RWTH Aachen

Department of Computer Science
Technical Report

Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution

Jera Hensel, Jürgen Giesl, Florian Frohn, Thomas Ströder

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2016-4

RWTH Aachen · Department of Computer Science · April 2016

Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution*

Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. In earlier work, we developed an approach for automated termination analysis of C programs with explicit pointer arithmetic, which is based on symbolic execution. However, similar to many other termination techniques, this approach assumed the program variables to range over mathematical integers instead of bitvectors. This eases mathematical reasoning but is unsound in general. In this paper, we extend our approach in order to handle fixed-width bitvector integers. Thus, we present the first technique for termination analysis of C programs that covers both byte-accurate pointer arithmetic and bit-precise modeling of integers. We implemented our approach in the automated termination prover AProVE and evaluate its power by extensive experiments.

1 Introduction

In [14], we developed an approach for termination analysis of C with explicit pointer arithmetic, which we implemented in our tool AProVE [9]. AProVE won the termination category of the *International Competition on Software Verification (SV-COMP)*[1] at *TACAS* in 2015 and 2016. However, like the other termination tools at *SV-COMP*, our approach was restricted to mathematical integers.

In general, this is unsound: The function `f` below does not terminate if `x` has the maximum value of its type.[2] But we can falsely prove termination if we treat `x` and `j` as mathematical integers. For `g`, we could falsely conclude non-termination, although `g` always terminates due to the wrap-around for unsigned overflows.

```
void f(unsigned int x)  {
    unsigned int j = 0;
    while (j <= x) j++; }
```

```
void g(unsigned int j) {
    while (j > 0) j++; }
```

In this paper, we adapt our approach for termination of C from [14] to handle the bitvector semantics correctly. To avoid dealing with the intricacies of C, we analyze programs in the platform-independent intermediate representation of the LLVM compilation framework [12]. Our approach works in two steps: First, a *symbolic execution graph* is automatically constructed that represents an over-approximation of all possible program runs (Sect. 2 and 3). This graph can also be used to prove that the program does not result in undefined behavior (so in

* Supported by the DFG grant GI 274/6-1.

[1] See `http://sv-comp.sosy-lab.org/`.

[2] In C, adding 1 to the maximal unsigned integer results in 0. In contrast, for signed integers, adding 1 to the maximal signed integer results in undefined behavior. However, most C implementations return the minimal signed integer as the result.

particular, it is memory safe). In a second step (Sect. 4), this graph is transformed into an *integer transition system (ITS)*, whose termination can be proved by existing techniques. In Sect. 5, we compare our approach with related work and evaluate our corresponding implementation in AProVE. The appendix discusses details on the semantics of abstract states, gives proofs of the theorems, and summarizes the symbolic execution rules for LLVM bitvector instructions.

To extend our approach to fixed-width integers, we express relations between bitvectors by corresponding relations between mathematical integers $\mathbb{Z}$. In this way, we can use standard SMT solving over $\mathbb{Z}$ for all steps needed to construct the symbolic execution graph. Moreover, this allows us to obtain ITSs over $\mathbb{Z}$ from these graphs, and to use standard approaches for generating ranking functions to prove termination of these ITSs. So our contribution is a general technique to adapt byte-accurate symbolic execution to the handling of bitvectors, which can also be used for many other program analyses besides proving termination.

Limitations To simplify the presentation and to concentrate on the issues related to bitvectors, we restrict ourselves to a single LLVM function and to LLVM *types* of the form `i`n (for n-bit integers), `i`n`*` (for pointers to values of type `i`n), `i`n`**`, `i`n`***`, etc. Moreover, we assume a 1 byte data alignment (i.e., values may be stored at any address) and only handle memory allocation by the LLVM instruction `alloca`. See [14] for an extension of our approach to programs with several LLVM functions, arbitrary alignment, and external functions like `malloc`. As discussed in [14], some LLVM concepts are not yet supported by our approach (e.g., `undef`, floating point values, vectors, `struct` types, and recursion). Another limitation is that our approach cannot directly *disprove* properties like memory safety or termination, as it is based on over-approximating all possible program runs.

2 LLVM States for Symbolic Execution

In this section, we define concrete and abstract LLVM states that represent sets of concrete states. These states will be needed for symbolic execution in Sect. 3. As an example, consider the function `g` from Sect. 1. In the corresponding[3] LLVM code in Fig. 1, the integer variable `j` has the type `i32`, as it is represented as a bitvector of length 32. The program is split into the *basic blocks* `entry`, `cmp`, `body`, and `done`. We will explain this

```
define i32 @g(i32 j) {
entry: 0: ad = alloca i32
       1: store i32 j, i32* ad
       2: br label cmp
cmp:   0: j1 = load i32* ad
       1: j1pos = icmp ugt i32 j1, 0
       2: br i1 j1pos, label body, label done
body:  0: j2 = load i32* ad
       1: inc = add i32 j2, 1
       2: store i32 inc, i32* ad
       3: br label cmp
done:  0: ret void }
```

Fig. 1: LLVM code for the function `g`

[3] This LLVM program corresponds to the code obtained from `g` with the Clang compiler [3]. To ease readability, we wrote variables without "`%`" in front (i.e., we wrote "`j`" instead of "`%j`" as in proper LLVM) and added line numbers.

LLVM code in detail when constructing the symbolic execution graph in Sect. 3.

In our abstract domain, an LLVM state consists of the current program position, the values of the local program variables, a knowledge base with information about these values, and two sets to describe allocations and the contents of memory. The *program position* is represented by a pair $(\mathtt{b}, k)$. Here, b is the name of the current basic block and k is the index of the next instruction. So if *Blks* is the set of all basic blocks, then the set of program positions is $Pos = Blks \times \mathbb{N}$. We represent an assignment to the *local program variables* $\mathcal{V}_\mathcal{P}$ (e.g., $\mathcal{V}_\mathcal{P} = \{\mathtt{j}, \mathtt{ad}, \ldots\}$) by an injective function $LV : \mathcal{V}_\mathcal{P} \to \mathcal{V}_{sym}$, where $\mathcal{V}_{sym}$ is an infinite set of symbolic variables with $\mathcal{V}_{sym} \cap \mathcal{V}_\mathcal{P} = \varnothing$. Let $\mathcal{V}_{sym}(LV) \subseteq \mathcal{V}_{sym}$ be the set of all symbolic variables v where $LV(\mathtt{x}) = v$ for some $\mathtt{x} \in \mathcal{V}_\mathcal{P}$.

The third component of states is the *knowledge base* $KB \subseteq QF\_IA(\mathcal{V}_{sym})$, a set of first-order quantifier-free integer arithmetic formulas. For concrete states, KB uniquely determines the values of symbolic variables, whereas for abstract states several values are possible. We identify *sets* of formulas $\{\varphi_1, \ldots, \varphi_n\}$ with their *conjunction* $\varphi_1 \wedge \ldots \wedge \varphi_n$ and require that KB is just a conjunction of equalities and inequalities in order to speed up SMT-based arithmetic reasoning.

The fourth component of a state is an *allocation list* AL. It contains expressions of the form $[\![v_1,\, v_2]\!]$ for $v_1, v_2 \in \mathcal{V}_{sym}$, which indicate that $v_1 \leq v_2$ and that all addresses between v_1 and v_2 have been allocated by an alloca instruction.

The fifth component PT is a set of "points-to" atoms $v_1 \hookrightarrow_{\mathtt{ty},i} v_2$ where $v_1, v_2 \in \mathcal{V}_{sym}$, ty is an LLVM type, and $i \in \{u, s\}$. This means that the value v_2 of type ty is stored at the address v_1, where $i \in \{u, s\}$ indicates whether v_2 represents this value as an unsigned or signed integer. As each memory cell stores one byte, $v_1 \hookrightarrow_{\mathtt{i32},i} v_2$ states that v_2 is stored in the four cells $v_1, \ldots, v_1 + 3$.

Finally, we use a special state ERR to be reached if we cannot prove absence of undefined behavior (e.g., if a non-allowed overflow or a violation of memory safety by accessing non-allocated memory might take place).

Definition 1 (States). LLVM states *have the form* (p, LV, KB, AL, PT) *where* $p \in Pos$, $LV : \mathcal{V}_\mathcal{P} \to \mathcal{V}_{sym}$, $KB \subseteq QF\_IA(\mathcal{V}_{sym})$, $AL \subseteq \{[\![v_1,\, v_2]\!] \mid v_1, v_2 \in \mathcal{V}_{sym}\}$, *and* $PT \subseteq \{(v_1 \hookrightarrow_{\mathtt{ty},i} v_2) \mid v_1, v_2 \in \mathcal{V}_{sym}, \mathtt{ty}$ *is an* LLVM *type*, $i \in \{u, s\}\}$. *In addition, there is a state ERR for undefined behavior. For* $a = (p, LV, KB, AL, PT)$, *let* $\mathcal{V}_{sym}(a)$ *consist of* $\mathcal{V}_{sym}(LV)$ *and all symbolic variables in KB, AL, or PT.*

We often identify the mapping LV with the equations $\{\mathtt{x} = LV(\mathtt{x}) \mid \mathtt{x} \in \mathcal{V}_\mathcal{P}\}$. As an example, consider the following abstract state for our function g:

$$(\,(\mathtt{entry}, 2),\, \{\mathtt{j} = v_\mathtt{j}, \mathtt{ad} = v_\mathtt{ad}\},\, \{v_{end} = v_\mathtt{ad} + 3\},\, \{[\![v_\mathtt{ad},\, v_{end}]\!]\},\, \{v_\mathtt{ad} \hookrightarrow_{\mathtt{i32},u} v_\mathtt{j}\}\,) \quad (1)$$

It represents states in the entry block immediately before executing the instruction in line 2. Here, $LV(\mathtt{j}) = v_\mathtt{j}$, the memory cells between $LV(\mathtt{ad}) = v_\mathtt{ad}$ and $v_{end} = v_\mathtt{ad} + 3$ have been allocated, and $v_\mathtt{j}$ is stored in the 4 cells $v_\mathtt{ad}, \ldots, v_{end}$.

In contrast to [14], we partition the program variables $\mathcal{V}_\mathcal{P}$ into two disjoint sets $\mathcal{U}_\mathcal{P}$ and $\mathcal{S}_\mathcal{P}$. If $\mathtt{x} \in \mathcal{U}_\mathcal{P}$ (resp. $\mathtt{x} \in \mathcal{S}_\mathcal{P}$), then $LV(\mathtt{x})$ is x's value as an unsigned (resp. signed) integer. This is advantageous when formulating rules to execute LLVM instructions like icmp ugt and sgt, since the LLVM types do not distinguish between unsigned and signed integers. Instead, some LLVM instructions consider their arguments as "unsigned" whereas others consider them as "signed".

To determine $\mathcal{U}_\mathcal{P}$ and $\mathcal{S}_\mathcal{P}$, we use the following heuristic which statically scans the program $\mathcal{P}$ for variables which are (mainly) used in unsigned resp. signed interpretation. We iteratively add a variable `x` to $\mathcal{U}_\mathcal{P}$ if

- `x` is an address (i.e., it has a type of the form `ty*`),
- `x` occurs in an unsigned comparison instruction (e.g., `icmp ugt` for the integer comparison "unsigned greater than") or in another unsigned operation (e.g., `udiv` or `urem` for "unsigned division" or "remainder"),
- `x` occurs in a sign neutral comparison (`icmp eq` or `ne`) or in a `phi` or `select` instruction together with another variable $\mathtt{y} \in \mathcal{U}_\mathcal{P}$, where `y` is not the condition,
- `x` occurs in an `add`, `sub`, `mul`, or `shl` instruction without `nsw` flag ("no signed wrap-up" means that overflow of signed integers yields undefined behavior),
- `x` occurs in a binary or conversion instruction with another $\mathtt{y} \in \mathcal{U}_\mathcal{P}$,
- `x` is the result of `icmp` or the condition of a branch (`br`) or `select` instruction,
- `x` occurs in a `lshr` ("logical shift right") instruction,
- `x` occurs in a `zext` instruction (the "zero extension" adds zero bits in front),
- `x` is loaded from an address where a variable $\mathtt{y} \in \mathcal{U}_\mathcal{P}$ is stored to, or
- `x` is stored to an address where a variable $\mathtt{y} \in \mathcal{U}_\mathcal{P}$ is loaded from.

Afterwards, we iteratively remove `x` from $\mathcal{U}_\mathcal{P}$ again if

- `x` is one of the two arguments of a signed comparison (e.g., `icmp sgt`) or `x` occurs in another signed operation (e.g., `sdiv` or `srem`),
- `x` occurs in a comparison or in a `phi` or `select` instruction together with another variable $\mathtt{y} \in \mathcal{V}_\mathcal{P} \setminus \mathcal{U}_\mathcal{P}$, where `x` is not the condition,
- `x` occurs in an instruction flagged by `nsw`,
- `x` occurs in a binary or conversion instruction with another $\mathtt{y} \in \mathcal{V}_\mathcal{P} \setminus \mathcal{U}_\mathcal{P}$,
- `x` occurs in an `ashr` ("arithmetic shift right") instruction,
- `x` occurs in a `sext` instruction (the "sign extension" adds copies of the most significant bit in front),
- `x` is loaded from an address where a variable $\mathtt{y} \in \mathcal{V}_\mathcal{P} \setminus \mathcal{U}_\mathcal{P}$ is stored to, or
- `x` is stored to an address where a variable $\mathtt{y} \in \mathcal{V}_\mathcal{P} \setminus \mathcal{U}_\mathcal{P}$ is loaded from.

We then define $\mathcal{S}_\mathcal{P} = \mathcal{V}_\mathcal{P} \setminus \mathcal{U}_\mathcal{P}$. In this way, we make sure that in each instruction in $\mathcal{P}$, all occurring program variables of type $\mathtt{i}n$ with $n > 1$ are either from $\mathcal{U}_\mathcal{P}$ or from $\mathcal{S}_\mathcal{P}$. In our example, we obtain $\mathcal{U}_\mathcal{P} = \mathcal{V}_\mathcal{P} = \{\mathtt{j}, \mathtt{ad}, \ldots, \mathtt{inc}\}$ and $\mathcal{S}_\mathcal{P} = \varnothing$. Note that there is no guarantee that all variables in $\mathcal{U}_\mathcal{P}$ resp. $\mathcal{S}_\mathcal{P}$ are used as unsigned resp. signed integers in the original C program (e.g., if $\mathtt{y}, \mathtt{z} \in \mathcal{S}_\mathcal{P}$ and the C program contains "`unsigned int x = y + z;`", then our heuristic would conclude $\mathtt{x} \in \mathcal{S}_\mathcal{P}$, since the resulting LLVM code has the instruction "`x = add i32 y, z`"). Our analysis remains correct if there are (un)signed variables that we do not recognize as being (un)signed (i.e., failure of the above heuristic for $\mathcal{U}_\mathcal{P}$ and $\mathcal{S}_\mathcal{P}$ only affects the performance, but not the soundness of our approach).

To construct symbolic execution graphs, for any state a we use a first-order formula $\langle a \rangle_{FO}$, which is a conjunction of equalities and inequalities containing KB and obvious consequences of AL and PT. Moreover, $\langle a \rangle_{FO}$ states that all integers belong to intervals corresponding to their types. Here, let $\mathsf{umax}_n = 2^n - 1$, $\mathsf{smin}_n = -2^{n-1}$, and $\mathsf{smax}_n = 2^{n-1} - 1$. Moreover, $size(\mathtt{ty})$ is the number of bits required for values of type `ty` (e.g., $size(\mathtt{i}n) = n$ and $size(\mathtt{ty*}) = 32$ (resp.

64) on 32-bit (resp. 64-bit) architectures). As usual, "$v \in [k, m]$" is a shorthand for "$k \leq v \wedge v \leq m$" and "$\models \varphi$" means that φ is a tautology.

Definition 2 (*FO* Formulas for States). *$\langle a \rangle_{FO}$ is the smallest set with*[4]

$$\begin{aligned}\langle a \rangle_{FO} = KB \cup \{0 < v_1 \leq v_2 \mid [\![v_1,\, v_2]\!] \in AL\} \cup&\\ \{v_2 = w_2 \mid (v_1 \hookrightarrow_{\mathtt{ty},i} v_2), (w_1 \hookrightarrow_{\mathtt{ty},i} w_2) \in PT \text{ and } \models \langle a \rangle_{FO} \Rightarrow v_1 = w_1\} \cup&\\ \{v_1 \neq w_1 \mid (v_1 \hookrightarrow_{\mathtt{ty},i} v_2), (w_1 \hookrightarrow_{\mathtt{ty},i} w_2) \in PT \text{ and } \models \langle a \rangle_{FO} \Rightarrow v_2 \neq w_2\} \cup&\\ \{0 < v_1 \wedge v_2 \in [0, \mathsf{umax}_{size(\mathtt{ty})}] \mid (v_1 \hookrightarrow_{\mathtt{ty},u} v_2) \in PT\} \cup&\\ \{0 < v_1 \wedge v_2 \in [\mathsf{smin}_{size(\mathtt{ty})}, \mathsf{smax}_{size(\mathtt{ty})}] \mid (v_1 \hookrightarrow_{\mathtt{ty},s} v_2) \in PT\} \cup&\\ \{LV(\mathtt{x}) \in [0, \mathsf{umax}_{size(\mathtt{ty})}] \mid \mathtt{x} \in \mathcal{U}_{\mathcal{P}}, \mathtt{x} \text{ has type } \mathtt{ty}\} \cup&\\ \{LV(\mathtt{x}) \in [\mathsf{smin}_{size(\mathtt{ty})}, \mathsf{smax}_{size(\mathtt{ty})}] \mid \mathtt{x} \in \mathcal{S}_{\mathcal{P}}, \mathtt{x} \text{ has type } \mathtt{ty}\}.&\end{aligned}$$

Concrete states determine the values of variables and the contents of the memory *uniquely*. To enforce a uniform representation, in concrete states we only allow statements of the form $(w_1 \hookrightarrow_{\mathtt{ty},i} w_2)$ in PT where $\mathtt{ty} = \mathtt{i8}$ and $i = u$. In addition, concrete states (p, LV, KB, AL, PT) must be *well formed*, i.e., for every $(w_1 \hookrightarrow_{\mathtt{ty},i} w_2) \in PT$, there is an $[\![v_1,\, v_2]\!] \in AL$ such that $\models KB \Rightarrow v_1 \leq w_1 \leq v_2$. So PT only contains information about addresses that are known to be allocated.

Definition 3 (Concrete States). *An* LLVM *state c is* concrete *iff $c = ERR$ or $c = (p, LV, KB, AL, PT)$ is well formed, $\langle c \rangle_{FO}$ is satisfiable, and*

- *For all $v \in \mathcal{V}_{sym}(c)$ there exists an $n \in \mathbb{Z}$ such that $\models \langle c \rangle_{FO} \Rightarrow v = n$.*
- *For all $[\![v_1,\, v_2]\!] \in AL$ and for all integers n with $\models \langle c \rangle_{FO} \Rightarrow v_1 \leq n \leq v_2$, there exists $(w_1 \hookrightarrow_{\mathtt{i8},u} w_2) \in PT$ for some $w_1, w_2 \in \mathcal{V}_{sym}$ such that $\models \langle c \rangle_{FO} \Rightarrow w_1 = n$ and $\models \langle c \rangle_{FO} \Rightarrow w_2 = k$, for some $k \in [0, \mathsf{umax}_8]$.*
- *There is no $(w_1 \hookrightarrow_{\mathtt{ty},i} w_2) \in PT$ for $\mathtt{ty} \neq \mathtt{i8}$ or $i = s$.*

In [14], for every abstract state a, we also introduced a *separation logic* formula $\langle a \rangle_{SL}$ which extends $\langle a \rangle_{FO}$ by detailed information about the memory. (We recapitulate $\langle a \rangle_{SL}$ and the semantics of separation logic in App. A.) For this semantics, we use *interpretations* (as, mem). Here, $as : \mathcal{V}_{\mathcal{P}} \to \mathbb{Z}$ is an *assignment* of the program variables, where for $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ of type $\mathtt{ty}$, we have $as(\mathtt{x}) \in [0, \mathsf{umax}_{size(\mathtt{ty})}]$ if $\mathtt{x} \in \mathcal{U}_{\mathcal{P}}$ and $as(\mathtt{x}) \in [\mathsf{smin}_{size(\mathtt{ty})}, \mathsf{smax}_{size(\mathtt{ty})}]$ if $\mathtt{x} \in \mathcal{S}_{\mathcal{P}}$. The partial function $mem : \mathbb{N}_{>0} \rightharpoonup \{0, \ldots, \mathsf{umax}_8\}$ with finite domain describes the memory contents at allocated addresses (as unsigned integers). We use "$\rightharpoonup$" for partial functions. For any abstract state a, we have $\models \langle a \rangle_{SL} \Rightarrow \langle a \rangle_{FO}$. So $\langle a \rangle_{FO}$ is a weakened version of $\langle a \rangle_{SL}$, used to construct symbolic execution graphs. This allows us to use standard first-order SMT solving for all reasoning in our approach.

Now we define which concrete states are represented by an abstract state a. We extract an interpretation (as^c, mem^c) from every concrete state $c \neq ERR$. Then a *represents* all concrete states c where (as^c, mem^c) is a model of some concrete instantiation of $\langle a \rangle_{SL}$. A *concrete instantiation* is a function $\sigma : \mathcal{V}_{sym} \to \mathbb{Z}$. So σ does not instantiate $\mathcal{V}_{\mathcal{P}}$. Instantiations are extended to formulas as usual.

[4] Of course, $\langle a \rangle_{FO}$ can be extended by more formulas, e.g., on the connection between v_2 and v_2' if $(v_1 \hookrightarrow_{\mathtt{i}n,u} v_2), (v_1 \hookrightarrow_{\mathtt{i}m,u} v_2') \in PT$ for $n < m$. Then we can also handle programs which load an $\mathtt{i}n$ integer from an address where an $\mathtt{i}m$ integer was stored.

Definition 4 (Representing Concrete by Abstract States). *Let $c = (p, LV^c, KB^c, AL^c, PT^c)$ be a concrete state. For every $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, let $as^c(\mathtt{x}) = n$ for the number $n \in \mathbb{Z}$ with $\models \langle c \rangle_{FO} \Rightarrow LV^c(\mathtt{x}) = n$. For $n \in \mathbb{N}_{>0}$, the function $mem^c(n)$ is defined iff there exists a $(w_1 \hookrightarrow_{\mathtt{i8},u} w_2) \in PT$ such that $\models \langle c \rangle_{FO} \Rightarrow w_1 = n$. Let $\models \langle c \rangle_{FO} \Rightarrow w_2 = k$, where $k \in [0, \mathsf{umax}_8]$. Then $mem^c(n) = k$.*

We say that an abstract state $a = (p, LV^a, KB^a, AL^a, PT^a)$ represents *a concrete state $c = (p, LV^c, KB^c, AL^c, PT^c)$ iff a is well formed and (as^c, mem^c) is a* model *of $\sigma(\langle a \rangle_{SL})$ for some concrete instantiation σ of the symbolic variables. The only state that represents the error state ERR is ERR itself.*

So the abstract state (1) represents all concrete states $c = ((\mathtt{entry}, 2), LV, KB, AL, PT)$ where mem^c stores the 32-bit integer $as^c(\mathtt{j})$ at the address $as^c(\mathtt{ad})$.

3 From LLVM to Symbolic Execution Graphs

We now show how to automatically generate a *symbolic execution graph* that over-approximates all executions of a program. To this end, we define operations to convert any integer expression t into an unsigned resp. signed n-bit integer:[5]

$$\mathsf{uns}_n(t) = t \bmod 2^n \qquad\qquad \mathsf{sig}_n(t) = ((t + 2^{n-1}) \bmod 2^n) - 2^{n-1}$$

The correctness of uns_n is obvious and by Thm. 5, sig_n is correct as well.

Theorem 5 (Converting Integers to Signed n-Bit Integers). *Let $n \in \mathbb{N}$ with $n \geq 1$. Then $\mathsf{sig}_n(t) \in [\mathsf{smin}_n, \mathsf{smax}_n]$ and $t \bmod 2^n = \mathsf{sig}_n(t) \bmod 2^n$.*

Moreover, we extend LV to apply it also to concrete integers. To this end, we use $LV_{u,n}, LV_{s,n} : \mathcal{V}_\mathcal{P} \uplus \mathbb{Z} \to \mathcal{V}_{sym} \uplus \mathbb{Z}$, where $LV_{u,n}(t)$ (resp. $LV_{s,n}(t)$) is t represented as an unsigned (resp. signed) integer with n bits, for any $t \in \mathcal{V}_\mathcal{P} \uplus \mathbb{Z}$:

$$LV_{u,n}(t) = \begin{cases} LV(t), & \text{if } t \in \mathcal{U}_\mathcal{P} \\ \mathsf{uns}_n(LV(t)), & \text{if } t \in \mathcal{S}_\mathcal{P} \\ \mathsf{uns}_n(t), & \text{if } t \in \mathbb{Z} \end{cases} \qquad LV_{s,n}(t) = \begin{cases} \mathsf{sig}_n(LV(t)), & \text{if } t \in \mathcal{U}_\mathcal{P} \\ LV(t), & \text{if } t \in \mathcal{S}_\mathcal{P} \\ \mathsf{sig}_n(t), & \text{if } t \in \mathbb{Z} \end{cases}$$

We developed symbolic execution rules for all LLVM instructions that are affected by the adaption to bitvectors. We handle overflows by appropriate case analyses (Sect. 3.1) or by introducing "modulo" relations (Sect. 3.2). Moreover, Sect. 3.3 presents rules for bitwise binary and conversion instructions. The remaining bitvector instructions of LLVM are handled in an analogous way (see App. C for details), and rules for other LLVM instructions can be found in [14].

3.1 Handling Bitvector Operations by Case Analysis

We start with the initial states that one wants to analyze for termination, e.g., with the abstract state A where $\mathtt{j}$ has an unknown value. In the symbolic execution graph for g in Fig. 2, we abbreviated parts by "..." and wrote $\hookrightarrow_{\mathtt{i32}}$ and umax instead of $\hookrightarrow_{\mathtt{i32},u}$ and umax_{32}. To ease readability, we replaced some sym-

[5] As usual, mod is defined as follows: For any $m \in \mathbb{Z}$ and $n \in \mathbb{N}_{>0}$, we have $t = m \bmod n$ iff $t \in [0, n-1]$ and there exists a $k \in \mathbb{Z}$ such that $t = k \cdot n + m$.

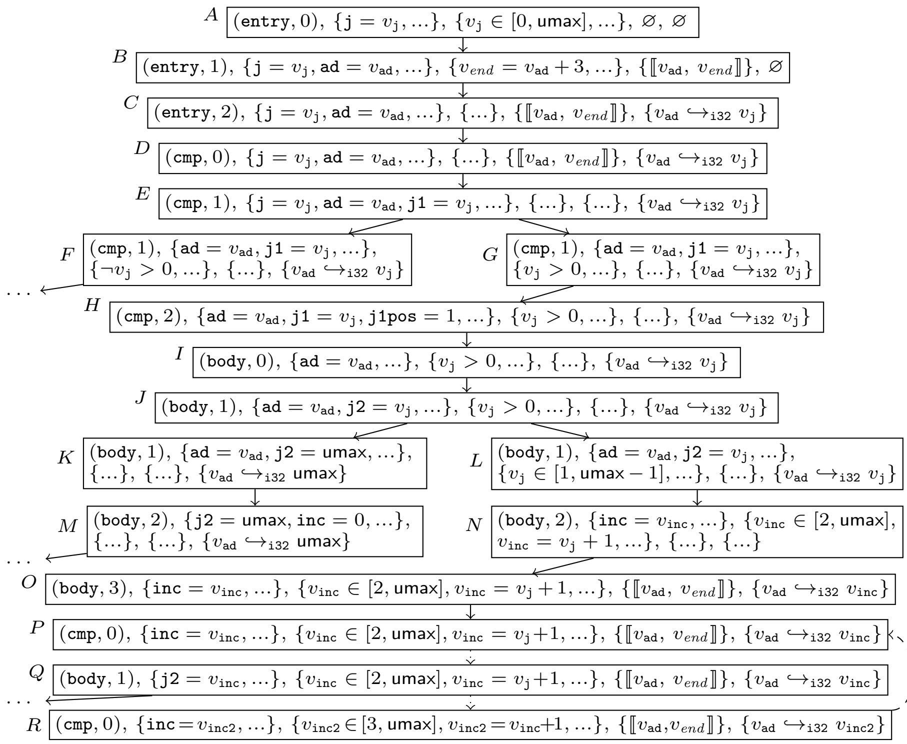

Fig. 2: Symbolic execution graph for the function g

bolic variables by their values (e.g., we wrote $\texttt{j1pos} = 1$) and explicitly depicted formulas like $v_{\texttt{j}} \in [0, \mathsf{umax}]$ that follow from $\langle A \rangle_{FO}$ since $\texttt{j} \in \mathcal{U}_{\mathcal{P}}$ and $LV(\texttt{j}) = v_{\texttt{j}}$.

The function g allocates $[\![v_{\texttt{ad}},\, v_{end}]\!]$ and stores the value $v_{\texttt{j}}$ of j at address ad. Next, we jump to the block cmp for the loop comparison. After loading the value $v_{\texttt{j}}$ (stored at address ad) to the program variable j1, in State E we check whether j1's value in unsigned interpretation is greater than 0 (icmp ugt).

The following rule evaluates such instructions symbolically. In our rules, "p: *ins*" states that *ins* is the instruction at position p. Let a always denote the abstract state *before* the execution step (i.e., above the horizontal line of the rule), where we write $\langle a \rangle$ instead of $\langle a \rangle_{FO}$. Moreover, $LV[\texttt{x} := v]$ is the function where $(LV[\texttt{x} := v])(\texttt{x}) = v$ and $(LV[\texttt{x} := v])(\texttt{y}) = LV(\texttt{y})$ for $\texttt{y} \neq \texttt{x}$. If $p = (\texttt{b}, k)$, then $p^+ = (\texttt{b}, k+1)$ is the position of the next instruction in the same block.

icmp ugt (p: **"x = icmp ugt ty** t_1, t_2**" with** $\texttt{x} \in \mathcal{V}_{\mathcal{P}}$**,** $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\texttt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and if}$$

either $\models \langle a \rangle \Rightarrow (LV_{u,size(\texttt{ty})}(t_1) > LV_{u,size(\texttt{ty})}(t_2))$ and φ is "$v = 1$"
or $\models \langle a \rangle \Rightarrow (LV_{u,size(\texttt{ty})}(t_1) \leq LV_{u,size(\texttt{ty})}(t_2))$ and φ is "$v = 0$"

However, in our example the value of $LV_{u,32}(\texttt{j1}) = LV(\texttt{j1}) = v_{\texttt{j}}$ is unknown. Therefore, we first have to *refine* State E to States F and G such that the comparison can be decided. For this case analysis, we use the following rule.

`icmp ugt` refinement (p: "`x = icmp ugt ty` t_1, t_2" with $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)} \quad \text{if}$$

φ is "$LV_{u,size(\mathtt{ty})}(t_1) > LV_{u,size(\mathtt{ty})}(t_2)$" and we have both $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$

The rules for other comparisons are analogous. So the rules for the *signed* "greater than" comparison (`sgt`) are obtained by replacing $LV_{u,size(\mathtt{ty})}$ with $LV_{s,size(\mathtt{ty})}$.

If `y` is compared by `ugt` and $\mathtt{y} \in \mathcal{U}_{\mathcal{P}}$, then $LV(\mathtt{y})$ is `y`'s value as an unsigned integer, which makes the comparison very simple. (Similarly, $LV(\mathtt{y})$ is signed if `y` is compared by `sgt`). In contrast, if LV represented the value of *all* program variables as signed integers, then for `icmp ugt` we would have to consider more cases, which results in a significantly larger graph (i.e., in a less efficient approach).[6]

In our example, if $\neg v_{\mathtt{j}} > 0$ (State F), we `return` from the function. If $v_{\mathtt{j}} > 0$ (State G), the conditional `branch` instruction leads us to the block that corresponds to the `body` of the `while`-loop. In the step from State I to J, again the value $v_{\mathtt{j}}$ stored at address $v_{\mathtt{ad}}$ is loaded to a program variable `j2`. The next instruction is an overflow-sensitive addition: If $v_{\mathtt{j}} < \mathsf{umax}_{32}$, then $v_{\mathtt{j}} + 1$ is assigned to `inc`. But if $v_{\mathtt{j}} = \mathsf{umax}_{32}$, then there is an overflow. If KB does not contain enough information to decide whether an overflow occurs, we perform a case analysis.

unsigned `add` refinement (p: "`x = add i`n t_1, t_2" with $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)} \quad \text{if } \mathtt{x} \in \mathcal{U}_{\mathcal{P}} \text{ and}$$

φ is "$LV_{u,n}(t_1) + LV_{u,n}(t_2) \leq \mathsf{umax}_n$", where $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$

Therefore, State J is refined to K and L. In K, `j2` has the value umax_{32}, i.e., adding 1 results in an overflow. In State L, this overflow cannot happen.

The rule for "signed `add` refinement" is analogous, but here we have $\mathtt{x} \in \mathcal{S}_{\mathcal{P}}$ and we obtain three instead of two cases: "$LV_{s,n}(t_1) + LV_{s,n}(t_2) < \mathsf{smin}_n$", "$LV_{s,n}(t_1) + LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n]$", and "$LV_{s,n}(t_1) + LV_{s,n}(t_2) > \mathsf{smax}_n$".

Now we define rules for `add`. If no overflow can occur, then the result is the addition of the operators. Thus, State L evaluates to N, where $v_{\mathtt{inc}} = v_{\mathtt{j}} + 1$.

`add` without overflow (p: "`x = add [nsw] i`n t_1, t_2" with $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and if}$$

either $\mathtt{x} \in \mathcal{U}_{\mathcal{P}}$, $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) + LV_{u,n}(t_2) \in [0, \mathsf{umax}_n])$, and φ is "$v = LV_{u,n}(t_1) + LV_{u,n}(t_2)$"

or $\mathtt{x} \in \mathcal{S}_{\mathcal{P}}$, $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) + LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n])$, and φ is "$v = LV_{s,n}(t_1) + LV_{s,n}(t_2)$"

---

[6] Then we would have to check first whether $LV_{s,size(\mathtt{ty})}(t_1) < 0$ and $LV_{s,size(\mathtt{ty})}(t_2) \geq 0$. In that case, "`icmp ugt ty` t_1, t_2" is *true*, since the most significant bits of t_1 and t_2 are 1 and 0, respectively. The other cases are $LV_{s,size(\mathtt{ty})}(t_1) \geq 0 \wedge LV_{s,size(\mathtt{ty})}(t_2) < 0$, and the two cases where $LV_{s,size(\mathtt{ty})}(t_1)$ and $LV_{s,size(\mathtt{ty})}(t_2)$ have the same sign and either $LV_{s,size(\mathtt{ty})}(t_1) > LV_{s,size(\mathtt{ty})}(t_2)$ or $LV_{s,size(\mathtt{ty})}(t_1) \leq LV_{s,size(\mathtt{ty})}(t_2)$.

If an overflow occurs, then due to the wrap-around, the unsigned result value is the sum of the operands minus the type size 2^n. For example, in the evaluation of State K to M, we add the relation $v_{\mathtt{inc}} = \mathsf{umax}_{32} + 1 - 2^{32} = 0$.

unsigned `add` with overflow (p: "`x = add i`n t_1, t_2" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{v = LV_{u,n}(t_1) + LV_{u,n}(t_2) - 2^n\},\ AL,\ PT)} \quad \text{if}$$

$\mathtt{x} \in \mathcal{U}_\mathcal{P}$, $v \in \mathcal{V}_{sym}$ is fresh, and $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) + LV_{u,n}(t_2) > \mathsf{umax}_n)$

When adding two signed integers in C, an overflow leads to undefined behavior. Thus, this is translated into an LLVM instruction with the flag `nsw`. However, when adding an unsigned and a signed integer in C, an overflow does not yield undefined behavior (i.e., the resulting LLVM instruction is not flagged with `nsw`). Our heuristic for $\mathcal{U}_\mathcal{P}$ and $\mathcal{S}_\mathcal{P}$ would consider this to be "signed" addition. Thus, we also need a rule for overflow of signed `add` without the flag `nsw`.

Moreover, most C implementations use a wrap-around semantics also for signed integers. Thus, they compile C to LLVM code where `nsw` is not used at all. Our approach is independent of the actual C compiler, as it analyzes termination of the resulting LLVM program instead and it can also handle signed overflows.

Thus, we use a similar rule for $\mathtt{x} \in \mathcal{S}_\mathcal{P}$. If $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) + LV_{s,n}(t_2) > \mathsf{smax}_n)$, then we add "$v = LV_{s,n}(t_1) + LV_{s,n}(t_2) - 2^n$" to the knowledge base KB. If $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) + LV_{s,n}(t_2) < \mathsf{smin}_n)$, we add "$v = LV_{s,n}(t_1) + LV_{s,n}(t_2) + 2^n$". However, a potential signed overflow that is flagged with `nsw` leads to ERR.

signed `add` with `nsw` overflow (p: "`x = add nsw i`n t_1, t_2", $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P} \text{ and } \not\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) + LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n])$$

For M, the execution ends after some more steps. For N, after storing $v_{\mathtt{inc}}$ to $v_{\mathtt{ad}}$, we branch to block `cmp` again. State P is like D (but `ad` points to `j` in D whereas `ad` points to `inc` in P). Therefore, we continue the execution, where the steps from P to Q are similar to the steps from D to J. Here, dotted arrows abbreviate several steps. Q is again refined and in the case where no overflow occurs, we finally reach State R at the same program position as D and P.

To obtain *finite* symbolic execution graphs, we can *generalize* states whenever an evaluation visits a program position $(\mathtt{b}, k)$ multiple times. We say that a' is a *generalization* of a with the instantiation μ whenever the conditions (b) – (e) of the following rule from [14] are satisfied. Again, a is the state *before* the generalization step and a' is the state *resulting* from the generalization.

generalization with μ

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p',\ LV',\ KB',\ AL',\ PT')} \quad \text{if}$$

(a) a has an incoming "evaluation edge" (not just refinement or generalization edges)
(b) $LV(\mathtt{x}) = \mu(LV'(\mathtt{x}))$ for all $\mathtt{x} \in \mathcal{V}_\mathcal{P}$
(c) $\models \langle a \rangle \Rightarrow \mu(KB')$
(d) if $[\![v_1,\ v_2]\!] \in AL'$, then $[\![\mu(v_1),\ \mu(v_2)]\!] \in AL$
(e) for $i \in \{u, s\}$, if $(v_1 \hookrightarrow_{\mathtt{ty},i} v_2) \in PT'$, then $(\mu(v_1) \hookrightarrow_{\mathtt{ty},i} \mu(v_2)) \in PT$

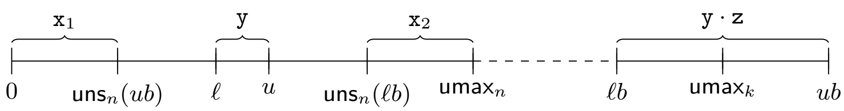

Fig. 3: Multiplication of unsigned integers

Clearly, we have $\models \langle a \rangle_{SL} \Rightarrow \mu(\langle a' \rangle_{SL})$. Condition (a) is needed to avoid cycles of refinement and generalization steps, which do not correspond to any computation. See [14] for a heuristic to compute suitable generalizations automatically.

In our graph, P is a generalization of State R. If we use an instantiation μ with $\mu(v_{\mathtt{j}}) = v_{\mathtt{inc}}$ and $\mu(v_{\mathtt{inc}}) = v_{\mathtt{inc2}}$, then all conditions of the rule are satisfied. So we can conclude the graph construction with a (dashed) *generalization edge* from R to P. A symbolic execution graph is *complete* if all its leaves correspond to `ret` instructions (so in particular, the graph does not contain *ERR* states). As shown in [14], any LLVM evaluation of concrete states can be simulated by our symbolic execution rules. So in particular, a program with a complete symbolic execution graph does not exhibit undefined behavior (thus, it is memory safe).

3.2 Handling Bitvector Operations by Modulo Relations

We now consider further LLVM instructions whose symbolic execution rules have to be adapted to bitvector arithmetic. A refinement with two cases was sufficient to express the result of unsigned addition (or subtraction): if $\mathtt{y}+\mathtt{z}$ exceeds $\mathsf{umax}_n = 2^n - 1$ for unsigned integers $\mathtt{y}$ and $\mathtt{z}$, then the result of the addition is $(\mathtt{y}+\mathtt{z}) - 2^n \in [0, \mathsf{umax}_n]$, since $\mathtt{y}+\mathtt{z}$ can never exceed $2 \cdot \mathsf{umax}_n$. But for multiplication, if $\mathtt{y} \cdot \mathtt{z}$ exceeds umax_n, then $(\mathtt{y} \cdot \mathtt{z}) - 2^n$ is not necessarily in $[0, \mathsf{umax}_n]$. In contrast, one might have to subtract 2^n multiple times. Even worse, if one only knows that $\mathtt{y}$ and $\mathtt{z}$ are values from some interval, then for some values of $\mathtt{y} \cdot \mathtt{z}$ one may have to subtract 2^n more often than for others in order to obtain a result in $[0, \mathsf{umax}_n]$. So for multiplication, performing case analysis to handle overflows is not practical.[7] Thus, we use modulo relations instead, which hold regardless of whether an overflow occurs or not: for unsigned integers, if $\mathtt{x}$ is the result of multiplying $\mathtt{y}$ and $\mathtt{z}$, then the relation "$\mathtt{x} = \mathtt{y} \cdot \mathtt{z} \bmod 2^n$" (i.e., $\mathtt{x} = \mathsf{uns}_n(\mathtt{y} \cdot \mathtt{z})$) correctly models the overflow of bitvectors of size n. To use standard SMT solvers for "modulo", any expression "$t = m \bmod n$" can be transformed into "$t = k \cdot n + m$", where $0 \leq t < m$ and k is an existentially quantified fresh variable.

In some cases, the result of a multiplication "`x = mul i`n t_1`,` t_2" can be in disjoint intervals. For example, if $\mathtt{y} \in [\ell, u]$ such that $\ell \cdot \mathtt{z} \leq \mathsf{umax}_k < u \cdot \mathtt{z}$ for some k, then there can be two intervals ($\mathtt{x}_1$, $\mathtt{x}_2$ in Fig. 3) for $\mathtt{x} = \mathtt{y} \cdot \mathtt{z}$, when $\mathtt{x}$ is regarded as an unsigned integer in $[0, \mathsf{umax}_n]$. Here, it is useful to extend KB by additional information on the intervals of the result. If $LV_{u,n}(t_1) \in [\ell_1, u_1]$ and $LV_{u,n}(t_2) \in [\ell_2, u_2]$ for numbers $\ell_1, \ell_2, u_1, u_2 \in \mathbb{N}$, then for $\ell b = \ell_1 \cdot \ell_2$ and $ub = u_1 \cdot u_2$, we have $LV_{u,n}(t_1) \cdot LV_{u,n}(t_2) \in [\ell b, ub]$. However, our goal is to infer

[7] If $\mathtt{y}, \mathtt{z} \in [0, 2^n - 1]$, then $\mathtt{y} \cdot \mathtt{z} \in [0, 2^{2 \cdot n} - 2^{n+1} + 1]$. So there are $\mathcal{O}(2^n)$ many potential intervals of size 2^n for the result, i.e., we would have to consider $\mathcal{O}(2^n)$ many cases.

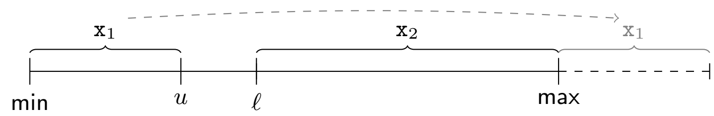

Fig. 4: Expressing unions of intervals

information on the possible value of $\mathsf{uns}_n(LV_{u,n}(t_1) \cdot LV_{u,n}(t_2))$.

To this end, we compute the size of the interval $[\ell b, ub]$. If $ub - \ell b + 1 \geq 2^n$, then $[\ell b, ub]$ contains more numbers than those that can be represented with n bits and $LV(\mathtt{x})$ can be any n-bit integer. Otherwise, we check whether $\mathsf{uns}_n(\ell b) \leq \mathsf{uns}_n(ub)$ holds. In this case, we add "$LV(\mathtt{x}) \in [\mathsf{uns}_n(\ell b), \mathsf{uns}_n(ub)]$" to KB. Finally, if the size of $[\ell b, ub]$ is $< 2^n$ but $\mathsf{uns}_n(\ell b) > \mathsf{uns}_n(ub)$, then $LV(\mathtt{x}) \in [0, \mathsf{uns}_n(ub)] \cup [\mathsf{uns}_n(\ell b), \mathsf{umax}_n]$, i.e., $LV(\mathtt{x})$ is not between the inner bounds $\mathsf{uns}_n(ub)$ and $\mathsf{uns}_n(\ell b)$, cf. Fig. 3. However, we cannot add "$LV(\mathtt{x}) \leq \mathsf{uns}_n(ub) \vee LV(\mathtt{x}) \geq \mathsf{uns}_n(\ell b)$" to KB as it contains "$\vee$", but KB is a conjunction of (in)equalities.

Hence, Thm. 6 shows how to express a condition of the form "$t \in [\mathsf{min}, u] \cup [\ell, \mathsf{max}]$" for $\mathsf{min} \leq u < \ell \leq \mathsf{max}$ by a single inequality. To this end, we subtract ℓ so that the second subinterval $[\ell, \mathsf{max}]$ ($\mathtt{x}_2$ in Fig. 4) starts with 0. Then we apply "mod 2^n" (this results in moving the first subinterval $\mathtt{x}_1$, cf. the dashed arrow in Fig. 4). Afterwards, we shift the whole interval back (by adding ℓ again).

Theorem 6 (Expressing Unions of Intervals in a Single Inequality). *Let $n \in \mathbb{N}_{>0}$, $\mathsf{min} \in \mathbb{Z}$, $\mathsf{max} = \mathsf{min} + 2^n - 1$, $t \in [\mathsf{min}, \mathsf{max}]$, and $\mathsf{min} \leq u < \ell \leq \mathsf{max}$. Let $\mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max})$ be the formula "$((t - \ell) \bmod 2^n) + \ell \ \leq \ 2^n + u$". Then we have $t \in [\mathsf{min}, u] \ \cup \ [\ell, \mathsf{max}]$ iff $\mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max})$ holds.*

unsigned `mul` (p: "`x = mul in` t_1, t_2" with $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi, \psi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{U}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- If $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) \cdot LV_{u,n}(t_2) \in [0, \mathsf{umax}_n])$, then φ is "$v = LV_{u,n}(t_1) \cdot LV_{u,n}(t_2)$". Otherwise, φ is "$v = \mathsf{uns}_n(LV_{u,n}(t_1) \cdot LV_{u,n}(t_2))$".
- $\ell_1, \ell_2, u_1, u_2 \in \mathbb{N}$ such that $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) \in [\ell_1, u_1] \wedge LV_{u,n}(t_2) \in [\ell_2, u_2])$
- $\ell b = \ell_1 \cdot \ell_2$ and $ub = u_1 \cdot u_2$
- If $ub - \ell b + 1 \geq 2^n$, then ψ is *true*. Otherwise, if $\mathsf{uns}_n(\ell b) \leq \mathsf{uns}_n(ub)$, then ψ is "$v \in [\mathsf{uns}_n(\ell b), \mathsf{uns}_n(ub)]$". Otherwise, ψ is $\mathsf{inBounds}(v, 0, \mathsf{uns}_n(ub), \mathsf{uns}_n(\ell b), \mathsf{umax}_n)$.

We have an analogous rule for signed multiplication by using $\mathtt{x} \in \mathcal{S}_\mathcal{P}$ instead of $\mathcal{U}_\mathcal{P}$, $LV_{s,n}$ instead of $LV_{u,n}$, smin_n and smax_n instead of 0 and umax_n, sig_n instead of uns_n, $\mathbb{Z}$ instead of $\mathbb{N}$, and by defining ℓb (resp. ub) as the minimum (resp. maximum) of $\{x_1 \cdot x_2 \mid x_1 \in [\ell_1, u_1], x_2 \in [\ell_2, u_2]\}$. Moreover, for signed multiplication with the flag "`nsw`", we reach ERR if $\not\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \cdot LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n])$. We also use similar rules for division and remainder (where LLVM has separate instructions for unsigned and signed integers), cf. [1].

3.3 Handling Bitwise Operations

For bitwise binary LLVM operations like "`and`" (computing bitwise logical conjunction), we also infer knowledge about the range of the result. For instance, the conjunction of 3 (0 1 1) and 5 (1 0 1) is 1 (0 0 1). So if "`x = and i`n t_1, t_2" and $\mathtt{x} \in \mathcal{U}_{\mathcal{P}}$, then $LV(\mathtt{x}) \leq LV_{u,n}(t_1)$ and $LV(\mathtt{x}) \leq LV_{u,n}(t_2)$, since a "1" on a position of the bitvector results in a larger number than a "0" on that position.

The same is true for signed integers, if both are positive or negative. So the conjunction of -1 (1 1 ... 1 1) and -2 (1 1 ... 1 0) is -2. The conjunction of a negative and a positive signed integer is at most as large as the positive integer.

signed `and` (p: **"`x = and i`n t_1, t_2" with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_{\mathcal{P}},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- $\ell_1, \ell_2, u_1, u_2 \in \mathbb{Z}$ such that $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \in [\ell_1, u_1] \wedge LV_{s,n}(t_2) \in [\ell_2, u_2])$
- If $\langle a \rangle \Rightarrow (LV_{s,n}(t_1) = LV_{s,n}(t_2))$, then φ is "$v = LV_{s,n}(t_1)$".
  Otherwise, if $\ell_1 \geq 0 \wedge \ell_2 \geq 0$ or $u_1 < 0 \wedge u_2 < 0$, φ is "$v \leq LV_{s,n}(t_1) \wedge v \leq LV_{s,n}(t_2)$".
  Otherwise, if $\ell_1 \geq 0$ then φ is "$v \leq LV_{s,n}(t_1)$" and if $\ell_2 \geq 0$ then φ is "$v \leq LV_{s,n}(t_2)$".
  Otherwise, φ is "$v \leq \max(u_1, u_2)$".

In the corresponding rule for unsigned `and`, φ is "$v = LV_{u,n}(t_1)$" if $\langle a \rangle \Rightarrow (LV_{u,n}(t_1) = LV_{u,n}(t_2))$. Otherwise, φ is "$v \leq LV_{u,n}(t_1) \wedge v \leq LV_{u,n}(t_2)$".

Moreover, we adapt the rules for conversion instructions (e.g., extension and truncation). *Sign extension* (`sext`) copies the most significant bit to all extension bits, while for zero extension (`zext`) only zeros are used. So for 1 0 1, the sign extension is 1 ... 1 1 0 1 and the zero extension is 0 ... 0 1 0 1. The following rule for `sext` (resp. `zext`) considers its argument as a signed (resp. unsigned) integer. Then these instructions do not change the value of their operands.

extension (p: **"`x = sext/zext i`n t `to i`m" with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$, $n < m$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and if}$$

either p : "`x = sext i`n t `to i`m", $\mathtt{x} \in \mathcal{S}_{\mathcal{P}}$, and φ is "$v = LV_{s,n}(t)$"
or p : "`x = zext i`n t `to i`m", $\mathtt{x} \in \mathcal{U}_{\mathcal{P}}$, and φ is "$v = LV_{u,n}(t)$"

The instruction `trunc` truncates a value to the n least significant bits. Similar to the rules for multiplication, we again use the operations sig_n (resp. uns_n) and $\mathsf{inBounds}$ to express our knowledge about the result of the truncation.

signed `trunc` (p: **"`x = trunc i`m t `to i`n" with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$, $n < m$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi, \psi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_{\mathcal{P}},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- If $\models \langle a \rangle \Rightarrow (LV_{s,m}(t) \in [\mathsf{smin}_n, \mathsf{smax}_n])$, then φ is "$v = LV_{s,m}(t)$".
  Otherwise, φ is "$v = \mathsf{sig}_n(LV_{s,m}(t))$".
- $\ell, u \in \mathbb{Z}$ such that $\models \langle a \rangle \Rightarrow (LV_{s,m}(t) \in [\ell, u])$
- If $u - \ell + 1 \geq 2^n$, then ψ is *true*.
  Otherwise, if $\mathsf{sig}_n(\ell) \leq \mathsf{sig}_n(u)$, then ψ is "$v \in [\mathsf{sig}_n(\ell), \mathsf{sig}_n(u)]$".
  Otherwise, ψ is $\mathsf{inBounds}(v, \mathsf{smin}_n, \mathsf{sig}_n(u), \mathsf{sig}_n(\ell), \mathsf{smax}_n)$.

In the rule for unsigned `trunc`, we have $\mathtt{x} \in \mathcal{U}_{\mathcal{P}}$ instead of $\mathcal{S}_{\mathcal{P}}$, $LV_{u,m}(t)$ instead of $LV_{s,m}(t)$, 0 and umax_n instead of smin_n and smax_n, and uns_n instead of sig_n.

4 From Symbolic Execution Graphs to Integer Systems

After the graph construction has been completed, we extract an *integer transition system* (ITS) from the cycles of the symbolic execution graph and then use existing tools to prove its termination.

ITSs can be represented as graphs whose nodes correspond to program locations and whose edges correspond to transitions. A transition is labeled with conditions that are required for its application. These conditions are quantifier-free formulas over a set of variables $\mathcal{V}$ and a corresponding set $\mathcal{V}' = \{x' \mid x \in \mathcal{V}\}$ which refers to the values of the variables *after* applying the transition.

The only cycle of the symbolic execution graph in Fig. 2 is the one from P to R and back. The resulting ITS is shown in Fig. 5. The values of the variables do not change in transitions that correspond to evaluation edges of the symbolic execution graph. For the generalization edge from R to P with the instantiation μ, the corresponding transition in the ITS gets the condition $v' = \mu(v)$ for all $v \in \mathcal{V}_{sym}(P)$. So we obtain the condition $v'_{\mathtt{inc}} = \mu(v_{\mathtt{inc}})$, i.e., $v'_{\mathtt{inc}} = v_{\mathtt{inc2}} = v_{\mathtt{inc}} + 1$. In contrast, $v_{\mathtt{inc2}}$'s value can change arbitrarily here, since $v_{\mathtt{inc2}} \notin \mathcal{V}_{sym}(P)$. Moreover, the transitions of the ITS contain conditions like $v_{\mathtt{inc}} \leq \mathsf{umax}_{32}$, which are also present in the states P – R. Standard tools can easily prove termination of this ITS. See [14] for details on extracting ITSs from symbolic execution graphs.

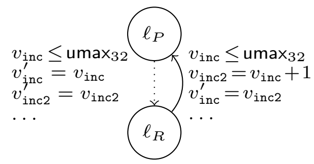

Fig. 5: ITS for function g

Recall that the bitvector arithmetic is covered by the rules to construct the symbolic execution graph, whereas the variables in the graph and in the resulting ITS range over $\mathbb{Z}$. Therefore, the following theorem from [14] still holds. It states that termination of the ITS implies termination of the analyzed LLVM program.

Theorem 7 (Termination). *Let $\mathcal{P}$ be an* LLVM *program with a complete symbolic execution graph $\mathcal{G}$ and let $\mathcal{I}_\mathcal{G}$ be the ITS resulting from $\mathcal{G}$. If $\mathcal{I}_\mathcal{G}$ terminates, then $\mathcal{P}$ also terminates for all concrete states represented by the states in $\mathcal{G}$.*

5 Related Work, Experiments, and Conclusion

We adapted our approach for proving memory safety and termination of C (resp. LLVM) programs to bitvectors. Since we represent bitvectors by relations on $\mathbb{Z}$, we can use standard SMT solving and standard termination analysis on $\mathbb{Z}$ for the symbolic execution and the termination proofs in our approach.

There are few other methods and tools for termination of bitvector programs (e.g., KITTeL [7,8], TAN [4,11], 2LS [2], Juggernaut [5], Ultimate [10][8]).[9] Com-

[8] However, there is not yet any paper describing Ultimate's adaption to bitvectors.

[9] Outside of termination analysis, there exist several tools for overflow detection. However, we cannot easily apply such external tools in our approach, since we want to use the result of potential overflows to continue our symbolic execution and analysis.

pared to related work, our approach has the following characteristics:

(a) Handling Memory: KITTeL, TAN, 2LS, and Juggernaut either do not handle dynamic data structures, strings, and arrays, or they abstract their properties to arithmetic ones. Thus, they fail for programs whose termination depends on explicit pointer arithmetic. Note that without considering the memory, termination of bitvector programs is decidable in *PSPACE* [4]. In contrast, our approach is the first which combines the handling of bitvectors with the precise representation of low-level memory operations, by using symbolic execution.

(b) Representation with $\mathbb{Z}$: Similar to KITTeL and the first approach in [4], we represent bitvectors by relations on $\mathbb{Z}$. In contrast, 2LS, Juggernaut, and the second approach in [4] use vectors of Boolean variables instead and reduce the termination problem to second-order satisfiability. This would have drawbacks when constructing symbolic execution graphs, where large numbers of SMT queries have to be solved. Here, using $\mathbb{Z}$ instead of bitvectors often simplifies the graph structure and lets us benefit from the efficiency of SMT solving over $\mathbb{Z}$.

(c) Unsigned resp. Signed Representation: We use a heuristic to determine whether we represent information about the unsigned or the signed value of variables in the states for symbolic execution. In contrast, KITTeL resp. the first approach of [4] represent only the signed resp. the unsigned values. The drawback is that then one needs a larger case analysis for instructions like `icmp ugt` resp. `sgt` which differ for unsigned and signed integers. Thus, this affects efficiency.

(d) Case Analysis vs. "Modulo": When representing bitvectors by relations on $\mathbb{Z}$, the wrap-around for overflows can either be handled by case analysis or by "modulo" relations. We use a hybrid approach with case analysis for instructions like addition (to avoid "modulo" which is less efficient for SMT solving) and with "modulo" for operations like multiplication (where case analysis could lead to an exponential blow-up). KITTeL only uses case analysis. While [4] also applies "modulo", our approach infers more complex relations about the ranges of variables, even if these ranges are unions of disjoint intervals. For an efficient SMT reasoning during symbolic execution, we express such "disjunctive properties" by single inequalities, cf. the formula $\mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max})$.

We implemented our approach in AProVE [14] using the SMT solvers Yices [6] and Z3 [13] in the back-end. The previous version of AProVE won the *SV-COMP* 2015 and 2016 competitions for termination of C programs (where tools were restricted to mathematical integers). To evaluate the new version of AProVE with bitvectors, we performed experiments on 118 C programs. We took the 61 *Windows Driver Development Kit* examples used for the evaluation of [4] and [8], 61 of the 62 examples from the repository of Juggernaut where we excluded one example containing `float`, 7 of the 9 examples of [5] where we excluded two examples with `float`, 4 new examples where termination depends on overflows of multiplication, and 4 new examples combining pointer and bitvector arithmetic. From these 137 examples, we removed 19 examples which are known to be non-terminating. Since Ultimate does not support bitvector arithmetic for signed

	T	F	TO	RT	T	F	TO	RT	%
AProVE	34	9	9	10.23	61	3	2	5.55	80.5
2LS	23	29	0	0.37	45	21	0	0.33	57.6
KITTeL	27	4	21	1.81	33	3	30	14.17	50.8
Juggernaut	10	19	23	34.12	22	26	18	6.22	27.1
Ultimate	–	–	–	–	11	54	1	12.77	16.7

Fig. 6: Experimental evaluation

integers yet, the right half of the table in Fig. 6 consists of those examples where termination does not depend on signed integers. We ran all tools in a mode where signed overflows are allowed and result in a wrap-around behavior.

Fig. 6 shows the performance of the tools for a time limit of 300 seconds per example on an Intel Core i7-950 with 6 GB memory. We did not compare with TAN, since it was outperformed by its successor 2LS in [2]. "**T**" is the number of examples where termination was proved, "**F**" states how often the termination proof failed in $\leq$ 300 seconds, "**TO**" is the number of time-outs, "**RT**" is the average run time in seconds for the examples where the tool showed termination, and "**%**" is the percentage of examples where termination was proved.

So on our collection (which mainly consists of the examples from the evaluations of the other tools), AProVE is most powerful. To evaluate the benefit of representing both unsigned and signed values (cf. ***(c)***), we also ran AProVE in a mode where all values are represented as signed integers (i.e., $\mathcal{S}_\mathcal{P} = \mathcal{V}_\mathcal{P}$). Here, we lost 11 termination proofs. To evaluate the use of case analysis vs. "modulo" (cf. ***(d)***), we tested a version of AProVE where we used "modulo" also for operations like addition. Here, we failed on 13 more examples. For details on our experiments and to access our implementation via a web interface, we refer to [1]. In future work, we plan to extend our approach to recursion, to inductive data `struct`ures, and to a compositional treatment of LLVM functions (the main challenge is to combine these tasks with the handling of explicit pointer arithmetic).

Acknowledgments. We are grateful to M. Heizmann, D. Kroening, M. Lewis, and P. Schrammel for their help with the experiments.

References

1. AProVE: `http://aprove.informatik.rwth-aachen.de/eval/Bitvectors/`
2. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wächter, B.: Synthesising interprocedural bit-precise termination proofs. In: Proc. ASE '15. pp. 53–64 (2015)
3. Clang compiler: `http://clang.llvm.org`
4. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.: Ranking function synthesis for bit-vector relations. Formal Methods in System Design 43(1), 93–120 (2013)
5. David, C., Kroening, D., Lewis, M.: Unrestricted termination and non-termination arguments for bit-vector programs. In: Proc. ESOP '15. pp. 183–204. LNCS 9032 (2015)

6. Dutertre, B., de Moura, L.M.: The Yices SMT solver (2006), Tool paper at http://yices.csl.sri.com/tool-paper.pdf
7. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Proc. RTA '11. pp. 41–50. LIPIcs 10 (2011)
8. Falke, S., Kapur, D., Sinz, C.: Termination analysis of imperative programs using bitvector arithmetic. In: Proc. VSTTE '12. pp. 261–277. LNCS 7152 (2012)
9. Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Proving termination of programs automatically with AProVE. In: Proc. IJCAR '14. pp. 184–191. LNAI 8562 (2014)
10. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Proc. ATVA '13. pp. 365–380. LNCS 8172 (2013)
11. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.: Termination analysis with compositional transition invariants. In: Proc. CAV '10. pp. 89–103. LNCS 6174 (2010)
12. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO '04. pp. 75–88 (2004)
13. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS '08. pp. 337–340. LNCS 4963 (2008)
14. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P., Aschermann, C.: Automatically proving termination and memory safety for programs with pointer arithmetic, submitted to the *Journal of Automated Reasoning*, available from [1]. Extended version of a paper in Proc. IJCAR '14. pp. 208-223. LNAI 8562 (2014)

A Separation Logic Semantics of Abstract States

To formalize the semantics of an abstract state a, in [14] we introduced a separation logic formula $\langle a \rangle_{SL}$, which extends $\langle a \rangle_{FO}$ by information about the memory (i.e., about AL and PT). In $\langle a \rangle_{SL}$, we combine the elements of AL with the separating conjunction "$*$" to express that different allocated memory blocks are disjoint. As usual, $\varphi_1 * \varphi_2$ means that φ_1 and φ_2 hold for disjoint parts of the memory. In contrast, the elements of PT are combined by the ordinary conjunction "$\wedge$". So $(v_1 \hookrightarrow_{\mathtt{ty},i} v_2) \in PT$ does not imply that v_1 is different from other addresses in PT. Similarly, we also combine the two formulas resulting from AL and PT by "$\wedge$", as both express different properties of the same addresses.

Definition 8 (SL Formulas for States). *For $v_1, v_2 \in \mathcal{V}_{sym}$, let $\langle [\![v_1,\, v_2]\!] \rangle_{SL} = (\forall x. \exists y.\ (v_1 \leq x \leq v_2) \Rightarrow (x \hookrightarrow y))$. For any* LLVM *type* `ty`*, we define*

$$\langle v_1 \hookrightarrow_{\mathtt{ty},u} v_2 \rangle_{SL} \;=\; \langle v_1 \hookrightarrow_{size(\mathtt{ty})} v_2 \rangle_{SL}.$$

To handle the two's complement representation of signed integers, we define $\langle v_1 \hookrightarrow_{\mathtt{ty},s} v_2 \rangle_{SL} =$

$$\langle v_1 \hookrightarrow_{size(\mathtt{ty})} v_3 \rangle_{SL} \;\wedge\; (v_2 \geq 0 \;\Rightarrow\; v_3 = v_2) \;\wedge\; (v_2 < 0 \;\Rightarrow\; v_3 = v_2 + 2^{size(\mathtt{ty})}),$$

where $v_3 \in \mathcal{V}_{sym}$ is fresh. We assume a little-endian data layout (where least significant bytes are stored in the lowest address). Hence, we define $\langle v_1 \hookrightarrow_0 v_3 \rangle_{SL} = true$ and $\langle v_1 \hookrightarrow_{n+8} v_3 \rangle_{SL} = (v_1 \hookrightarrow (v_3 \bmod 2^8)) \wedge \langle\, (v_1 + 1) \hookrightarrow_n (v_3 \operatorname{div} 2^8) \,\rangle_{SL}$.

A state $a = (p, LV, KB, AL, PT)$ is represented in separation logic by

$$\langle a\rangle_{SL} = \langle a\rangle_{FO} \ \wedge\ (\mathop{\ast}_{\varphi\in AL}\ \langle\varphi\rangle_{SL})\ \wedge\ (\bigwedge_{\varphi\in PT}\ \langle\varphi\rangle_{SL}).$$

We use *interpretations* (as, mem) for the semantics of separation logic (Sect. 2).

Definition 9 (Semantics of Separation Logic). *Let* $as : \mathcal{V}_{\mathcal{P}} \to \mathbb{Z}$ *be an assignment,* $mem : \mathbb{N}_{>0} \rightharpoonup \{0,\dots,\mathsf{umax}_8\}$*, and* φ *be a formula. Let* $as(\varphi)$ *result from replacing all local variables* `x` *in* φ *by the value* $as(\mathtt{x})$*. By construction, local variables* `x` *are never quantified in our formulas. Then we define* $(as, mem) \models \varphi$ *iff* $mem \models as(\varphi)$*.*

We now define $mem \models \psi$ *for formulas* ψ *that may contain symbolic variables from* $\mathcal{V}_{sym}$*. As usual, all free variables* $v_1,\dots,v_n$ *in* ψ *are implicitly universally quantified, i.e.,* $mem \models \psi$ *iff* $mem \models \forall v_1,\dots,v_n.\,\psi$*. The semantics of arithmetic operations and predicates as well as of first-order connectives and quantifiers are as usual. In particular, we define* $mem \models \forall v.\,\psi$ *iff* $mem \models \sigma(\psi)$ *holds for all instantiations* σ *where* $\sigma(v) \in \mathbb{Z}$ *and* $\sigma(w) = w$ *for all* $w \in \mathcal{V}_{sym} \setminus \{v\}$*.*

We still have to define the semantics of $\hookrightarrow$ *and* $\ast$ *for variable-free formulas. For* $n_1, n_2 \in \mathbb{Z}$*, let* $mem \models n_1 \hookrightarrow n_2$ *hold iff* $mem(n_1) = n_2$*.*[10] *The semantics of* $\ast$ *is defined as usual in separation logic: For two partial functions* $mem_1, mem_2 : \mathbb{N}_{>0} \rightharpoonup \mathbb{Z}$*, we write* $mem_1 \bot mem_2$ *to indicate that the domains of* mem_1 *and* mem_2 *are disjoint. If* $mem_1 \bot mem_2$*, then* $mem_1 \uplus mem_2$ *denotes the union of* mem_1 *and* mem_2*. Now* $mem \models \varphi_1 \ast \varphi_2$ *holds iff there exist* $mem_1 \bot mem_2$ *such that* $mem = mem_1 \uplus mem_2$ *where* $mem_1 \models \varphi_1$ *and* $mem_2 \models \varphi_2$*.*

B Proofs

Proof of Thm. 5. Since the result of "mod 2^n" is always in the interval $[0, 2^n - 1]$, we immediately obtain $\mathsf{sig}_n(t) = ((t + 2^{n-1}) \bmod 2^n) - 2^{n-1} \in [0 - 2^{n-1}, 2^n - 1 - 2^{n-1}] = [-2^{n-1}, 2^{n-1} - 1] = [\mathsf{smin}_n, \mathsf{smax}_n]$. Moreover, we have

$$\begin{aligned}
&\ t \bmod 2^n\\
=&\ (t + 2^{n-1} - 2^{n-1}) \bmod 2^n\\
=&\ (((t + 2^{n-1}) \bmod 2^n) - 2^{n-1}) \bmod 2^n\\
=&\ \mathsf{sig}_n(t) \bmod 2^n.
\end{aligned}$$

□

Proof of Thm. 6. We consider three cases.

<u>Case 1: $t \in [\mathsf{min}, u]$</u>

Clearly, $u < \ell$ implies $u - \ell < 0$. Moreover, we also have $u - \ell \geq \mathsf{min} - \mathsf{max} = -2^n + 1$, which together implies

[10] We use "$\hookrightarrow$" instead of "$\mapsto$" in separation logic, since $mem \models n_1 \mapsto n_2$ would imply that $mem(n)$ is undefined for all $n \neq n_1$. This would be inconvenient in our formalization, since PT usually only contains information about a *part* of the allocated memory.

$$-2^n < u - \ell < 0. \tag{2}$$

Thus, we have:
$$\begin{aligned} t \leq u \quad &\Rightarrow \quad t - \ell \leq u - \ell \\ &\Rightarrow \quad (t-\ell) \mod 2^n \leq u - \ell + 2^n \qquad \text{by (2)} \\ &\Rightarrow \quad ((t-\ell) \mod 2^n) + \ell \leq u + 2^n \\ &\Rightarrow \quad \mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max}) \text{ holds} \end{aligned}$$

Case 2: $t \in [u+1, \ell-1]$

This entails $u+1 \leq \ell - 1$, i.e., $u - \ell + 1 < 0$. Moreover, we also have $u - \ell + 1 \geq \mathsf{min} - \mathsf{max} + 1 = -2^n + 2$, which together implies

$$-2^n < u - \ell + 1 < 0. \tag{3}$$

We obtain:
$$\begin{aligned} t \geq u+1 \quad &\Rightarrow \quad t - \ell \geq u - \ell + 1 \\ &\Rightarrow \quad (t-\ell) \mod 2^n \geq u - \ell + 1 + 2^n \qquad \text{by (3)} \\ &\Rightarrow \quad ((t-\ell) \mod 2^n) + \ell \geq u + 1 + 2^n \\ &\Rightarrow \quad \mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max}) \text{ does not hold} \end{aligned}$$

Case 3: $t \in [\ell, \mathsf{max}]$

Note that $\mathsf{max} - \ell \geq 0$ and moreover, $\mathsf{max} - \ell < \mathsf{max} - \mathsf{min} = 2^n - 1$, i.e.,

$$0 \leq \mathsf{max} - \ell < 2^n. \tag{4}$$

In addition, we have

$$\mathsf{max} = \mathsf{min} + 2^n - 1 \leq u + 2^n - 1. \tag{5}$$

Here, we obtain:
$$\begin{aligned} t \leq \mathsf{max} \quad &\Rightarrow \quad t - \ell \leq \mathsf{max} - \ell \\ &\Rightarrow \quad (t-\ell) \mod 2^n \leq \mathsf{max} - \ell \qquad \text{by (4)} \\ &\Rightarrow \quad ((t-\ell) \mod 2^n) + \ell \leq \mathsf{max} \\ &\Rightarrow \quad ((t-\ell) \mod 2^n) + \ell \leq u + 2^n \qquad \text{by (5)} \\ &\Rightarrow \quad \mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max}) \text{ holds} \end{aligned}$$
□

Proof of Thm. 7. The proof of Thm. 7 is identical to the proofs of Thm. 10 and 13 in [14]. It relies on the fact that our symbolic execution rules correspond to the actual execution of LLVM when they are applied to concrete states (this also holds for the new bitvector rules of the current paper). So if a concrete state c is represented in the symbolic execution graph, then every LLVM evaluation of c corresponds to a path in the graph. The generation of an ITS from the graph is done in such a way that termination of the ITS implies that there is no such infinite path in the graph. As all integers in the symbolic execution graphs and in the ITSs are still *mathematical* integers, the construction of ITSs has not changed in the current paper, i.e., the corresponding proof of [14] directly carries over to the present setting.

C Symbolic Execution Rules

In this section we give symbolic execution rules for all (interesting) LLVM instructions that are affected by the bitvector semantics. We start with the rules for comparisons of unsigned and signed integers. Here, one may have to perform a refinement according to the cases of the comparison.

`icmp ugt` refinement (p: "`x = icmp ugt ty` t_1, t_2" with $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)} \quad \text{if}$$

φ is "$LV_{u,size(\mathtt{ty})}(t_1) > LV_{u,size(\mathtt{ty})}(t_2)$" and we have both $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$

`icmp sgt` refinement (p: "`x = icmp sgt ty` t_1, t_2" with $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)} \quad \text{if}$$

φ is "$LV_{s,size(\mathtt{ty})}(t_1) > LV_{s,size(\mathtt{ty})}(t_2)$" and we have both $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$

The refinement rules for `icmp` (`uge`|`ult`|`ule`|`sge`|`slt`|`sle`|`eq`|`ne`) are analogous.

We now show the rules to evaluate the comparison instruction.

`icmp ugt` (p: "`x = icmp ugt ty` t_1, t_2" with $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and if}$$

either $\models \langle a \rangle \Rightarrow (LV_{u,size(\mathtt{ty})}(t_1) > LV_{u,size(\mathtt{ty})}(t_2))$ and φ is "$v = 1$"
or $\models \langle a \rangle \Rightarrow (LV_{u,size(\mathtt{ty})}(t_1) \leq LV_{u,size(\mathtt{ty})}(t_2))$ and φ is "$v = 0$"

`icmp sgt` (p: "`x = icmp sgt ty` t_1, t_2" with $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and if}$$

either $\models \langle a \rangle \Rightarrow (LV_{s,size(\mathtt{ty})}(t_1) > LV_{s,size(\mathtt{ty})}(t_2))$ and φ is "$v = 1$"
or $\models \langle a \rangle \Rightarrow (LV_{s,size(\mathtt{ty})}(t_1) \leq LV_{s,size(\mathtt{ty})}(t_2))$ and φ is "$v = 0$"

The evaluation rules for `icmp` (`uge`|`ult`|`ule`|`sge`|`slt`|`sle`|`eq`|`ne`) are analogous.

Next, we present rules for arithmetic operations like addition. Here, we perform a refinement (i.e., a case analysis) to distinguish the cases where an overflow occurs from those cases where an overflow cannot happen.

unsigned `add` refinement (p: "`x = add i`n t_1, t_2" with $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)} \quad \text{if } \mathtt{x} \in \mathcal{U}_{\mathcal{P}} \text{ and}$$

φ is "$LV_{u,n}(t_1) + LV_{u,n}(t_2) \leq \mathsf{umax}_n$", where $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$

signed add refinement (p: "$\mathtt{x = add\ i}n\ t_1, t_2$" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi_1\}, AL, PT) \ \mid\ (p, LV, KB \cup \{\varphi_2\}, AL, PT) \ \mid\ (p, LV, KB \cup \{\varphi_3\}, AL, PT)} \quad \text{if}$$

- $\mathtt{x} \in \mathcal{S}_\mathcal{P}$
- φ_1 is "$LV_{s,n}(t_1) + LV_{s,n}(t_2) < \mathsf{smin}_n$"
- φ_2 is "$LV_{s,n}(t_1) + LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n]$"
- φ_3 is "$LV_{s,n}(t_1) + LV_{s,n}(t_2) > \mathsf{smax}_n$"
- $\not\models \langle a \rangle \Rightarrow \varphi_1$, $\not\models \langle a \rangle \Rightarrow \varphi_2$, and $\not\models \langle a \rangle \Rightarrow \varphi_3$

Now we show the rules to evaluate the `add` instruction. The first rule handles the case where overflow does not occur.

`add` without overflow (p: "$\mathtt{x = add\ [nsw]\ i}n\ t_1, t_2$" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and if}$$

either $\mathtt{x} \in \mathcal{U}_\mathcal{P}$, $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) + LV_{u,n}(t_2) \in [0, \mathsf{umax}_n])$,
and φ is "$v = LV_{u,n}(t_1) + LV_{u,n}(t_2)$"
or $\mathtt{x} \in \mathcal{S}_\mathcal{P}$, $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) + LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n])$,
and φ is "$v = LV_{s,n}(t_1) + LV_{s,n}(t_2)$"

In the next three rules, we consider the case where we have an overflow.

unsigned `add` with overflow (p: "$\mathtt{x = add\ i}n\ t_1, t_2$" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{v = LV_{u,n}(t_1) + LV_{u,n}(t_2) - 2^n\},\ AL,\ PT)} \quad \text{if}$$

$\mathtt{x} \in \mathcal{U}_\mathcal{P}$, $v \in \mathcal{V}_{sym}$ is fresh, and $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) + LV_{u,n}(t_2) > \mathsf{umax}_n)$

signed `add` with overflow (p: "$\mathtt{x = add\ i}n\ t_1, t_2$" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and either}$$

$\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) + LV_{s,n}(t_2) > \mathsf{smax}_n)$ and φ is "$v = LV_{s,n}(t_1) + LV_{s,n}(t_2) - 2^n$"
or $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) + LV_{s,n}(t_2) < \mathsf{smin}_n)$ and φ is "$v = LV_{s,n}(t_1) + LV_{s,n}(t_2) + 2^n$"

signed `add` with `nsw` overflow (p: "$\mathtt{x = add\ nsw\ i}n\ t_1, t_2$", $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P} \text{ and } \not\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) + LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n])$$

The rules for the subtraction instruction `sub` are analogous to the rules for `add`.

For multiplication (`mul`), handling overflow by case analysis is not feasible. Therefore, we use formulas containing "modulo" instead.

unsigned `mul` **(**p**: "**`x = mul i`n t_1, t_2**" with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi, \psi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{U}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- If $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) \cdot LV_{u,n}(t_2) \in [0, \mathsf{umax}_n])$, then φ is "$v = LV_{u,n}(t_1) \cdot LV_{u,n}(t_2)$". Otherwise, φ is "$v = \mathsf{uns}_n(LV_{u,n}(t_1) \cdot LV_{u,n}(t_2))$".
- $\ell_1, \ell_2, u_1, u_2 \in \mathbb{N}$ such that $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) \in [\ell_1, u_1] \wedge LV_{u,n}(t_2) \in [\ell_2, u_2])$
- $\ell b = \ell_1 \cdot \ell_2$ and $ub = u_1 \cdot u_2$
- If $ub - \ell b + 1 \geq 2^n$, then ψ is $true$. Otherwise, if $\mathsf{uns}_n(\ell b) \leq \mathsf{uns}_n(ub)$, then ψ is "$v \in [\mathsf{uns}_n(\ell b), \mathsf{uns}_n(ub)]$". Otherwise, ψ is $\mathsf{inBounds}(v, 0, \mathsf{uns}_n(ub), \mathsf{uns}_n(\ell b), \mathsf{umax}_n)$.

signed `mul` **(**p**: "**`x = mul i`n t_1, t_2**" with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi, \psi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- If $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \cdot LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n])$, φ is "$v = LV_{s,n}(t_1) \cdot LV_{s,n}(t_2)$". Otherwise, φ is "$v = \mathsf{sig}_n(LV_{s,n}(t_1) \cdot LV_{s,n}(t_2))$".
- $\ell_1, \ell_2, u_1, u_2 \in \mathbb{Z}$ such that $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \in [\ell_1, u_1] \wedge LV_{s,n}(t_2) \in [\ell_2, u_2])$
- $\ell b = \min\{x_1 \cdot x_2 \mid x_1 \in [\ell_1, u_1], x_2 \in [\ell_2, u_2]\}$ $ub = \max\{x_1 \cdot x_2 \mid x_1 \in [\ell_1, u_1], x_2 \in [\ell_2, u_2]\}$
- If $ub - \ell b + 1 \geq 2^n$, then ψ is $true$. Otherwise, if $\mathsf{sig}_n(\ell b) \leq \mathsf{sig}_n(ub)$, then ψ is "$v \in [\mathsf{sig}_n(\ell b), \mathsf{sig}_n(ub)]$". Otherwise, ψ is $\mathsf{inBounds}(v, \mathsf{smin}_n, \mathsf{sig}_n(ub), \mathsf{sig}_n(\ell b), \mathsf{smax}_n)$.

`mul nsw` **without overflow (**p**: "**`x = mul nsw i`n t_1, t_2**",** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi, \psi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh,}$$

$\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \cdot LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n])$, and φ is "$v = LV_{s,n}(t_1) \cdot LV_{s,n}(t_2)$".

`mul nsw` **with overflow (**p**: "**`x = mul nsw i`n t_1, t_2**",** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P} \text{ and } \not\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \cdot LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n])$$

In contrast to addition, subtraction, and multiplication, signed and unsigned division are operationally different and thus, LLVM has separate instructions for them. For unsigned division, an error only occurs when dividing by 0.

`udiv` **(**p**: "**`x = udiv i`n t_1, t_2**" with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and}$$

- $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_2) > 0)$
- φ is "$v \cdot LV_{u,n}(t_2) \leq LV_{u,n}(t_1)\ \wedge\ (v+1) \cdot LV_{u,n}(t_2) > LV_{u,n}(t_1)$"

udiv by zero (p: "`x = udiv i`n t_1, t_2", $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if} \not\models \langle a \rangle \Rightarrow (LV_{u,n}(t_2) > 0)$$

The rules for signed division are analogous. Here, an overflow happens when dividing smin_n by -1. This instruction does not have the possible flag "`nsw`", but it always leads to undefined behavior in case of an overflow.

sdiv without overflow (p: "`x = sdiv i`n t_1, t_2" **with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and}$$

- if $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \geq 0 \wedge LV_{s,n}(t_2) > 0)$,
  then φ is "$v \cdot LV_{s,n}(t_2) \leq LV_{s,n}(t_1) \ \wedge\ (v+1) \cdot LV_{s,n}(t_2) > LV_{s,n}(t_1)$"
- if $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) < 0 \wedge LV_{s,n}(t_2) > 0)$,
  then φ is "$(v-1) \cdot LV_{s,n}(t_2) < LV_{s,n}(t_1) \ \wedge\ v \cdot LV_{s,n}(t_2) \geq LV_{s,n}(t_1)$"
- if $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \geq 0 \wedge LV_{s,n}(t_2) < 0)$,
  then φ is "$v \cdot LV_{s,n}(t_2) \leq LV_{s,n}(t_1) \ \wedge\ (v-1) \cdot LV_{s,n}(t_2) > LV_{s,n}(t_1)$"
- if $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) < 0 \wedge LV_{s,n}(t_2) < 0)$
  and $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \neq \mathsf{smin}_n \vee LV_{s,n}(t_2) \neq -1)$,
  then φ is "$(v+1) \cdot LV_{s,n}(t_2) < LV_{s,n}(t_1) \ \wedge\ v \cdot LV_{s,n}(t_2) \geq LV_{s,n}(t_1)$"
- otherwise, we must have $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_2) \neq 0)$
  and $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \neq \mathsf{smin}_n \vee LV_{s,n}(t_2) \neq -1)$,
  and φ is *true*

sdiv with overflow (p: "`x = sdiv i`n t_1, t_2", $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if}$$

$\not\models \langle a \rangle \Rightarrow (LV_{s,n}(t_2) \neq 0)$ or
$\not\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \neq \mathsf{smin}_n \vee LV_{s,n}(t_2) \neq -1)$

The rules for unsigned and signed remainder (`urem` and `srem`) are analogous to the rules for `udiv` and `sdiv`.

Next we present rules for the logical right-shift. After the shift, the (new) most significant bits are filled with zeros. Here, an operand of type `i`n may be shifted by at most $n-1$ bits.

lshr (p: "`x = lshr i`n t_1, t_2" **with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and}$$

- $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_2) < n)$
- φ is "$v \cdot 2^{LV_{u,n}(t_2)} \leq LV_{u,n}(t_1) \ \wedge\ (v+1) \cdot 2^{LV_{u,n}(t_2)} > LV_{u,n}(t_1)$"

undefined `lshr` **(**p: **"**`x = lshr i`n t_1**,** t_2**",** $\mathtt{x}\in\mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if} \not\models \langle a\rangle \Rightarrow (LV_{u,n}(t_2) < n)$$

In the arithmetic right-shift, the new bits are filled with the most significant bit. Note that although the second operand of a shift is always interpreted as an unsigned integer in LLVM, the correctness of our rule is not affected if our heuristic regards the second operand t_2 as "signed" (e.g., t_2 could be a variable $\mathtt{y} \in \mathcal{S}_\mathcal{P}$). The reason is that we consider $LV_{u,n}(t_2)$ in the following rules for `ashr`, where $LV_{u,n}$ converts its argument into the corresponding unsigned value. In fact, the shift operation for an operand t_1 of type `i`n is again only defined if $LV_{u,n}(t_2) < n$. Note that $LV_{u,n}(t_2) < n$ even implies $LV_{u,n}(t_2) = LV_{s,n}(t_2)$, i.e., t_2 cannot be a value whose signed interpretation is negative.

`ashr` **(**p: **"**`x = ashr i`n t_1**,** t_2**" with** $\mathtt{x}\in\mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and}$$

- $\models \langle a\rangle \Rightarrow (LV_{u,n}(t_2) < n)$
- φ is "$v \cdot 2^{LV_{u,n}(t_2)} \le LV_{s,n}(t_1)\ \wedge\ (v+1)\cdot 2^{LV_{u,n}(t_2)} > LV_{s,n}(t_1)$"

undefined `ashr` **(**p: **"**`x = ashr i`n t_1**,** t_2**",** $\mathtt{x}\in\mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if} \not\models \langle a\rangle \Rightarrow (LV_{u,n}(t_2) < n)$$

Now we present corresponding rules for the left-shift instruction `shl`.

undefined `shl` **(**p: **"**`x = shl nsw i`n t_1**,** t_2**" with** $\mathtt{x}\in\mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if} \not\models \langle a\rangle \Rightarrow (LV_{u,n}(t_2) < n)$$

unsigned `shl` **(**p: **"**`x = shl i`n t_1**,** t_2**" with** $\mathtt{x}\in\mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{U}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- $\models \langle a\rangle \Rightarrow (LV_{u,n}(t_2) < n)$
- If $\models \langle a\rangle \Rightarrow (LV_{u,n}(t_1)\cdot 2^{LV_{u,n}(t_2)} \in [0, \mathsf{umax}_n])$, then φ is "$v = LV_{u,n}(t_1)\cdot 2^{LV_{u,n}(t_2)}$". Otherwise, φ is "$v = \mathsf{uns}_n(LV_{u,n}(t_1)\cdot 2^{LV_{u,n}(t_2)})$".

signed `shl` **(**p: **"**`x = shl i`n t_1**,** t_2**" with** $\mathtt{x}\in\mathcal{V}_\mathcal{P}$**,** $t_1, t_2 \in \mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**)**

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- $\models \langle a\rangle \Rightarrow (LV_{u,n}(t_2) < n)$
- If $\models \langle a\rangle \Rightarrow (LV_{s,n}(t_1)\cdot 2^{LV_{u,n}(t_2)} \in [\mathsf{smin}_n, \mathsf{smax}_n])$, φ is "$v = LV_{s,n}(t_1)\cdot 2^{LV_{u,n}(t_2)}$". Otherwise, φ is "$v = \mathsf{sig}_n(LV_{s,n}(t_1)\cdot 2^{LV_{u,n}(t_2)})$".

`shl nsw` without overflow (p: "`x = shl nsw i`n t_1, t_2", $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_2) < n)$
- $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \cdot 2^{LV_{u,n}(t_2)} \in [\mathsf{smin}_n, \mathsf{smax}_n])$ and φ is "$v = LV_{s,n}(t_1) \cdot 2^{LV_{u,n}(t_2)}$".

`shl nsw` with overflow (p: "`x = shl nsw i`n t_1, t_2", $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if } \not\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \cdot 2^{LV_{u,n}(t_2)} \in [\mathsf{smin}_n, \mathsf{smax}_n])$$

For the logical conjunction or disjunction, we have two rules, depending on whether the result variable is considered to be unsigned or signed.

unsigned `and` (p: "`x = and i`n t_1, t_2" with $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{U}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

If $\langle a \rangle \Rightarrow (LV_{u,n}(t_1) = LV_{u,n}(t_2))$, then φ is "$v = LV_{u,n}(t_1)$".
Otherwise, φ is "$v \leq LV_{u,n}(t_1) \wedge v \leq LV_{u,n}(t_2)$".

signed `and` (p: "`x = and i`n t_1, t_2" with $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- $\ell_1, \ell_2, u_1, u_2 \in \mathbb{Z}$ such that $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \in [\ell_1, u_1] \wedge LV_{s,n}(t_2) \in [\ell_2, u_2])$
- If $\langle a \rangle \Rightarrow (LV_{s,n}(t_1) = LV_{s,n}(t_2))$, then φ is "$v = LV_{s,n}(t_1)$".
  Otherwise, if $\ell_1 \geq 0 \wedge \ell_2 \geq 0$ or $u_1 < 0 \wedge u_2 < 0$, φ is "$v \leq LV_{s,n}(t_1) \wedge v \leq LV_{s,n}(t_2)$".
  Otherwise, if $\ell_1 \geq 0$ then φ is "$v \leq LV_{s,n}(t_1)$" and if $\ell_2 \geq 0$ then φ is "$v \leq LV_{s,n}(t_2)$".
  Otherwise, φ is "$v \leq \max(u_1, u_2)$".

unsigned `or` (p: "`x = or i`n t_1, t_2" with $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{U}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

If $\langle a \rangle \Rightarrow (LV_{u,n}(t_1) = LV_{u,n}(t_2))$, then φ is "$v = LV_{u,n}(t_1)$".
Otherwise, φ is "$v \geq LV_{u,n}(t_1) \wedge v \geq LV_{u,n}(t_2)$".

signed `or` (p: "`x = or i`n t_1, t_2" with $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- $\ell_1, \ell_2, u_1, u_2 \in \mathbb{Z}$ such that $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \in [\ell_1, u_1] \wedge LV_{s,n}(t_2) \in [\ell_2, u_2])$
- If $\langle a \rangle \Rightarrow (LV_{s,n}(t_1) = LV_{s,n}(t_2))$, then φ is "$v = LV_{s,n}(t_1)$".
  Otherwise, if $\ell_1 \geq 0 \wedge \ell_2 \geq 0$ or $u_1 < 0 \wedge u_2 < 0$, φ is "$v \geq LV_{s,n}(t_1) \wedge v \geq LV_{s,n}(t_2)$".
  Otherwise, if $u_1 < 0$ then φ is "$v \geq LV_{s,n}(t_1)$" and if $u_2 < 0$, then φ is "$v \geq LV_{s,n}(t_2)$".
  Otherwise, φ is "$v \geq \min(\ell_1, \ell_2)$".

Next we also present a rule for exclusive disjunction.

xor (p: "$\texttt{x = xor i}n\ t_1, t_2$" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and}$$

- if $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \geq 0 \wedge LV_{s,n}(t_2) \geq 0)$
  or $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) < 0 \wedge LV_{s,n}(t_2) < 0)$,
  then φ is "$LV_{s,n}(v) \geq 0$"
- if $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) \geq 0 \wedge LV_{s,n}(t_2) < 0)$
  or $\models \langle a \rangle \Rightarrow (LV_{s,n}(t_1) < 0 \wedge LV_{s,n}(t_2) \geq 0)$,
  then φ is "$LV_{s,n}(v) < 0$"
- otherwise, φ is "*true*"

The next rule is used to extend an n-bit integer to an m-bit integer by filling the new bits with the most significant bit (for `sext`) or by zeros (for `zext`).

extension (p: "$\texttt{x = sext/zext i}n\ t\ \texttt{to i}m$" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$, $n < m$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh and if}$$

either p : "$\texttt{x = sext i}n\ t\ \texttt{to i}m$", $\mathtt{x} \in \mathcal{S}_\mathcal{P}$, and φ is "$v = LV_{s,n}(t)$"
or p : "$\texttt{x = zext i}n\ t\ \texttt{to i}m$", $\mathtt{x} \in \mathcal{U}_\mathcal{P}$, and φ is "$v = LV_{u,n}(t)$"

Now we present an unsigned and a signed variant of the `trunc` instruction that truncates a value to the n least significant bits.

unsigned `trunc` (p: "$\texttt{x = trunc i}m\ t\ \texttt{to i}n$" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$, $n < m$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi, \psi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{U}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- If $\models \langle a \rangle \Rightarrow (LV_{u,m}(t) \in [0, \mathsf{umax}_n])$, then φ is "$v = LV_{u,m}(t)$".
  Otherwise, φ is "$v = \mathsf{uns}_n(LV_{u,m}(t))$".
- $\ell, u \in \mathbb{N}$ such that $\models \langle a \rangle \Rightarrow (LV_{u,m}(t) \in [\ell, u])$
- If $u - \ell + 1 \geq 2^n$, then ψ is *true*.
  Otherwise, if $\mathsf{uns}_n(\ell) \leq \mathsf{uns}_n(u)$, then ψ is "$v \in [\mathsf{uns}_n(\ell), \mathsf{uns}_n(u)]$".
  Otherwise, ψ is $\mathsf{inBounds}(v, 0, \mathsf{uns}_n(u), \mathsf{uns}_n(\ell), \mathsf{umax}_n)$.

signed `trunc` (p: "$\texttt{x = trunc i}m\ t\ \texttt{to i}n$" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$, $n < m$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi, \psi\},\ AL,\ PT)} \quad \text{if } \mathtt{x} \in \mathcal{S}_\mathcal{P},\ v \in \mathcal{V}_{sym} \text{ is fresh, and}$$

- If $\models \langle a \rangle \Rightarrow (LV_{s,m}(t) \in [\mathsf{smin}_n, \mathsf{smax}_n])$, then φ is "$v = LV_{s,m}(t)$".
  Otherwise, φ is "$v = \mathsf{sig}_n(LV_{s,m}(t))$".
- $\ell, u \in \mathbb{Z}$ such that $\models \langle a \rangle \Rightarrow (LV_{s,m}(t) \in [\ell, u])$
- If $u - \ell + 1 \geq 2^n$, then ψ is *true*.
  Otherwise, if $\mathsf{sig}_n(\ell) \leq \mathsf{sig}_n(u)$, then ψ is "$v \in [\mathsf{sig}_n(\ell), \mathsf{sig}_n(u)]$".
  Otherwise, ψ is $\mathsf{inBounds}(v, \mathsf{smin}_n, \mathsf{sig}_n(u), \mathsf{sig}_n(\ell), \mathsf{smax}_n)$.

Finally, we handle the `store` instruction that stores an integer value at a certain address in the memory. We present two versions of the rule, depending on whether one wants to store the value as an unsigned or as a signed integer. Here, for any terms t_1, t_2, "$[\![t_1,\ t_2]\!] \perp [\![t'_1,\ t'_2]\!]$" is a shorthand for $t_2 < t'_1 \vee t'_2 < t_1$.

unsigned `store` (p: "`store ty` t`, ty* ad`" **with** $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$, `ad` $\in \mathcal{V}_\mathcal{P}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV,\ KB \cup \{v = LV_{u,n}(t)\},\ AL,\ PT' \cup \{LV_{u,n}(\mathtt{ad}) \hookrightarrow_{\mathtt{ty},u} v\})} \quad \text{if}$$

- $t \in \mathcal{U}_\mathcal{P}$, or $t \in \mathbb{Z}$ and there is no $t' \in \mathcal{S}_\mathcal{P}$ s.t. t' is loaded from or stored to `ad` in $\mathcal{P}$
- there is $[\![v_1, v_2]\!] \in AL$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV_{u,n}(\mathtt{ad}) \wedge ad_{end} \leq v_2)$
- $PT' = \{(w_1 \hookrightarrow_{\mathtt{sy},i} w_2) \in PT \mid\ \models \langle a \rangle \Rightarrow ([\![LV_{u,n}(\mathtt{ad}), ad_{end}]\!] \perp [\![w_1,\ w_{1\,end}]\!])\}$
- $ad_{end} = LV_{u,n}(\mathtt{ad}) + \lceil \frac{size(\mathtt{ty})}{8} \rceil - 1$ and $w_{1\,end} = w_1 + \lceil \frac{size(\mathtt{sy})}{8} \rceil - 1$
- $v \in \mathcal{V}_{sym}$ is fresh

signed `store` (p: "`store ty` t`, ty* ad`" **with** $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$, `ad` $\in \mathcal{V}_\mathcal{P}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV,\ KB \cup \{v = LV_{s,n}(t)\},\ AL,\ PT' \cup \{LV_{u,n}(\mathtt{ad}) \hookrightarrow_{\mathtt{ty},s} v\})} \quad \text{if}$$

- $t \in \mathcal{S}_\mathcal{P}$, or $t \in \mathbb{Z}$ and there is a $t' \in \mathcal{S}_\mathcal{P}$ s.t. t' is loaded from or stored to `ad` in $\mathcal{P}$
- there is $[\![v_1, v_2]\!] \in AL$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV_{u,n}(\mathtt{ad}) \wedge ad_{end} \leq v_2)$
- $PT' = \{(w_1 \hookrightarrow_{\mathtt{sy},i} w_2) \in PT \mid\ \models \langle a \rangle \Rightarrow ([\![LV_{u,n}(\mathtt{ad}), ad_{end}]\!] \perp [\![w_1,\ w_{1\,end}]\!])\}$
- $ad_{end} = LV_{u,n}(\mathtt{ad}) + \lceil \frac{size(\mathtt{ty})}{8} \rceil - 1$ and $w_{1\,end} = w_1 + \lceil \frac{size(\mathtt{sy})}{8} \rceil - 1$
- $v \in \mathcal{V}_{sym}$ is fresh

Storing a value at an unallocated address violates memory safety.

`store` **on unallocated memory** (p: "`store ty` t`, ty* ad`", $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$, `ad` $\in \mathcal{V}_\mathcal{P}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{ERR} \quad \text{if}$$

- there is no $[\![v_1, v_2]\!] \in AL$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV_{u,n}(\mathtt{ad}) \wedge ad_{end} \leq v_2)$
- $ad_{end} = LV_{u,n}(\mathtt{ad}) + \lceil \frac{size(\mathtt{ty})}{8} \rceil - 1$

For other rules and for extensions of the rules to several LLVM functions and to arbitrary alignments, we refer to [14].

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2013-01 *	Fachgruppe Informatik: Annual Report 2013
2013-02	Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen
2013-03	Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM
2013-04	Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries
2013-05	Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013
2013-06	Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation
2013-07	André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung
2013-08	Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers
2013-10	Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata
2013-12	Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs
2013-13	Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators
2013-14	Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures
2013-16	Carsten Otto: Java Program Analysis by Symbolic Execution
2013-19	Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol

2013-20 Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models
2014-01 * Fachgruppe Informatik: Annual Report 2014
2014-02 Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software
2014-03 Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04 Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06 Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07 Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08 Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09 Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14 Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15 Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16 Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results
2015-01 * Fachgruppe Informatik: Annual Report 2015
2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"
2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon
2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization
2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models
2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus
2015-12 Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic

2015-13 Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2015-14 Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines

2016-01 * Fachgruppe Informatik: Annual Report 2016

2016-02 Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems

2016-03 Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs

* These reports are only available as a printed version.

Please contact biblio@informatik.rwth-aachen.de to obtain copies.