

Value Set Analysis in LLVM

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

IN2053 - Program Optimization Lab 2018

June 22, 2018

Value Set Analysis

Bounded Set Analysis:

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

Interval Analysis:

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

Strided Interval Analysis:

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

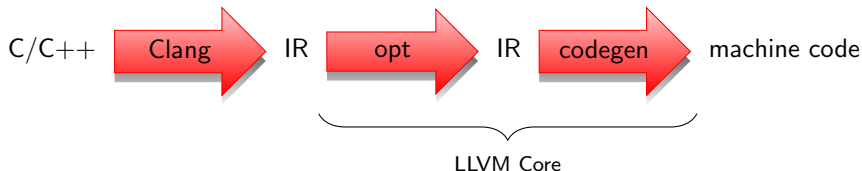
SSA \Rightarrow sufficient to store the **abstract value for each variable once per basic block**:¹

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

¹The 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`:



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; }
```

Example

```
int main(int argc, char const *argv[]) {
    unsigned a = 0, b = 12, c = rand();

    while (a < b) { a+=4; b-=2; }

    if(a>6 && b<6){
        switch (a) {
            case 6: b = 99; break;           // reachable?
            case 12:
            case 13: b = a*2; break;
            default:
                c = c%18;
        }
    } else {
        a = 88;
    }

    printf("%d\n", a);                      // what will/might be printed out?
    printf("%d\n", b);
    printf("%d\n", c);
}
```

LLVM's Intermediate Representation IR

```
define dso_local i32 @main(i32 %argc, i8** %argv) #0 {  
entry:  
    br label %while.cond  
  
while.cond:  
    %a.0 = phi i32 [ 0, %entry ], [ %add, %while.body ]  
    %b.0 = phi i32 [ 12, %entry ], [ %sub, %while.body ]  
    %cmp = icmp ult i32 %a.0, %b.0  
    br i1 %cmp, label %while.body, label %while.end  
  
while.body:  
    %add = add i32 %a.0, 4  
    %sub = sub i32 %b.0, 2  
    br label %while.cond  
  
while.end:  
    %call = call i32 @rand() #3  
    %cmp1 = icmp ugt i32 %a.0, 6  
    br i1 %cmp1, label %land.lhs.true, label %if.else  
  
land.lhs.true:  
    %cmp2 = icmp ult i32 %b.0, 6  
    br i1 %cmp2, label %if.then, label %if.else
```

... and many more lines of code

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) ▷ [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: [T](#)

Part 1:

Abstract Domain

Background to LLVM Integer Types

- In LLVM the type of N -bit integers is the set $iN := \{0, 1\}^N$, for $N \in \{1, \dots, 2^{23} - 1\}$.
- " $iN \cong \mathbb{Z}/2^N$ "
- 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 (cont.)

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 :
$$\text{BS}_N := \{M \in \mathcal{P}(\mathfrak{i}N) \mid |M| \leq k\} \dot{\cup} \{\top\}$$
- \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 .
- $\gamma_{\text{BS}_N}: \text{BS}_N \rightarrow \mathcal{P}(\mathfrak{i}N), b \mapsto \begin{cases} \mathfrak{i}N, & \text{if } b = \top \\ b, & \text{otherwise} \end{cases}$

Modular Strided Interval (MSI)

- Intervals:

- ▶ $I := [a, b]$, for $a, b \in \mathbb{Z}$

- Strided Intervals:

- ▶ $SI := s[a, b]$, for $a, b \in \mathbb{Z}, s \in \mathbb{N}$
- ▶ $\gamma_{SI}: SI_N \rightarrow \mathcal{P}(\mathbb{Z}), s[a, b] \mapsto \{k \in \mathbb{Z} \mid a \leq k \leq b, k \equiv a \pmod{s}\}$

- Modular Strided Intervals:

- ▶ $MSI_N := \{s[\bar{a}, \bar{b}]_N \mid \bar{a}, \bar{b} \in \mathbb{Z}/2^N, s \in \{0, \dots, 2^N\}\} \dot{\cup} \{\perp\}$
- ▶ $\gamma_{MSI_N}: MSI_N \rightarrow \mathcal{P}(\mathbb{N}),$

$$i \mapsto \begin{cases} \emptyset, & \text{if } i = \perp \\ \{k + 2^N \mathbb{Z} \mid k \in \mathbb{Z}, a \leq k \leq c, k \equiv a \pmod{s}, \\ \quad \text{where } c = \min\{x \in \mathbb{Z} \mid x \geq a, \\ \quad \quad x \equiv b \pmod{2^N}\} \} & , \text{ if } i = s[\bar{a}, \bar{b}]_N \end{cases}$$

- ▶ 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

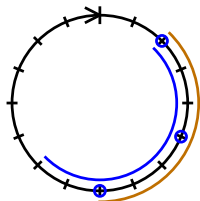
Note that some sets can be represented by multiple MSIs. Thus, we introduce a predicate *normal*, such that there is at most one *normalized* representation.

Examples:

$$3[2, 10]_4$$

↓

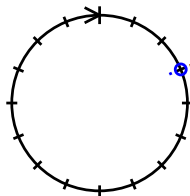
$$3[2, 8]_4$$



$$4[3, 3]_4$$

↓

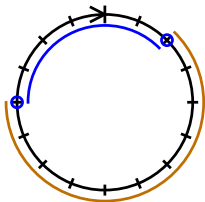
$$0[3, 3]_4$$



$$6[12, 2]_4$$

↓

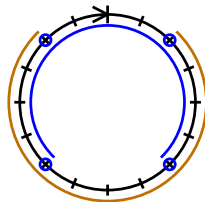
$$10[2, 12]_4$$



$$4[10, 6]_4$$

↓

$$4[2, 14]_4$$



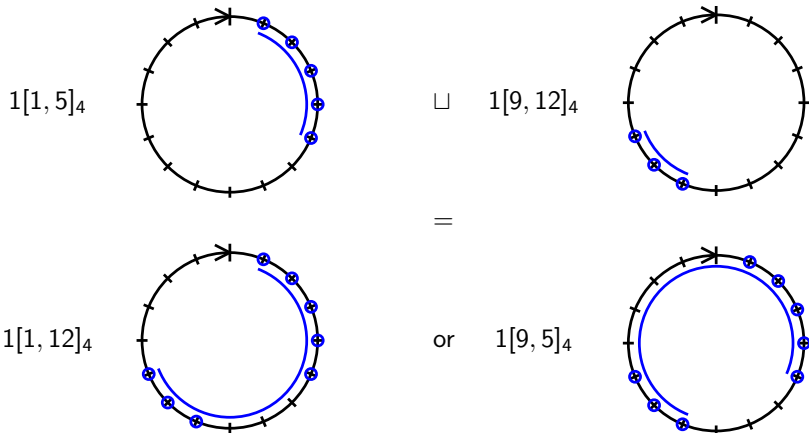
Modular Strided Interval: Normalization (cont.)

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

Modular Strided Interval: Union

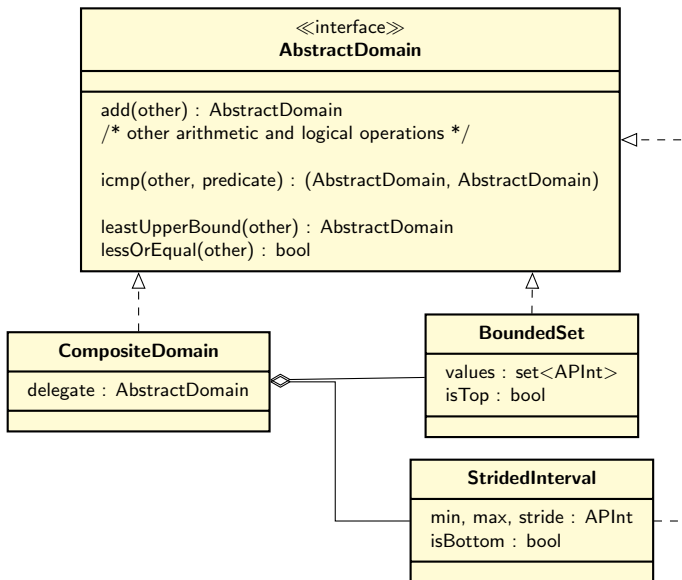
MSIs do not form a lattice, as, in general, there is no *least* upper bound of two elements.

Example:



Therefore we try to find a *minimal* upper bound wrt. $|\gamma(\cdot)|$.

Abstract Domain Class Structure



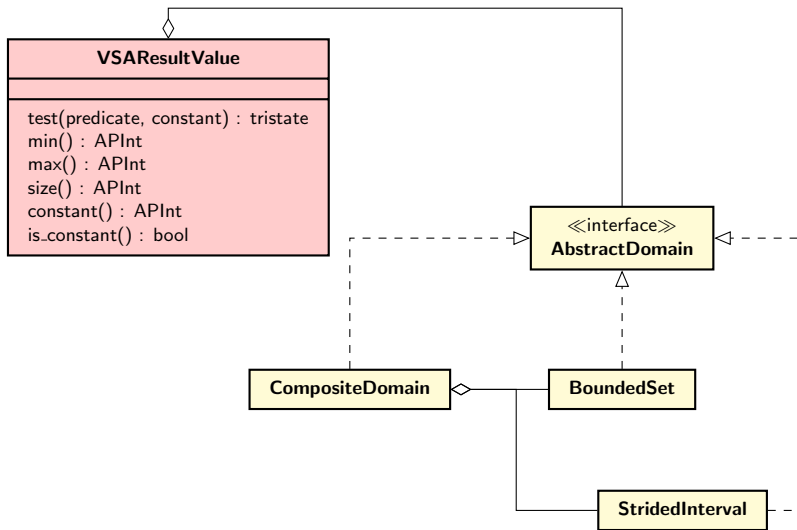
Application Interface

VSAResult
<code>is_reachable(basic_block) : bool</code> <code>is_resultat_available(bb, value) : bool</code> <code>get_abstract_value() : VSAResultValue</code>

VSAResultValue
<code>test(predicate, constant) : tristate</code> <code>min() : APInt</code> <code>max() : APInt</code> <code>size() : APInt</code> <code>constant() : APInt</code> <code>is_constant() : bool</code>

- 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

$$\mathcal{D} : BB \rightarrow \underbrace{(Var \rightarrow \text{red } Val)}_{\text{state } \mathcal{D}_{BB}} \perp$$

- **branch conditions:**

$$\mathcal{C} : \underbrace{(BB \rightarrow XX)}_{\text{edge}} \rightarrow \underbrace{(Var \rightarrow \text{red } Val)}_{\mathcal{C}_{BB \rightarrow XX}} \perp$$

→ effect of a guard:

$$\mathcal{D}_{XX} \leftarrow \llbracket BB \rightarrow XX \rrbracket \mathcal{D}_{BB} = \mathcal{D}_{BB} \oplus \mathcal{C}_{BB \rightarrow XX}$$

Fixpoint Algorithm: Worklist

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

Algorithm 1 Fixpoint algorithm

```
1: procedure FIXPOINT(Function)
2:    $\mathcal{W}.\text{push}(\text{Function}.\text{front}())$             $\triangleright$  push entry basic block of function
3:   while ! $\mathcal{W}.\text{empty}()$  then:
4:     visit  $\mathcal{W}.\text{pop}()$ 
```

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

Following **initial and entry states** are used (forward analysis):

$$\mathcal{D}_0 : BB \rightarrow \perp \quad \text{and} \quad \mathcal{D}_{\text{entry}} : BB \rightarrow (Arg_i \rightarrow \top)$$

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

```
1: procedure VISIT(BB)
2:    $\mathcal{N}_{BB} = \bigsqcup \{ \mathcal{D}_{XX} \oplus \mathcal{C}_{XX \rightarrow BB} \mid XX \in \text{prev}(BB) \wedge \mathcal{D}_{XX} \neq \perp \}$ 
3:   for each instruction  $\in$  instructions(BB):
4:     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

```
1: procedure VISIT(Terminator)
2:   if  $\mathcal{N}_{BB} \sqsubseteq \mathcal{D}_{BB}$  then:                                ▷ red: delegated to abstract domain
3:     return                                                    ▷ state has not changed
4:    $\mathcal{D}_{BB} \leftarrow \mathcal{N}_{BB}$ 
5:   for each  $XX \in \text{next}(BB)$ :
6:     if  $\text{reachable}(BB, XX)$  then:
7:        $\mathcal{W}.\text{push}(XX)$ 
```

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 $\exists(\# \rightarrow \perp) \in \mathcal{C}_{BB \rightarrow XX}$

Visitor: Conditional Branch

Algorithm 4 Visit conditional branch (terminator)

```
1: procedure VISIT(JMP ( $x \square y ? XX : YY$ ))
2:   if isVar( $x$ ) then:
3:      $C_{BB \rightarrow XX} \leftarrow C_{BB \rightarrow XX} \oplus \{x \rightarrow \mathcal{N}_{BB}[x] \square^{\#} \mathcal{N}_{BB}[y]\}$ 
4:      $C_{BB \rightarrow YY} \leftarrow C_{BB \rightarrow YY} \oplus \{x \rightarrow \mathcal{N}_{BB}[x] !\square^{\#} \mathcal{N}_{BB}[y]\}$ 
5:   if isVar( $y$ ) then:
6:     ...
7:   VISITTERMINATOR()
```

considered comparisons:

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

Visitor: Switch

Algorithm 5 Visit switch (terminator)

```
1: procedure VISIT(SWITCH [ $x = a : XX$ ][ $x = b : YY$ ][ $x = c : YY$ ][default :  $ZZ$ ])
2:    $\mathcal{C}_{BB \rightarrow XX} \leftarrow \{x \rightarrow \mathcal{N}_{BB}[x] =^{\#} a\}$ 
3:    $\mathcal{C}_{BB \rightarrow YY} \leftarrow \{x \rightarrow (\mathcal{N}_{BB}[x] =^{\#} b) \sqcup (\mathcal{N}_{BB}[x] =^{\#} c)\}$ 
4:    $\mathcal{C}_{BB \rightarrow ZZ} \leftarrow \{x \rightarrow \mathcal{N}_{BB}[x] \setminus \{a, b, c\}\}$ 
5:   VISIT_TERMINATOR()
```

Visitor: Parallel Assignments at PHI Node

Algorithm 6 PHI node in basic block BB

- 1: **procedure** PHI($x \leftarrow [YY : y][ZZ : z]$)
 - 2: $\mathcal{N}_{BB} \leftarrow \mathcal{N}_{BB} \oplus \{x \rightarrow (\mathcal{D}_{YY} \oplus C_{YY \rightarrow BB})[y] \sqcup (\mathcal{D}_{ZZ} \oplus C_{ZZ \rightarrow BB})[z]\}$
-

Visitor: Binary Expressions

Algorithm 7 Addition in basic block BB

- 1: **procedure** BINARY($x \leftarrow y \square z$)
 - 2: $\mathcal{N}_{BB} \leftarrow \mathcal{N}_{BB} \oplus \{x \rightarrow \mathcal{N}_{BB}[y] \square^{\#} \mathcal{N}_{BB}[z]\}$
-

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** VISIT($x \leftarrow \text{LOAD}(\dots)$)
 - 2: $\mathcal{N}_{BB} \leftarrow \mathcal{N}_{BB} \oplus \{x \rightarrow \top\}$
-

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 Münch, Alexander Roschlaub,
Michael Schwarz

IN2053 - Program Optimization Lab 2018

June 22, 2018