in \mathbb{Z}/2^N\} \dot{\cup} \{\perp\}$
  - $\gamma_{\mathrm{MSI}_N}, s[a, b]_N \mapsto \{k + 2^N\mathbb{Z} \mid k \in \mathbb{Z}, a \leq k \leq z, k \equiv a \mod s\}$, where $z = \min\{l \in \mathbb{Z} \mid l \geq a, l \equiv b \mod 2^N\}$
  - Examples:
    - $12[15, 63]_8 \xrightarrow{\gamma} \{15, 27, 39, 51, 63\} \subseteq \mathbb{Z}/2^8$
    - $4[10, 6]_4 \xrightarrow{\gamma} \{10, 14, 2, 6\} \subseteq \mathbb{Z}/2^4$

Modular Strided Interval: Normalization

Note that the representation of a set by a modular strided interval may not be unique. Thus, we introduce a predicate *normal*, such that there is always a unique *normalized* representation. Example:

$\mathrm{normal}_N(\overline{s}[\overline{a}, \overline{b}]_N) \leftrightarrow ($

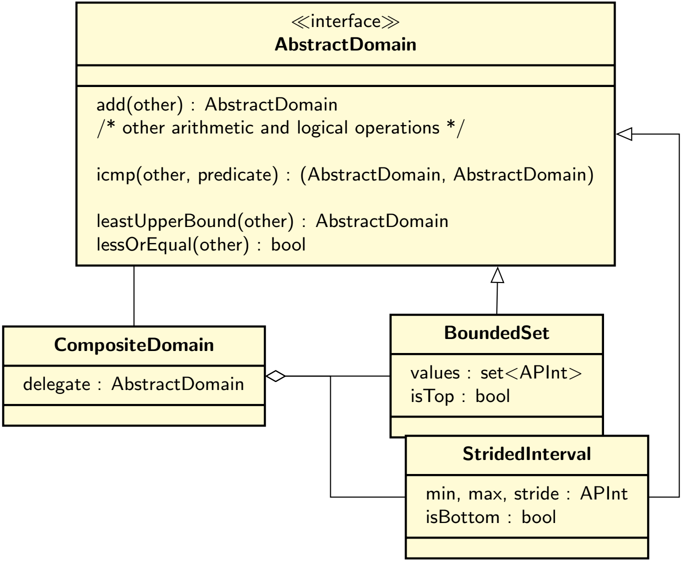
$$
\begin{aligned}
&\overline{s} = 0 \leftrightarrow \overline{a} = \overline{b} \\
\wedge\ &\overline{b} \in \gamma_N(\overline{s}[\overline{a}, \overline{b}]_N) \\
\wedge\ &\overline{a} = \min\{a' \in \{0 \ldots 2^N - 1\}.\, \gamma_N(\overline{s}[\overline{a'}, \overline{b}]_N) = \gamma_N(\overline{s}[\overline{a}, \overline{b}]_N)\} + 2^N\mathbb{Z}
\end{aligned}
$$

$)$

Modular Strided Interval: Union

Modular strided intervals do not form a lattice, as, in general, there is no least upper bound of two elements.
Example:

Abstract Domain Class Structure



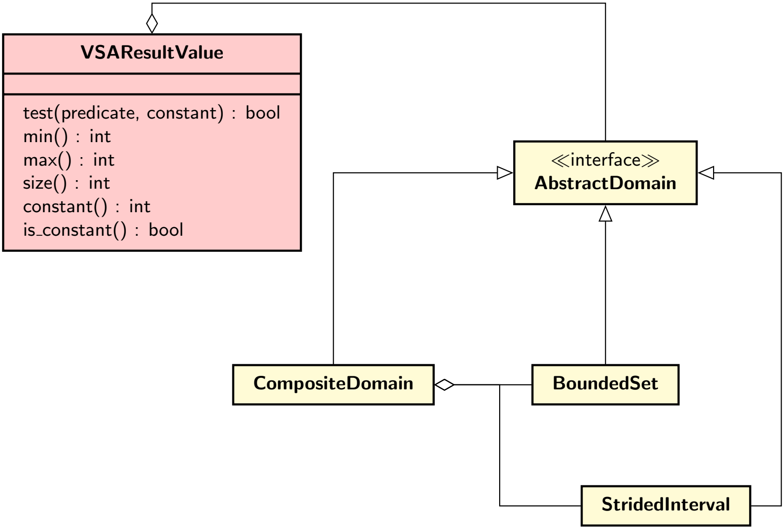
Application Interface

VSAResult
is_reachable(basic_block) : bool is_resultat_available(bb, value) : bool get_abstract_value() : VSAResultValue

VSAResultValue
test(predicate, constant) : tristate min() : int max() : int size() : int constant() : int is_constant() : bool

- after a successful pass:
  `auto& res = vsap.get_result();`
- query information related to basic block (reachable or not) and/or variable (abstract value)

Connection of the Results to the Internal Abstract Domain



Part 2:

Fixpoint Algorithm & Visitor

Data Structures

Data structures maintained during the analysis:

- state of each basic block (goal): red: abstract domain

$$\mathcal{D} \;:\; BB \to \underbrace{(Var \to Val)_\perp}_{state\ \mathcal{D}_{BB}}$$

- branch conditions:

$$\mathcal{C} \;:\; \underbrace{(BB \to XX)}_{edge} \to \underbrace{(Var \to Val)_\perp}_{\mathcal{C}_{BB \to XX}}$$

$\to$ effect of a guard:

$$\mathcal{D}_{XX} \leftarrow [\![BB \;\to\; XX]\!]\, \mathcal{D}_{BB} = \mathcal{D}_{BB} \,\oplus\, \mathcal{C}_{BB \to XX}$$

Fixpoint Algorithm: Worklist

We maintain a worklist $\mathcal{W}$ of basic blocks to be (re)evaluated.

Algorithm 1 Fixpoint algorithm

procedure $\textsc{Fixpoint}$(Function)
    $\mathcal{W}$.push(Function.front()) ▷ push entry basic block of function
    while $!\,\mathcal{W}.\mathit{empty}()$ **then**:
        visit $\mathcal{W}$.pop()

Fixpoint algorithm terminates iff $\mathcal{W}$ is empty: a fixpoint has been found.

Following initial and boundary conditions are used (forward analysis):

$$\mathcal{D}_0 \,:\, BB \to \bot \quad \text{and} \quad \mathcal{D}_{BC} \,:\, BB \to (Var \to \top)$$

Visitor: (Entering) Basic Block

Visiting a basic block is a two-step process:

1. setting up the (temporary) input state $\mathcal{N}_{BB}$ during entering
2. visiting all its instructions

Algorithm 2 Enter basic block BB

procedure $\textsc{Visit}$(BB)
    $\mathcal{N}_{BB} = \bigsqcup \{ \mathcal{D}_{XX} \oplus \mathcal{C}_{XX \to BB} \mid XX \in \mathrm{prev}(BB) \wedge \mathcal{D}_{XX} \neq \perp \}$
    for each *instruction* $\in$ instructions(BB):
        visit *instruction*

Explicitly considered instructions:

- terminators: (un)conditional jumps, switches
- PHI nodes
- binary expressions

Visitor: Leaving Basic Block at Terminator

check for change of the local state and save it in $\mathcal{D}$:

Algorithm 3 Visit terminator

procedure Visit(Terminator)
  if $\mathcal{N}_{BB} \sqsubseteq \mathcal{D}_{BB}$ **then**: ▷ red: delegated to abstract domain
    return ▷ state has not changed
  $\mathcal{D}_{BB} \leftarrow \mathcal{N}_{BB}$
  for each $XX \in \text{next}(BB)$:
    if reachable(BB, XX) **then**:
      $\mathcal{W}$.push(XX)

in the case of change ($\mathcal{N}_{BB} \not\sqsubseteq \mathcal{D}_{BB}$), push all reachable successors:

- unconditional branch: push all successors
- conditional branch/switch: check if $\exists(\# \rightarrow \bot) \in \mathcal{C}_{BB \rightarrow XX}$

Visitor: Conditional Branch

Algorithm 4 Visit conditional branch (terminator)

procedure $\textsc{Visit}(\textsc{jmp}\,(x \,\square\, y \,?\, XX \;:\; YY))$
  if isVar(x) **then**:
    $\mathcal{C}_{BB \to XX} \leftarrow \mathcal{C}_{BB \to XX} \oplus \left\{x \to \mathcal{N}_{BB}[x] \;\square^{\#}\, \mathcal{N}_{BB}[y]\right\}$
    $\mathcal{C}_{BB \to YY} \leftarrow \mathcal{C}_{BB \to YY} \oplus \left\{x \to \mathcal{N}_{BB}[x] \,!\square^{\#}\, \mathcal{N}_{BB}[y]\right\}$
  if isVar(y) **then**:
    ...
  $\textsc{VisitTerminator}()$

considered comparisons:

$$\square \in \{=, \neq, <, \leq, \geq, >\}$$

Visitor: Switch

Algorithm 5 Visit switch (terminator)

procedure $\textsc{Visit}(\textsc{switch}\,[x = a\ :\ XX][x = b\ :\ YY][x = c\ :\ YY][\mathit{default}\ :\ ZZ])$

$\quad \mathcal{C}_{BB \to XX} \leftarrow \{x \to \mathcal{N}_{BB}[x] =^{\#} a\}$

$\quad \mathcal{C}_{BB \to YY} \leftarrow \{x \to \mathcal{N}_{BB}[x] =^{\#} b \sqcup c\}$

$\quad \mathcal{C}_{BB \to ZZ} \leftarrow \{x \to \mathcal{N}_{BB}[x] \setminus \{a, b, c\}\}$

$\quad \textsc{VisitTerminator}()$

Visitor: Parallel Assignments at PHI Node

Algorithm 6 PHI node in basic block BB

1: **procedure** $\textsc{Phi}(x \leftarrow [YY\ : y]\,[ZZ\ : z])$

2: $\quad \mathcal{N}_{BB} \leftarrow \mathcal{N}_{BB} \oplus \{x \rightarrow (\mathcal{D}_{YY} \oplus C_{YY\rightarrow BB})\,[y] \sqcup (\mathcal{D}_{ZZ} \oplus C_{ZZ\rightarrow BB})\,[z]\}$

Visitor: Binary Expressions

Algorithm 7 Addition in basic block BB

1: **procedure** $\textsc{Binary}(x \leftarrow y \,\square\, z)$
2: $\quad \mathcal{N}_{BB} \leftarrow \mathcal{N}_{BB} \oplus \left\{x \rightarrow \mathcal{N}_{BB}[y] \,\square^{\#}\, \mathcal{N}_{BB}[z]\right\}$

---

considered binary instructions:

$$\square \in \{+,\ -,\ \times,\ /,\ \%,\ \ll,\ \gg\}$$

Visitor: Memory access and Not-implemented Operations

Data in memory is considered unknown.

Algorithm 8 Load in basic block BB

1: **procedure** $\textsc{Binary}(x \leftarrow \textsc{load}(...))$
2: $\quad \mathcal{N}_{BB} \leftarrow \mathcal{N}_{BB} \oplus \{x \rightarrow \top\}$

Not-implemented operations of form $x \leftarrow \#$ are treated implicitly in the same way.

Part 3:

Livedemo

Value Set Analysis in LLVM

Julian Erhard, Jakob Gottfriedsen, Peter Munch, Alexander Roschlaub, Michael Schwarz

IN2053 - Program Optimization Lab 2018

June 21, 2018