Termination and Complexity Analysis for 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. Moreover, we show that our approach can also be used to analyze the runtime complexity of bitvector programs. We implemented our contributions in the automated termination prover AProVE and evaluate its power by extensive experiments.

Keywords: termination analysis, bitvectors, symbolic execution, LLVM, runtime complexity

1. Introduction

In [27], we developed an approach for automated termination analysis of C programs with explicit pointer arithmetic, which we implemented in our tool AProVE [17, 21, 26]. AProVE won the termination category of the *International Competition on Software Verification (SV-COMP)*[1] at the *TACAS* conferences in 2015 and 2016. However, similar to the other termination tools at *SV-COMP*, up to now our approach was restricted to mathematical integers.

In general, this restriction is unsound: Consider the C functions `f` and `g` in Fig. 1, which increment a variable `j` as long as the loop condition holds. For `f`, one leaves the loop as soon as `j` exceeds the value of the parameter `x`. Thus, the function `f` does not terminate if `x` has the maximum value of its type.[2] But we

[☆]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,

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

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

Figure 1: C functions on bitvectors

can falsely prove termination if we treat `x` and `j` as mathematical integers. For `g`, the loop terminates as soon as the value of `j` becomes zero. So when considering mathematical integers, we would falsely conclude non-termination for positive initial values of `j`, although `g` always terminates due to the wrap-around for unsigned overflows.

In this paper, we adapt our approach for termination of C from [27] 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 [24]. 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 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. If the resulting ITS is terminating, then the original C resp. LLVM program terminates as well. In Sect. 5 we show that our transformation into ITSs can also be adapted in order to derive upper bounds on the program's runtime, i.e., our approach can be used for complexity analysis of bitvector programs as well. In Sect. 6, we compare our approach with related work and evaluate our corresponding implementation in AProVE.[3] Appendix A gives further formal details on separation logic and on the abstract states used for symbolic execution. Appendix B contains the proofs of the theorems.

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 mathematical integers from these graphs, and to use standard approaches for generating ranking functions in order to prove the termination or to analyze the complexity of these ITSs. So our contribution is a general technique to

most C implementations return the minimal signed integer as the result.

[3]Programs like `f` and `g` in Fig. 1 are often undesirable, since their termination behavior depends on overflows. However, there are also programs where overflows are intended. In such cases, only the results of verification techniques which handle bitvector semantics are meaningful. The most important class of such algorithms uses modular arithmetic, which can be implemented efficiently using unsigned integers and overflows. Our implementation in AProVE could also be used to prove the absence of overflows in general (although this is the not the main goal of our technique) and to detect programs whose termination behavior depends on overflows, cf. Sect. 6.

adapt byte-accurate symbolic execution to the handling of bitvectors, which can also be used for many other program analyses besides proving termination or complexity. The main characteristics of our adaption are:

(a) Handling Memory. In contrast to other approaches for bit-precise termination analysis, our rules for symbolic execution can also perform low-level memory management, including explicit pointer arithmetic.

(b) Representation with $\mathbb{Z}$. We represent the relation between bitvector variables by corresponding relations between integer variables, which allows us to use standard techniques and tools for SMT solving and for analyzing integer transition systems.

(c) Unsigned resp. Signed Representation. Based on a heuristic to classify program variables as "unsigned" or "signed", we represent information about their unsigned or signed value in the abstract states for symbolic execution. This simplifies the symbolic execution of instructions that differ for unsigned and signed integers. Note that LLVM does not provide the information whether a variable is signed or unsigned.

(d) Case Analysis vs. "Modulo". Due to the wrap-around behavior of C for overflows, representing bitvector relations by relations on mathematical integers can either be done by case analysis or by using "modulo" relations. For reasons of efficiency, we developed a hybrid approach which uses case analysis for instructions like addition and which uses "modulo" for operations like multiplication. To increase the precision of the resulting abstract states, we show how to infer information about the ranges of variables, even if these ranges are unions of disjoint intervals. For an efficient SMT reasoning during symbolic execution, we developed an approach to express such "disjunctive properties" by single inequalities.

Earlier Work. A preliminary version of parts of this paper was published in [20]. The present paper extends [20] as follows:

- We added many more details and explanations throughout the paper.
- In [20], symbolic execution rules were only given for a small subset of LLVM instructions. In contrast, we now present symbolic execution rules for all (interesting) LLVM instructions that are affected by the bitvector semantics:
  - In the new Sect. 3.1, we show how the rule for the `store` instruction is adapted in order to store unsigned or signed integer values.
  - In [20], Sect. 3.2 only contained the rules for the unsigned greater-than comparison, whereas we now also present the rules for the signed greater-than comparison. Moreover, we give the rules for signed addition which were missing in [20].

  - In [20], Sect. 3.3 only contained the rule for unsigned multiplication. We now also include rules for signed multiplication and for (signed or unsigned) division.
  - In Sect. 3.4, we now added the rule for unsigned `and`, as well as for `or` and `xor`. Moreover, while [20] only contained a rule for signed `trunc`, we now also present a rule for unsigned `trunc`. Finally, the rules for shift instructions in Sect. 3.4.3 are also new compared to [20].

- In [20] we only showed how to use our approach for termination analysis of bitvector programs. Now we extended our approach such that it can also be used to analyze the runtime complexity of bitvector programs. This extension is described in the new Sect. 5 as well as in new corresponding experiments in Sect. 6.

Limitations. To simplify the presentation and to concentrate on the issues related to bitvectors, we restrict ourselves to a single LLVM function without function calls 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 using the LLVM instruction `alloca`. See [27] for an extension of our approach to programs with several (non-recursive) LLVM functions, arbitrary alignment, and external functions like `malloc`. As discussed in [27], some LLVM concepts are not yet supported by our approach (e.g., `undef`, floating point values, vectors, recursion, and dynamic data structures that are realized as `struct` types). 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. We are currently working on a corresponding extension of our approach in order to handle recursive programs and to prove non-termination as well [21].

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[4] LLVM code in Fig. 2, the integer variable `j` has the type `i32`, since 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 LLVM code in detail when constructing the symbolic execution graph in Sect. 3.

[4] This LLVM program corresponds to the code obtained from `g` with the Clang compiler [8]. 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.

```
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 }
```

Figure 2: LLVM code for the function `g`

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 the values of symbolic variables, and two sets which describe memory 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 $\mathcal{V}_{sym} \cap \mathcal{V}_\mathcal{P} = \varnothing$. By using $\mathcal{V}_{sym}$ instead of concrete integers, we cannot only represent *concrete* states, where all symbolic variables $v \in \mathcal{V}_{sym}$ are constrained to concrete fixed numbers, but also *abstract* states, where v can stand for several possible values. Let $\mathcal{V}_{sym}(LV) \subseteq \mathcal{V}_{sym}$ be the set of all symbolic variables v where there exists some $\mathtt{x} \in \mathcal{V}_\mathcal{P}$ with $LV(\mathtt{x}) = v$.

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 simplify and to speed up SMT-based arithmetic reasoning.

The fourth component of a state is an *allocation list* AL. This list contains expressions of the form $[\![v_1, v_2]\!]$ for $v_1, v_2 \in \mathcal{V}_{sym}$, which indicate that $v_1 \leq v_2$ holds 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, $v_1 \hookrightarrow_{\texttt{i32},i} v_2$ 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). Def. 1 introduces our notion of (possibly abstract) LLVM states formally.

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* $PT \subseteq \{(v_1 \hookrightarrow_{\texttt{ty},i} v_2) \mid v_1, v_2 \in \mathcal{V}_{sym}, \texttt{ty}$ *is an* LLVM *type*, $i \in \{u, s\}\}$. *In addition, there is a state ERR for undefined behavior. For a state* $a = (p, LV, KB, AL, PT)$, *let* $\mathcal{V}_{sym}(a)$ *consist of* $\mathcal{V}_{sym}(LV)$ *and of all symbolic variables occurring in KB, AL, or PT.*

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

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

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

In contrast to [27], we partition the set of program variables $\mathcal{V}_\mathcal{P}$ into two disjoint sets $\mathcal{U}_\mathcal{P}$ and $\mathcal{S}_\mathcal{P}$ (i.e., $\mathcal{V}_\mathcal{P} = \mathcal{U}_\mathcal{P} \uplus \mathcal{S}_\mathcal{P}$). If $\texttt{x} \in \mathcal{U}_\mathcal{P}$ (resp. $\texttt{x} \in \mathcal{S}_\mathcal{P}$), then $LV(\texttt{x})$ is `x`'s value as an unsigned (resp. signed) integer. As will be shown in Sect. 3, this is advantageous when formulating rules to execute LLVM instructions like `icmp ugt` and `icmp sgt` (for the integer comparisons "unsigned greater than" and "signed greater than"). The reason is that the types of LLVM 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`) or in another unsigned operation (e.g., `udiv` or `urem` for "unsigned division" or "unsigned remainder"),
- `x` occurs in a sign neutral comparison (`icmp eq` or `icmp ne`) or in a `phi` or `select` instruction together with another variable $\texttt{y} \in \mathcal{U}_\mathcal{P}$, 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 in a conversion instruction with another $\texttt{y} \in \mathcal{U}_\mathcal{P}$,

- `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 in a 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 the program $\mathcal{P}$, all occurring program variables of type `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 really 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. For example, $\langle a \rangle_{FO}$ contains the constraint that $v_1 \leq v_2$ holds for each pair $[\![v_1, v_2]\!]$ in a's allocation list and that all addresses are positive numbers. Moreover, $\langle a \rangle_{FO}$ expresses that two values at the same address must be equal and two addresses must be different if they point to different (un)signed values of the same type. In addition, $\langle a \rangle_{FO}$ 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 $\texttt{ty}$ (e.g., $size(\texttt{i}n) = n$ and $size(\texttt{ty*}) = 32$ (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*[5]

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

Concrete states are abstract states of a particular form which determine the values of variables and the contents of the memory *uniquely*. To enforce a uniform representation, in concrete states we represent memory data byte-wise and only allow statements of the form $(w_1 \hookrightarrow_{\texttt{ty},i} w_2)$ in PT where $\texttt{ty} = \texttt{i8}$ and $i = u$. In addition, concrete states (p, LV, KB, AL, PT) must be *well formed*, i.e., for every $(w_1 \hookrightarrow_{\texttt{ty},i} w_2) \in PT$, there is an allocated area $[\![v_1, v_2]\!] \in AL$ 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 $(w_1 \hookrightarrow_{\texttt{i8},u} w_2) \in PT$ 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 $(w_1 \hookrightarrow_{\texttt{ty},i} w_2) \in PT$ for $\texttt{ty} \neq \texttt{i8}$ or $i = s$.*

To define the semantics of an abstract state a, in [27] we also introduced a *separation logic* formula $\langle a \rangle_{SL}$ which extends $\langle a \rangle_{FO}$ by detailed information about the memory (i.e., about AL and PT). (In Appendix A we recapitulate the formal definition of $\langle a \rangle_{SL}$ and the formal semantics of the fragment of separation logic that we consider here.) For these semantics, we use *interpretations* of the form (as, mem). Here, $as : \mathcal{V}_{\mathcal{P}} \to \mathbb{Z}$ is an *assignment* of the program variables, where for $\texttt{x} \in \mathcal{V}_{\mathcal{P}}$ of type $\texttt{ty}$, we have $as(\texttt{x}) \in [0, \mathsf{umax}_{size(\texttt{ty})}]$ if $\texttt{x} \in \mathcal{U}_{\mathcal{P}}$ and $as(\texttt{x}) \in [\mathsf{smin}_{size(\texttt{ty})}, \mathsf{smax}_{size(\texttt{ty})}]$ if $\texttt{x} \in \mathcal{S}_{\mathcal{P}}$. The partial function $mem : \mathbb{N}_{>0} \rightharpoonup$

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

$\{0, \ldots, \mathsf{umax}_8\}$ with finite domain describes the *memory* contents at allocated addresses (as unsigned integers). Here, we use $\mathbb{N}_{>0} = \mathbb{N} \setminus \{0\}$. As usual, "$\models \varphi$" means that $(as, mem) \models \varphi$ holds for any interpretation (as, mem). Then we have $\models \langle a \rangle_{SL} \Rightarrow \langle a \rangle_{FO}$ for any abstract state a. So $\langle a \rangle_{FO}$ is a weakened version of $\langle a \rangle_{SL}$ which we use to construct symbolic execution graphs. This allows us to apply 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 we define that an abstract state a *represents* all those 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 $\sigma : \mathcal{V}_{sym} \to \mathbb{Z}$. Thus, σ does not instantiate the program variables $\mathcal{V}_{\mathcal{P}}$. Instantiations are extended to formulas in the usual way, i.e., $\sigma(\varphi)$ instantiates every free occurrence of $v \in \mathcal{V}_{sym}$ in φ by $\sigma(v)$.

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 $(w_1 \hookrightarrow_{\mathtt{i8},u} w_2) \in PT^c$ such that $\models \langle c \rangle_{FO} \Rightarrow w_1 = n$. In this case, let $\models \langle c \rangle_{FO} \Rightarrow w_2 = k$, where $k \in [0, \mathsf{umax}_8]$. Then we have $mem^c(n) = k$.*

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 possible executions of a program. To this end, we define operations to convert any integer expression t into an unsigned resp. signed n-bit integer:[6]

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

The correctness of uns_n is obvious. By Thm. 5, sig_n is correct as well, i.e., $\mathsf{sig}_n(t)$ is indeed in the range $[\mathsf{smin}_n, \mathsf{smax}_n]$ of signed n-bit integers and t and $\mathsf{sig}_n(t)$ are the same modulo 2^n.

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 $t \bmod 2^n = \mathsf{sig}_n(t) \bmod 2^n$.*

[6] 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 $t = k \cdot n + m$.

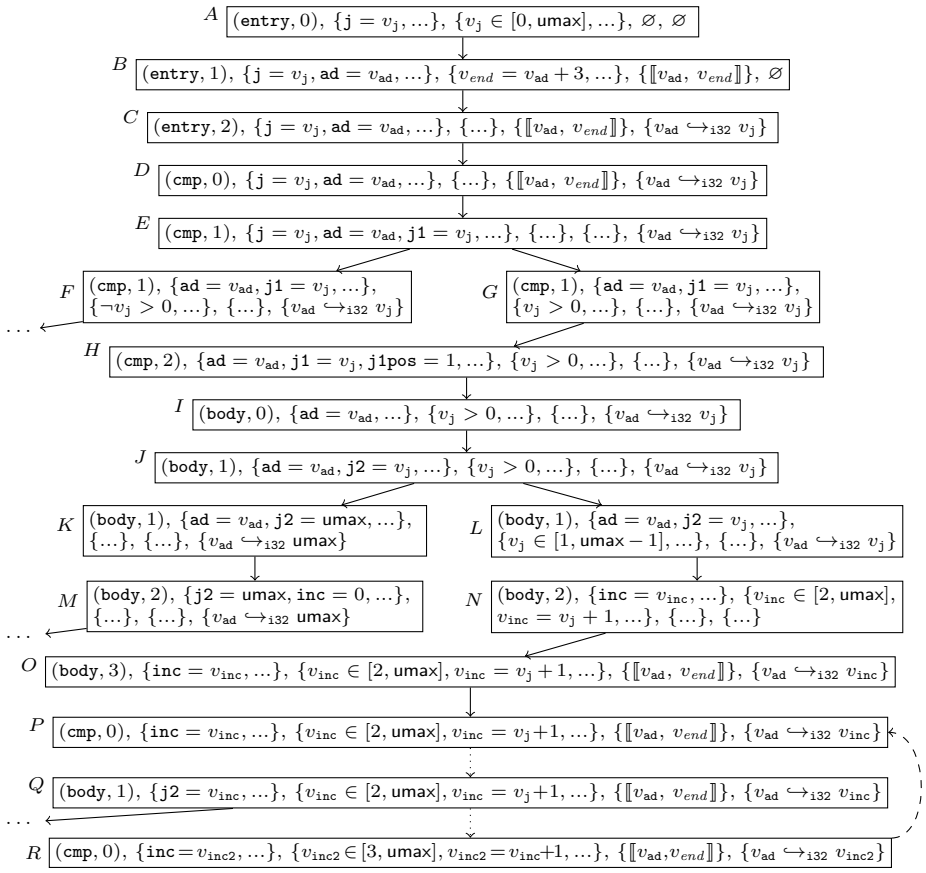

Figure 3: Symbolic execution graph for the function `g`

To ease the formalization, we extend LV such that it can also be applied to concrete integers. To this end, we use the functions $LV_{u,n}$, $LV_{s,n} : \mathcal{V}_{\mathcal{P}} \uplus \mathbb{Z} \to \mathcal{V}_{sym} \uplus \mathbb{Z}$, 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 (rules for other LLVM instructions can be found in [27]). After adapting the rule for the `store` instruction in Sect. 3.1, we show how to handle overflows by appropriate case analyses (Sect. 3.2) or by introducing "modulo" relations (Sect. 3.3). Finally, Sect. 3.4 presents rules for bitwise binary and conversion instructions.

3.1. Storing Unsigned or Signed Integer Values

We start with the initial state that one wants to analyze for termination, e.g., with the abstract state A where `j` has an unknown value. In the symbolic execution graph for `g` in Fig. 3, we abbreviated parts by "..." and wrote $\hookrightarrow_{\texttt{i32}}$ and umax instead of $\hookrightarrow_{\texttt{i32},u}$ and umax_{32}. To ease readability, we replaced some symbolic variables by their values (e.g., instead of $\texttt{j1pos} = z$ in LV and $z = 1$ in KB we directly wrote $\texttt{j1pos} = 1$ in LV). Moreover, we 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` starts with allocating a memory area $[\![v_{\texttt{ad}}, v_{end}]\!]$ (cf. State B) and then it stores the value $v_{\texttt{j}}$ of the parameter `j` at the address `ad`. The following rule shows how to evaluate the `store` instruction symbolically, i.e., it is used for the step from State B to C. This rule is affected by the change to the bitvector semantics, because it has to take into account which values should be stored as an unsigned or as a signed integer, respectively. For this reason, we present two corresponding versions of the symbolic execution rule below.

Let "$p\colon ins$" denote that ins is the instruction at the program position p. We now handle the case p: "`store ty` t`, ty* ad`", i.e., the integer value t of type `ty` should be stored at the address `ad` in the memory. In our rules, 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 $\langle a \rangle_{FO}$. As each memory cell stores one byte, in the `store` rule we first have to check whether the addresses $\texttt{ad}, \ldots, ad_{end}$ are allocated, i.e., whether there is a $[\![v_1, v_2]\!] \in AL$ such that $\langle a \rangle \Rightarrow (v_1 \leq LV_{u,n}(\texttt{ad}) \ \wedge \ ad_{end} \leq v_2)$ is valid. Here, $n = size(\texttt{ty*})$ is the bit-size of addresses and as addresses are stored as unsigned integers, the value of `ad` is $LV_{u,n}(\texttt{ad})$. The value ad_{end} can be computed from $LV_{u,n}(\texttt{ad})$ by taking into account how many bytes are needed to store a value of type `ty`. After executing the instruction, we reach a new state where the previous position $p = (\texttt{b}, k)$ is updated to the position $p^+ = (\texttt{b}, k+1)$ of the next instruction in the same block.

In this new state we store a new value v at the address $LV_{u,n}(\texttt{ad})$. Whether this value corresponds to the unsigned or the signed value of t is decided according to our heuristic from Sect. 2. If $t \in \mathcal{U}_{\mathcal{P}}$ or if t is an integer and no value from $\mathcal{S}_{\mathcal{P}}$ is loaded from or stored to the address `ad`, then we extend PT by the information that `ad` now points to the unsigned value of t and add $v = LV_{u,size(\texttt{ty})}$ to the knowledge base KB. Otherwise, we proceed analogously with the signed value. All information in PT that is not influenced by this change is kept. 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` $\in \mathcal{V}_\mathcal{P}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV,\ KB \cup \{v = LV_{u,size(\texttt{ty})}\},\ AL,\ PT' \cup \{LV_{u,n}(\texttt{ad}) \hookrightarrow_{\texttt{ty},u} v\})} \quad \text{if}$$

- $t \in \mathcal{U}_\mathcal{P}$, 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 $\models \langle a \rangle \Rightarrow (v_1 \leq LV_{u,n}(\texttt{ad}) \wedge ad_{end} \leq v_2)$
- $n = size(\texttt{ty*})$
- $ad_{end} = LV_{u,n}(\texttt{ad}) + \lceil \frac{size(\texttt{ty})}{8} \rceil - 1$ and $w_{1\,end} = w_1 + \lceil \frac{size(\texttt{sy})}{8} \rceil - 1$ for all w_1
- $PT' = \{(w_1 \hookrightarrow_{\texttt{sy},i} w_2) \in PT \mid \models \langle a \rangle \Rightarrow ([\![LV_{u,n}(\texttt{ad}), ad_{end}]\!] \perp [\![w_1,\ w_{1\,end}]\!])\}$
- $v \in \mathcal{V}_{sym}$ is fresh

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

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

- $t \in \mathcal{S}_\mathcal{P}$, 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 $\models \langle a \rangle \Rightarrow (v_1 \leq LV_{u,n}(\texttt{ad}) \wedge ad_{end} \leq v_2)$
- $n = size(\texttt{ty*})$
- $ad_{end} = LV_{u,n}(\texttt{ad}) + \lceil \frac{size(\texttt{ty})}{8} \rceil - 1$ and $w_{1\,end} = w_1 + \lceil \frac{size(\texttt{sy})}{8} \rceil - 1$ for all w_1
- $PT' = \{(w_1 \hookrightarrow_{\texttt{sy},i} w_2) \in PT \mid \models \langle a \rangle \Rightarrow ([\![LV_{u,n}(\texttt{ad}), ad_{end}]\!] \perp [\![w_1,\ w_{1\,end}]\!])\}$
- $v \in \mathcal{V}_{sym}$ is fresh

In our example, $\texttt{j} \in \mathcal{U}_\mathcal{P}$ and thus, the "unsigned `store`" rule is used to evaluate State B. We have $[\![v_{\texttt{ad}},\ v_{end}]\!] \in AL$, and $\langle B \rangle$ implies that $LV_{u,32}(\texttt{ad}) = v_{\texttt{ad}}$ is in this allocated area. Instead of adding $v_{\texttt{ad}} \hookrightarrow_{\texttt{i32},u} v$ to PT and $v = v_{\texttt{j}}$ to KB, in Fig. 3 we directly extended PT by $v_{\texttt{ad}} \hookrightarrow_{\texttt{i32},u} v_{\texttt{j}}$ to ease readability.

Storing a value at an unallocated address violates memory safety and thus, in this case we use a symbolic execution rule which reaches the *ERR* state.

`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 $\models \langle a \rangle \Rightarrow (v_1 \leq LV_{u,n}(\texttt{ad}) \wedge ad_{end} \leq v_2)$
- $n = size(\texttt{ty*})$
- $ad_{end} = LV_{u,n}(\texttt{ad}) + \lceil \frac{size(\texttt{ty})}{8} \rceil - 1$

3.2. Handling Bitvector Operations by Case Analysis

After executing the `store` instruction, our program `br`anches to the block `cmp` for the loop comparison. Now the value $v_{\texttt{j}}$ (stored at the address `ad`) is loaded to

the program variable `j1`. Next, for the integer comparison instruction in State E we have to check whether `j1`'s value in unsigned interpretation is greater than 0 (`icmp ugt`). In the symbolic execution rule for this instruction, we write $LV[\mathtt{x} := v]$ for the function where $(LV[\mathtt{x} := v])(\mathtt{x}) = v$ and where $(LV[\mathtt{x} := v])(\mathtt{y}) = LV(\mathtt{y})$ for all $\mathtt{y} \neq \mathtt{x}$.

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

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)}$$ if $v \in \mathcal{V}_{sym}$ 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 "$v = 0$"

However, in our example the value of $LV_{u,32}(\mathtt{j1}) = LV(\mathtt{j1}) = v_\mathtt{j}$ is unknown. Thus, we first have to *refine* State E to the states F and G in such a way 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** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)}$$ 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 the *signed* "greater than" comparison (`sgt`) are similar to the above rules, but they use $LV_{s,size(\mathtt{ty})}$ instead of $LV_{u,size(\mathtt{ty})}$.

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

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)}$$ if $v \in \mathcal{V}_{sym}$ 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 "$v = 0$"

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

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)}$$ 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 rules for `icmp` (`uge`|`ult`|`ule`|`sge`|`slt`|`sle`|`eq`|`ne`) are analogous.

When evaluating these `icmp` instructions symbolically, we benefit from the fact that if a program variable `y` is compared by `ugt` and $\mathtt{y} \in \mathcal{U}_\mathcal{P}$, then the symbolic variable $LV(\mathtt{y})$ already represents `y`'s value as an unsigned integer,

which makes the comparison very simple. (Similarly, $LV(\mathtt{y})$ represents a signed integer if `y` is compared by `sgt`.) In contrast, if LV represented the value of *all* program variables as signed integers, then in the above case analysis for the `icmp ugt` refinement we would have to consider more cases, which would result in a significantly larger graph (and thus, in a less efficient approach).[7]

In our example, if $\neg v_{\mathtt{j}} > 0$ (State F), then we `ret`urn from the function. If $v_{\mathtt{j}} > 0$ (State G), then the conditional `br`anch instruction leads us to the block `body`, which corresponds to the body of the `while`-loop. In the step from I to J, again the value $v_{\mathtt{j}}$ stored at the 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.

Therefore, we have to adapt the `add` rule for bitvectors. We only evaluate the addition operation if KB contains enough information to decide whether an overflow occurs or not. Otherwise, a case analysis needs to be performed, i.e., we refine the abstract state in order to distinguish all states where an overflow occurs from those where no overflow occurs.

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

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

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, `j2` has a value smaller than umax_{32} such that an 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.

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

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

- $\mathtt{x} \in \mathcal{S}_\mathcal{P}$
- 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 $\not\models \langle a \rangle \Rightarrow \varphi_3$

[7] 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" yields *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 $LV_{s,size(\mathtt{ty})}(t_1) \leq LV_{s,size(\mathtt{ty})}(t_2)$.

Now we define rules for evaluating `add`. If no overflow can occur, then the result is the addition of the operators. Thus, State L evaluates to N, where the result value $v_{\texttt{inc}}$ may be any value in $[2, \mathsf{umax}_{32}]$ and we know that $v_{\texttt{inc}} = v_{\texttt{j}} + 1$. Below, we give the rules for both the unsigned and the signed case.

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

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

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

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 $v_{\texttt{inc}} = \mathsf{umax}_{32} + 1 - 2^{32} = 0$.

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

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

$\texttt{x} \in \mathcal{U}_\mathcal{P}$, $v \in \mathcal{V}_{sym}$ 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 $\texttt{x} \in \mathcal{S}_\mathcal{P}$. However, a potential signed overflow that is flagged with `nsw` leads to *ERR*.

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

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\texttt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)}$$ if $\texttt{x} \in \mathcal{S}_\mathcal{P}$, $v \in \mathcal{V}_{sym}$ 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: "`x = add nsw in` t_1, t_2", $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

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

$\mathtt{x} \in \mathcal{S}_\mathcal{P}$ 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 M, the execution will end after some more steps, as the value used for the comparison in the loop condition will be 0 the next time we reach the corresponding program position ($\mathtt{cmp}, 1$).

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 execution steps. State 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 [27] are satisfied. Again, let a denote 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 $(\mu(v_1) \hookrightarrow_{\mathtt{ty},i} \mu(v_2)) \in PT$

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 in the symbolic execution graph, which would not correspond to any computation. See [27] for a heuristic to compute suitable generalizations automatically.

In our graph in Fig. 3, 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. Therefore, we can conclude the graph construction with a (dashed) *generalization edge* from R to P. We say that 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 [27], any LLVM evaluation

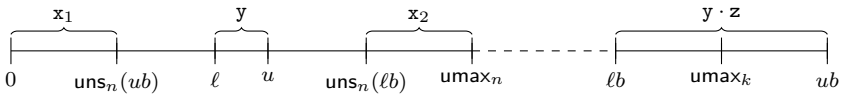

Figure 4: Multiplication of unsigned integers

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).

3.3. 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.[8] 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. In order to apply standard SMT solvers for expressions that contain "modulo", any equality "$t = m \bmod n$" 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 in` 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. 4) 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 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. Thus, $LV(\mathtt{x})$ can be any unsigned n-bit integer and we cannot infer any more specific information on its value. Otherwise, we check whether $\mathsf{uns}_n(\ell b) \leq \mathsf{uns}_n(ub)$ holds. In this case, we add the information "$LV(\mathtt{x}) \in [\mathsf{uns}_n(\ell b), \mathsf{uns}_n(ub)]$" to KB. Finally, if the size of the interval $[\ell b, ub]$ is $< 2^n$ but

[8] 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.

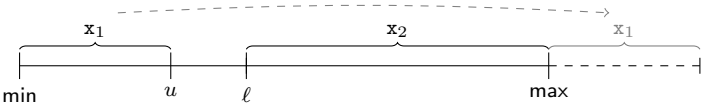

Figure 5: Expressing unions of intervals

$\mathsf{uns}_n(\ell b) > \mathsf{uns}_n(ub)$, 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. 4. 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 now 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 just a single inequality. To this end, we subtract ℓ so that the second subinterval $[\ell, \mathsf{max}]$ ($\mathtt{x}_2$ in Fig. 5) starts with 0. Then we apply "mod 2^n" (this results in moving the first subinterval $\mathtt{x}_1$ as indicated by the dashed arrow in Fig. 5). 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 $\mathsf{inBounds}(v, 0, \mathsf{uns}_n(ub), \mathsf{uns}_n(\ell b), \mathsf{umax}_n)$.

For signed integers, we use the operation sig_n to convert any integer number into a corresponding signed n-bit integer. So for the signed instruction "`x = mul in` t_1`,` t_2", we know that $LV(\mathtt{x})$ gets the value $\mathsf{sig}_n(LV_{s,n}(t_1) \cdot LV_{s,n}(t_2))$. Moreover, this can be simplified to $LV_{s,n}(t_1) \cdot LV_{s,n}(t_2)$ if $LV_{s,n}(t_1) \cdot LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n]$.

As for unsigned multiplication, we again extend KB by additional information on the possible values of the result of a multiplication "`x = mul in` t_1`,` t_2". If $LV_{s,n}(t_1) \in [\ell_1, u_1]$ and $LV_{s,n}(t_2) \in [\ell_2, u_2]$ for numbers $\ell_1, \ell_2, u_1, u_2 \in \mathbb{Z}$, then we can compute $\ell b = \min\{x_1 \cdot x_2 \mid x_1 \in [\ell_1, u_1], x_2 \in [\ell_2, u_2]\}$ and $ub = \max\{x_1 \cdot x_2 \mid x_1 \in [\ell_1, u_1], x_2 \in [\ell_2, u_2]\}$, and obtain $LV_{s,n}(t_1) \cdot LV_{s,n}(t_2) \in$

$[\ell b, ub]$. However, our goal is to infer information on the possible value of $\mathsf{sig}_n(LV_{s,n}(t_1) \cdot LV_{s,n}(t_2))$.

Similar to the unsigned case, we compute the size of the interval $[\ell b, ub]$. If $ub - \ell b + 1 < 2^n$ and $\mathsf{sig}_n(\ell b) \leq \mathsf{sig}_n(ub)$, we add the information that $LV(\mathtt{x}) \in [\mathsf{sig}_n(\ell b), \mathsf{sig}_n(ub)]$ holds. If $ub - \ell b + 1 < 2^n$ and $\mathsf{sig}_n(\ell b) > \mathsf{sig}_n(ub)$, then $LV(\mathtt{x}) \in [\mathsf{smin}_n, \mathsf{sig}_n(ub)] \cup [\mathsf{sig}_n(\ell b), \mathsf{smax}_n]$. Again we use Thm. 6 to encode this as a single inequality without disjunction.

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

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

- 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 $\mathsf{inBounds}(v, \mathsf{smin}_n, \mathsf{sig}_n(ub), \mathsf{sig}_n(\ell b), \mathsf{smax}_n)$.

Similar to addition, we have corresponding rules for signed multiplication with the flag "`nsw`". They correspond to the above rule if $\models \langle a \rangle \Rightarrow LV_{s,n}(t_1) \cdot LV_{s,n}(t_2) \in [\mathsf{smin}_n, \mathsf{smax}_n]$ and otherwise, we reach the state *ERR*.

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

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

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

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

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

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 in` t_1, t_2" with $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

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

- $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_2) > 0)$
- 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 in` 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 in` t_1, t_2" with $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

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

- 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 in` 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`.

3.4. Handling Bitwise Operations

We now handle bitwise binary LLVM operations like logical operators (Sect. 3.4.1), conversion instructions (Sect. 3.4.2), and shifts (Sect. 3.4.3). Again, our aim is to infer knowledge about the range of the result of the operation. Representing this information in the symbolic execution rules helps to improve the precision of our analysis.

3.4.1. Logical Operations

The operation "`and`" computes bitwise logical conjunction. For instance, the conjunction of 3 (0 1 1) and 5 (1 0 1) is 1 (0 0 1). So if "$\texttt{x = and } in\ t_1, t_2$" and $\texttt{x} \in \mathcal{U}_\mathcal{P}$, then $LV(\texttt{x}) \leq LV_{u,n}(t_1)$ and $LV(\texttt{x}) \leq LV_{u,n}(t_2)$, since a "1" on a position of the bitvector always 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 ($1\,1 \ldots 1\,1$) and -2 ($1\,1 \ldots 1\,0$) is -2. The conjunction of a negative and a positive signed integer is at most as large as the positive integer. This results in the following two rules.

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

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

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: "$\texttt{x = and } in\ t_1, t_2$" **with** $\texttt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

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

- $\ell_1, \ell_2, u_1, u_2 \in \mathbb{Z}$ 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 an analogous way, we also obtain two rules for logical disjunction, depending on whether the result variable is considered to be unsigned or signed.

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

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

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 in` t_1, t_2" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)}$$ if $\mathtt{x} \in \mathcal{S}_\mathcal{P}$, $v \in \mathcal{V}_{sym}$ 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 "$v \geq \min(\ell_1, \ell_2)$".

We also have a rule for exclusive disjunction. Here, we only infer the information whether the resulting value is positive or negative when regarding it as a signed integer.

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

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)}$$ if $v \in \mathcal{V}_{sym}$ 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*".

3.4.2. Conversion Instructions

Next we adapt the rules for conversion instructions (i.e., extension and truncation). One distinguishes between *zero extension* (`zext`) and *sign extension* (`sext`). For the sign extension, the sign bit (i.e., the most significant bit) is copied to all extension bits until the desired bit size is reached. In contrast, for the zero extension 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 in` t `to i`m" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$, $n < m$)

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

either p : "`x = sext in` t `to i`m", $\mathtt{x} \in \mathcal{S}_\mathcal{P}$, and φ is "$v = LV_{s,n}(t)$"
or p : "`x = zext in` 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. Here, the most interesting cases occur when this changes the sign of the operand value when regarding signed integers. As illustrated below, truncating the 16-bit signed integer 653 to a 8-bit signed integer yields -115.

$$0\,0\,0\,0\,0\,0\,1\,0 \,|\, 1\,0\,0\,0\,1\,1\,0\,1$$

Indeed, if $n = 8$, then we have $\mathsf{sig}_8(653) = -115$. So similar to the rules for multiplication, we can again use the operations uns_n resp. sig_n and $\mathsf{inBounds}$ to express our knowledge about the result of the truncation.

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

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

- 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: **"**`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 $\mathsf{inBounds}(v, \mathsf{smin}_n, \mathsf{sig}_n(u), \mathsf{sig}_n(\ell), \mathsf{smax}_n)$.

3.4.3. Shift Instructions

Finally, we present rules for the shift instructions of LLVM. We start with the logical right-shift (`lshr`). An operand of type `i`n may be shifted by at most $n - 1$ bits. After the shift, the (new) most significant bits are filled with zeros.

`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` in t_1, t_2", `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 (`ashr`), 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 in 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` in t_1, t_2" **with** `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 "$v \cdot 2^{LV_{u,n}(t_2)} \leq LV_{s,n}(t_1) \ \wedge\ (v+1) \cdot 2^{LV_{u,n}(t_2)} > LV_{s,n}(t_1)$"

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

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

The corresponding rules for the left-shift instruction `shl` are similar.

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

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)}$$ if $\mathtt{x} \in \mathcal{S}_\mathcal{P}$, $v \in \mathcal{V}_{sym}$ 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` in t_1, t_2", $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)}$$ if $\mathtt{x} \in \mathcal{S}_\mathcal{P}$, $v \in \mathcal{V}_{sym}$ 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` in t_1, t_2", $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

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

4. From Symbolic Execution Graphs to Integer Transition Systems

After the construction of the symbolic execution graph has been completed, we automatically 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 (program locations) correspond to the abstract states and whose edges are *transitions* that are labeled with conditions required for their application. Let $\mathcal{V} \subseteq \mathcal{V}_{sym}$ be the finite set of all symbolic variables occurring in the symbolic execution graph. Then formally, an *ITS transition* is a tuple $(a, CON, \overline{a})$ where $a, \overline{a}$ are abstract states and the *condition* $CON \subseteq QF\_IA(\mathcal{V} \uplus \mathcal{V}')$ is a set of quantifier-free formulas over the variables $\mathcal{V} \uplus \mathcal{V}'$. Here, $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ represents the values of the variables *after* the transition. An *ITS state* (a, σ) consists of an abstract state a and a concrete instantiation $\sigma : \mathcal{V} \to \mathbb{Z}$. For any such σ, let $\sigma' : \mathcal{V}' \to \mathbb{Z}$ with $\sigma'(v') = \sigma(v)$. Given an ITS $\mathcal{I}$, (a, σ) *evaluates* to $(\overline{a}, \overline{\sigma})$ (denoted "$(a, \sigma) \to_\mathcal{I} (\overline{a}, \overline{\sigma})$") iff $\mathcal{I}$ has a transition $(a, CON, \overline{a})$ with $\models (\sigma \cup \overline{\sigma}')\,(CON)$. Here, $(\sigma \cup \overline{\sigma}')$ is the instantiation of the variables $\mathcal{V} \uplus \mathcal{V}'$ which behaves like σ on $\mathcal{V}$ and like $\overline{\sigma}$ on $\mathcal{V}'$, i.e., we have $(\sigma \cup \overline{\sigma}')(v) = \sigma(v)$ and $(\sigma \cup \overline{\sigma}')(v') = \overline{\sigma}'(v') = \overline{\sigma}(v)$ for all $v \in \mathcal{V}$. An ITS $\mathcal{I}$ is *terminating* iff $\to_\mathcal{I}$ is well founded. To convert symbolic execution graphs to ITSs, we use the definition from [27].

Definition 7 (ITS from Symbolic Execution Graph). *Let $\mathcal{G}$ be a symbolic execution graph. Then the* corresponding integer transition system *$\mathcal{I}_\mathcal{G}$ has one transition for each edge in $\mathcal{G}$:*

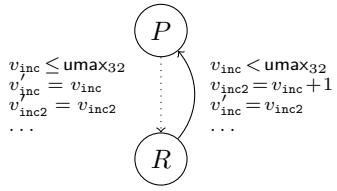

Figure 6: ITS for function g

- *If the edge from a to $\overline{a}$ is* not *a generalization edge, then $\mathcal{I}_\mathcal{G}$ has a transition from a to $\overline{a}$ with the condition $\langle a \rangle \cup \{v' = v \mid v \in \mathcal{V}_{sym}(a)\}$.*
- *If there is a generalization edge from a to $\overline{a}$ with the instantiation μ, then $\mathcal{I}_\mathcal{G}$ has a transition from a to $\overline{a}$ with the condition $\langle a \rangle \cup \{v' = \mu(v) \mid v \in \mathcal{V}_{sym}(\overline{a})\}$.*

The only cycle of the symbolic execution graph in Fig. 3 is the one leading from P to R and via the only generalization edge back to P. The resulting ITS is shown in Fig. 6. The values of the variables do not change in transitions that correspond to evaluation edges of the symbolic execution graph. So the edges from P to R in the symbolic execution graph yield transitions whose conditions contain $v'_{\mathtt{inc}} = v_{\mathtt{inc}}$ and $v'_{\mathtt{inc2}} = v_{\mathtt{inc2}}$. 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 on the path from P to R. Thus, $v_{\mathtt{inc}}$ is increased by 1 in the transition that corresponds to the generalization edge from R to P, and it remains the same in all other transitions of the cycle. Since the transitions are only executed as long as $v_{\mathtt{inc}} \leq \mathsf{umax}_{32}$ holds, termination of the resulting ITS can easily be proved automatically (by standard termination tools for ITSs over mathematical integers).

Recall that the bitvector arithmetic is covered by the rules to construct the symbolic execution graph, whereas the variables in the symbolic execution graph and in the resulting ITS range over mathematical integers $\mathbb{Z}$. Therefore, the following theorem from [27] also directly holds for the adaption of our approach to bitvectors. The theorem states that if there is an infinite LLVM computation starting with a concrete state that is represented in the symbolic execution graph, then the ITS resulting from the graph is not terminating. In other words, termination of the ITS implies termination of the analyzed LLVM program.

Theorem 8 (Termination of LLVM Programs). *Let $\mathcal{P}$ be an* LLVM *program with a complete symbolic execution graph $\mathcal{G}$ and let $\mathcal{I}_\mathcal{G}$ be the ITS corresponding to $\mathcal{G}$. If $\mathcal{I}_\mathcal{G}$ terminates, then $\mathcal{P}$ also terminates for all concrete states represented by the states in $\mathcal{G}$.*

5. Finding Upper Runtime Complexity Bounds

Often, one is not only interested in proving termination, but the runtime of the program is crucial. We now show that our new approach for symbolic execution of programs with bitvector arithmetic also allows us to analyze their runtime complexity. In order to infer runtime bounds instead of proving termination, one only has to adapt our technique to transform the symbolic execution graph into an ITS. While there exists a wealth of recent techniques and tools for automated complexity analysis of programs on mathematical integers (e.g., [1, 2, 4, 5, 6, 15, 16, 18, 22, 25]), to our knowledge our approach is the first which allows us to use these tools to analyze the runtime of bitvector programs automatically.

Our approach for runtime complexity analysis mainly succeeds on arithmetic programs. To analyze programs whose runtime depends on the memory, one would have to extend the abstraction we used in our symbolic execution since then the abstract states would also have to contain information on the sizes of the allocated memory areas. Note that for a terminating arithmetic program $\mathcal{P}$ with m instructions and k local variables $\mathcal{V}_\mathcal{P} = \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ of types $\mathtt{i}n_1, \ldots, \mathtt{i}n_k$, the runtime is bounded by $m \cdot \prod_{j=1}^{k} 2^{n_j}$, which is the number of possible concrete program states. The reason is that at each program position, every variable $\mathtt{x}_j$ may be assigned any value of its type (whose range is 2^{n_j}). Whenever a program state is visited twice, the program must be nonterminating. So since the state space is finite, every terminating arithmetic bitvector program has constant complexity. Thus, for arithmetic programs on bitvectors, asymptotic complexity is meaningless since all programs have a runtime in $\mathcal{O}(1)$.

Therefore, our goal is to infer concrete (non-asymptotic) bounds which are smaller than the maximum bound $m \cdot \prod_{j=1}^{k} 2^{n_j}$. In particular, we aim to find bounds that depend on the program's input parameters, because such bounds are usually more interesting than a huge constant that depends on the sizes of the types $\mathtt{i}n$. We developed the following adaptions of our approach for termination analysis in order to find runtime bounds for bitvector programs:

(1) For the runtime of a program, we count every execution step, which makes it necessary to extract the ITS from the whole graph and not only from its cycles. Moreover, this is required to infer correct runtime bounds for subsequent cycles. The reason is that the first cycle might increase values which are used afterwards when entering the next cycle.

(2) The initial abstract state a_0 of the symbolic execution graph is also considered to be the initial node of the ITS. So only evaluation sequences of the ITS that start with ITS states of the form (a_0, σ) have to be considered. Then the goal is to find a bound on the length of the ITS evaluations that depends only on the values $\sigma(v)$ for $v \in \mathcal{V}_{sym}(a_0)$.

(3) For an efficient analysis, we always simplify the transitions of the ITS by filtering away variables that do not influence the termination behavior and by iteratively compressing several transitions into one, cf. [17]. This is

unproblematic for termination analysis, but the compression of transitions would distort a concrete complexity result if several evaluation steps are counted as one. Therefore, we now assign a weight to each transition which over-approximates the number of evaluation steps that are represented by this transition. Refinement and generalization edges are transformed to transitions with weight 0.

(4) Since all handling of bitvector arithmetic is done during the symbolic execution, we generate ITSs over mathematical integers. Hence, their complexity can be analyzed by existing tools for such ITSs. In our implementation, we use the tool KoAT that we developed in earlier work [5] in parallel with the tool CoFloCo [15, 16]. Such back-end tools can then find an upper runtime bound for the extracted ITS, which is also a valid bound for the original LLVM program.

Note that up to now, such complexity tools have not been used to analyze the complexity of bitvector programs. In particular, some of these tools are targeted towards the inference of small asymptotic bounds (i.e., for a program with constant runtime, they would rather infer a huge constant bound than a linear bound that depends on the program's input parameters).

To facilitate the deduction of a bound depending on the program's parameters and to obtain more informative bounds, we therefore perform the following modification of the ITS that is generated from the symbolic execution graph. During the graph construction, we now keep track of all constants that originate from the size bounds of a variable's type. When the ITS is extracted, these constants are transformed to terms containing the respective size bounds as variables instead of constants. For example, the constant umax_{32} in State K of Fig. 3 is translated to a variable $b_{\mathsf{umax}_{32}}$. Similarly, the expression $\mathsf{umax}_{32} - 1$ in State L is translated to the term $b_{\mathsf{umax}_{32}} - 1$. Consequently, we now also have $b_{\mathsf{umax}_{32}} \in \mathcal{V}_{sym}(a)$ for all abstract states a of the graph.

Thus, instead of Fig. 6, we now obtain the weighted ITS in Fig. 7 from `g`'s symbolic execution graph in Fig. 3. Note that 14 instructions are executed from State A to State Q. The loop (i.e., the blocks `cmp` and `body`) contains 7 instructions. To evaluate State Q symbolically, we use an "unsigned `add` refinement" and if $v_{\texttt{inc}} = \mathsf{umax}_{32}$ holds, then one exits the loop. In this case, 7 further instructions are executed until the function `g` ends with a `return`.

(5) To ensure that the ITS complexity tool prefers bounds that contain the program's parameters over bounds containing the size bound variables like $b_{\mathsf{umax}_{32}}$, we first pass a modified ITS to the underlying ITS complexity tool where the initial transitions (a_0, CON, a) do not impose any conditions on the size bound variables. In other words, while all other transitions have requirements like $b'_{\mathsf{umax}_{32}} = b_{\mathsf{umax}_{32}}$ in their condition, the conditions CON of the initial transitions in this modified ITS do not contain variables like

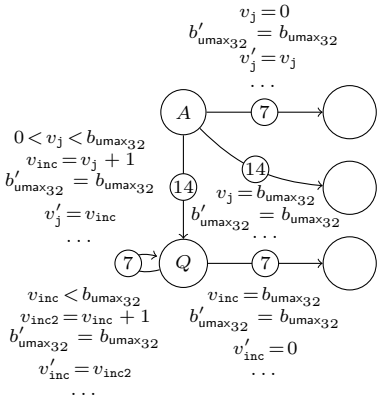

Figure 7: Weighted ITS for `g`

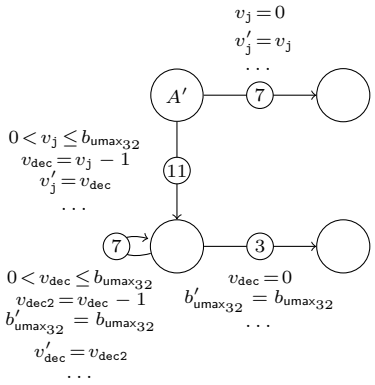

Figure 8: Modified weighted ITS for `h`

$b'_{\mathsf{umax}_{32}}$. Hence, now the size bound variables can change arbitrarily in the initial transitions and the runtime would be unbounded if it depends on one of these variables. Therefore, the complexity tool will try to find other runtime bounds that only depend on the program's parameters.

If the complexity analysis of this modified ITS fails, then instead we use the ITS as before, where the size bound variables like $b_{\mathsf{umax}_{32}}$ are considered to be input parameters. In other words, now the initial transitions also contain $b' = b$ for all size bound variables b and this ITS is now given to the complexity tool in the back-end.

As an example, consider the function `h` in Fig. 9. Here, we modified our leading example `g` from Fig. 1 and 2 by decreasing (instead of increasing) `j` until we reach `0`. The corresponding LLVM program is the same as for `g` in Fig. 2, but in Line `1` of the block `body`, `-1` is added instead of

```
void h(unsigned int j) {
    while (j > 0)
        j--;
}
```

```
define i32 @h(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: dec = add i32 j2, -1
       2: store i32 dec, i32* ad
       3: br label cmp

done:  0: ret void }
```

Figure 9: C and LLVM code for the function `h`

1. Now we first construct a modified ITS where the size variable $b_{\mathsf{umax}_{32}}$ is unbounded, i.e., where the conditions of the initial transitions do not contain $b'_{\mathsf{umax}_{32}}$. This ITS is shown in Fig. 8. When giving this ITS to the tool CoFloCo, it returns the upper bound $\max(14, 7 \cdot v_{\mathtt{j}} + 7)$. Since $v_{\mathtt{j}}$ is the initial value of the input parameter `j`, this is indeed a useful bound, since it shows that the runtime of this program is linear in the size of the program parameter `j`. To understand the reasons for the numbers 7 and 14, note that the loop of the ITS in Fig. 8 consists of 7 instructions, it is executed $(v_{\mathtt{j}} - 1)$ times, and the remaining number of instructions on the path of the loop is $11 + 3 = 14$. Hence, the path of the loop consists of $7 \cdot (v_{\mathtt{j}} - 1) + 14 = 7 \cdot v_{\mathtt{j}} + 7$ instructions. While $7 \cdot v_{\mathtt{j}} + 7$ would be a tight upper bound in this example, CoFloCo combines this with the maximum weight of all paths that do not traverse loops, which results in $\max(14, 7 \cdot v_{\mathtt{j}} + 7)$.

In contrast, for the function `g` from our leading example, no upper bound is found if the size bound variables are treated as being unbounded (i.e., if one deletes $b'_{\mathsf{umax}_{32}} = b_{\mathsf{umax}_{32}}$ from the conditions of the initial transitions). On the other hand, if one calls CoFloCo with the ITS where $b_{\mathsf{umax}_{32}}$ is considered to be an input parameter, then we obtain the bound $\max(21, 7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot v_{\mathtt{j}} + 14)$. In fact, if $0 < v_{\mathtt{j}} < b_{\mathsf{umax}_{32}}$ holds at the beginning of the program, then the loop is executed $b_{\mathsf{umax}_{32}} - v_{\mathtt{j}} - 1$ times. Since the loop of the ITS consists of 7 instructions and the path of the loop has $14 + 7 = 21$ remaining instructions, in this case we obtain $7 \cdot (b_{\mathsf{umax}_{32}} - v_{\mathtt{j}} - 1) + 21 = 7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot v_{\mathtt{j}} + 14$ instructions for the path of the loop. Again, CoFloCo combines this with the maximum weight of all paths that do not traverse loops, which results in the bound $\max(21, 7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot v_{\mathtt{j}} + 14)$.

Note that the replacement of constants by corresponding terms with variables like $b_{\mathsf{umax}_{32}}$ yields a more informative bound than the corresponding term $30064771079 - 7 \cdot v_{\mathtt{j}}$: the bound $7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot v_{\mathtt{j}} + 14$ clearly shows that the runtime depends on the range of the type `i32`. For this program, there is indeed no reasonable upper bound that depends on $v_{\mathtt{j}}$ but not on umax_{32}.

6. Related Work, Experiments, and Conclusion

We adapted our approach for proving memory safety and termination of C (resp. LLVM) programs w.r.t. the bitvector semantics of these programs. While before, program variables were treated as mathematical integers and overflows were ignored, bitvector operations such as type conversions and overflow-sensitive binary operations are now modeled correctly. To this end, we developed symbolic execution rules for all LLVM instructions that are affected by the bitvector semantics, including multiplication, division, truncation, shifts, etc., for both unsigned and signed integers. Since we represent bitvectors by relations on mathematical integers $\mathbb{Z}$, we can use standard SMT solving and standard termination analysis on $\mathbb{Z}$ for the symbolic execution and the termination proofs in our approach. Moreover, we showed that our adaption of symbolic execution

can also be used for complexity analysis. So by using existing standard tools for ITSs over mathematical integers, one can now infer upper runtime bounds for bitvector programs.

There are few other methods and tools for termination of bitvector programs (e.g., KITTeL [13, 14], TAN [9, 23], 2LS [7], Juggernaut [10], Ultimate [19][9]).[10] Compared 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* [9]. 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 [9], we represent bitvectors by relations on $\mathbb{Z}$. In contrast, 2LS, Juggernaut, and the second approach in [9] 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 abstract states for symbolic execution. In contrast, KITTeL resp. the first approach of [9] 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. `icmp 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 of the symbolic execution graph). In contrast, KITTeL only uses case analysis. While [9] 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

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

[10] 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.

	signed overflow possible				no signed overflow				
	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

Figure 10: Experimental evaluation for termination analysis

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 the termination prover AProVE [17, 21, 26] using the SMT solvers Yices [12] and Z3 [11] 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 [9] and [14], 61 of the 62 examples from the repository of Juggernaut where we excluded one example containing `float`, 7 of the 9 examples of [10] 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 integers yet, the right half of the table in Fig. 10 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. 10 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 [7]. "**T**" is the number of examples where termination was proved, "**F**" states how often the termination proof failed in ≤ 300 seconds, "**TO**" is the number of time-outs, "**RT**" is the average runtime 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. For the examples without signed overflow, there is only a single example where AProVE fails within the given time limit and one of the other tools succeeds. For the examples with signed overflows, AProVE fails or times out on 18 examples. For 10 of them, some of the other tools can prove termination. Essentially, the reason is that the symbolic execution of AProVE usually yields a more precise representation of the possible runs than the abstractions used in other tools, which is often advantageous. However, the symbolic execution may sometimes lead to extremely many nested case analyses which in turn result in a very large ITS that has to

be analyzed for termination in the end. Moreover, sometimes the generalization used to obtain finite symbolic execution graphs is too coarse such that one finally obtains a non-terminating ITS although the original program is terminating.

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.

As mentioned in Sect. 1, programs like `f` and `g` in Fig. 1 are often undesirable, since their termination behavior depends on overflows. To detect such programs, in our implementation in AProVE, the user can choose whether to interpret integers as mathematical integers (as in [27]) or as bitvectors. If one variant proves termination and the other proves non-termination, then the termination behavior indeed depends on overflows. Moreover, one could improve this analysis by implementing a variant of AProVE that disregards all knowledge about variables whenever our symbolic execution detects that they might overflow. This variant would only infer results on the termination behavior that are independent of overflows. At the same time, it would still benefit from the precise modeling of bitvector integers when dealing with bitwise operations, cf. Sect. 3.4.

To get an idea on how the termination proofs were influenced by the change from mathematical integers to bitvectors, we also ran the variant of AProVE where integers are treated as mathematical integers on the examples from our experimental evaluation. For the 52 examples with potential signed overflow, termination could be proved by both variants of AProVE for 30 examples. The bitvector variant succeeded on only 4 more examples, whereas the variant with mathematical integers could prove termination for 8 additional examples. In fact, even if many examples terminate with both semantics for integers, proving their termination with the bitvector semantics is considerably more difficult, since one has to take the potential signed overflow into account.

For the 66 examples without signed overflow, the situation is quite different. There are only 14 examples where the variant of AProVE with mathematical integers proves termination. The bitvector variant succeeds on all of them as well, and it can show termination for 47 additional examples. For one of these examples, the variant with unbounded integers shows non-termination (i.e., this example only terminates because of overflows). But for 37 of these examples, the AProVE version with mathematical integers fails because it infers that variables of type `unsigned int` could be negative or because the examples contain bitwise operations (cf. Sect. 3.4) that can only be handled when considering bitvectors. So the contributions of this paper are not only needed for the correct treatment of overflows, but they are also crucial for the handling of logical operators, conversion instructions, and shifts.

To evaluate our approach for runtime complexity presented in Sect. 5, finally we also ran a version of AProVE that searches for upper runtime bounds. As described in Sect. 5, to this end we adapted our generation of ITSs from the

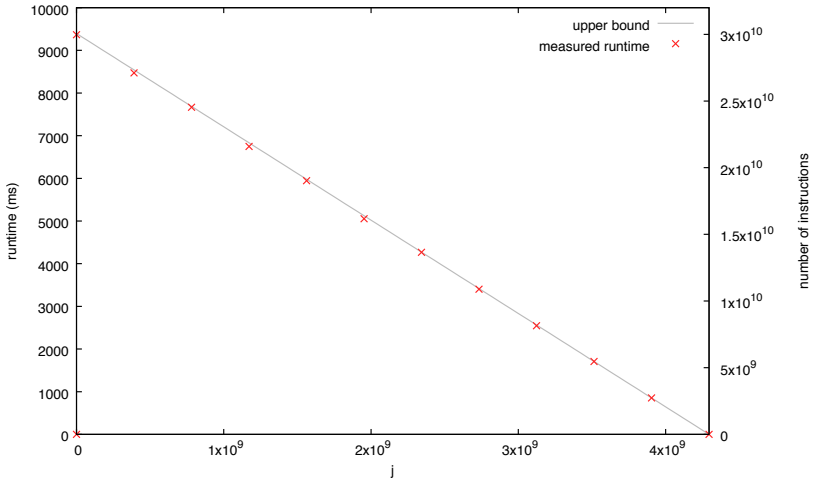

Figure 11: Actual runtimes and computed bound for the function g from Fig. 1

symbolic execution graph and used the tools KoAT [5] and CoFloCo [15, 16] in the back-end to infer upper runtime bounds for these ITSs. We always ran KoAT and CoFloCo in parallel and took the minimum of the bounds obtained by the two tools. Note that the 118 programs in our example collection were formulated according to the regulations of the *SV-COMP* competition, where there is a function `main()` that may call other functions and all input is modeled by non-deterministic values. To obtain meaningful results for runtime analysis, we modified the example programs in such a way that all non-deterministically initialized variables were moved to the parameter list of the respective functions. Out of the 95 programs where AProVE could show termination (cf. Fig. 10), AProVE infers an upper bound for 60 programs, where we again used a time-out of 300 seconds per example. For 7 of these programs, AProVE finds a small constant bound. For these 7 programs, the runtime indeed does not depend on the input variables or on the sizes of the types. For 38 programs, an upper bound is found that depends linearly on the input variable(s) and for 3 more programs, a quadratic upper bound is obtained. Thus, the runtime of these 41 programs is independent of the sizes of the integer types. For 4 programs, AProVE generates an upper bound that only depends on size bound variables. For the remaining 8 programs, the inferred runtime bound depends on both size bound variables and input variables of the function.

To evaluate the precision of the bounds computed by our approach, Fig. 11 and 12 compare these bounds with the actual runtime of the analyzed program for several inputs. In Fig. 11 we consider the function g from our leading example and indicate the measured runtime in ms for several values of the `unsigned int` variable `j` (cf. the values on the left vertical axis). Moreover, the figure shows the bound $\max(21, 7 \cdot b_{\mathsf{umax}_{32}} - 7 \cdot \mathtt{j} + 14)$ that is inferred by our implementation. Since this is a bound on the number of LLVM instructions that are executed,

```
int f(int x, int y){
 int z, i;
 z = 0;
 i = x;
 if(y<=0 || x<=0){
     return 1;
 }
 while(i > 0){
        i--;
        z++;
 }
 while(i < y){
        z--;
        i++;
 }
 return z;
}
```

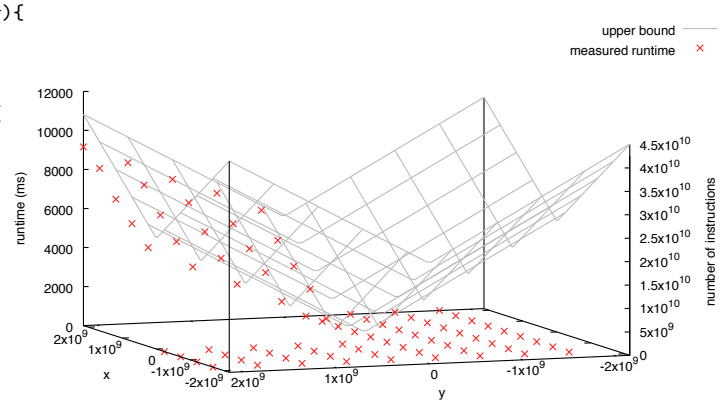

Figure 12: Actual runtimes and computed bound for a function with two inputs

here we used the scale indicated on the right vertical axis. Thus, the figure shows that up to a constant factor, the bound computed by AProVE is almost the actual runtime needed by the program, except for the case where `j` has the value 0. Nevertheless, the bound does not take into account that not all instructions need the same runtime and thus, not all actual runtimes are exactly on the line of the bound.

In Fig. 12, we consider a program from the *SV-COMP* competition with two input parameters `x` and `y`. It computes the subtraction of `x` and `y` using two `while`-loops that are executed `x` and `y` times, respectively. Here, AProVE computes the bound $10 \cdot |\mathtt{x}| + 11 \cdot |\mathtt{y}| + 161$. Fig. 12 shows that for positive values of `x` and `y`, this bound is quite precise. Nevertheless, even for positive `x` and `y` it again does not correspond to the exact runtime, because as before, our bounds ignore that different LLVM instructions can require different runtimes. Note that our bound is extremely imprecise if one of `x` or `y` is negative. The problem is that the underlying ITS complexity tools may compute bounds w.r.t. the absolute values of the integer input variables. Such bounds can lead to very coarse over-approximations of the runtime for some classes of inputs, even if the program terminates immediately for these inputs. This could be improved by a case analysis which considers the different possible signs of integer inputs separately when inferring runtime bounds.

For details on our experiments (including the results of each termination analyzer for each example and the exact runtime bounds generated by AProVE) and to access our implementation via a web interface, we refer to [3]. Moreover, we also give download links for AProVE (including a version of AProVE which allows to choose whether to interpret integers as bitvectors or as mathematical integers). Note that AProVE is constantly developed further and therefore, the results obtained by the download versions of AProVE may differ from the experimental results in this section. To reproduce the experiments, the web interface may be used. In future work, we plan to extend our approach to recursion,

to inductive data structures defined via `struct`, and to a compositional treatment of LLVM functions (the main challenge is to combine these tasks with the handling of byte-precise explicit pointer arithmetic).

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

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of object-oriented bytecode programs, Theoretical Computer Science 413 (1) (2012) 142–159.

[2] C. Alias, A. Darte, P. Feautrier, L. Gonnord, Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs, in: Proc. SAS '10, LNCS 6337, 2010, pp. 117–133.

[3] AProVE Website: `http://aprove.informatik.rwth-aachen.de/eval/BitvectorTerminationComplexity/`.

[4] R. Blanc, T. A. Henzinger, T. Hottelier, L. Kovács, ABC: Algebraic bound computation for loops, in: Proc. LPAR (Dakar) '10, LNAI 6355, 2010, pp. 103–118.

[5] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, J. Giesl, Analyzing runtime and size complexity of integer programs, ACM Transactions on Programming Languages and Systems 38 (4) (2016) 13:1–13:50.

[6] Q. Carbonneaux, J. Hoffmann, Z. Shao, Compositional certified resource bounds, in: Proc. PLDI '15, 2015, pp. 467–478.

[7] H.-Y. Chen, C. David, D. Kroening, P. Schrammel, B. Wächter, Synthesising interprocedural bit-precise termination proofs, in: Proc. ASE '15, 2015, pp. 53–64.

[8] Clang compiler: `http://clang.llvm.org`.

[9] B. Cook, D. Kroening, P. Rümmer, C. Wintersteiger, Ranking function synthesis for bit-vector relations, Formal Methods in System Design 43 (1) (2013) 93–120.

[10] C. David, D. Kroening, M. Lewis, Unrestricted termination and non-termination arguments for bit-vector programs, in: Proc. ESOP '15, LNCS 9032, 2015, pp. 183–204.

[11] L. M. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: Proc. TACAS '08, LNCS 4963, 2008, pp. 337–340.

[12] B. Dutertre, L. M. de Moura, The Yices SMT solver, Tool paper at `http://yices.csl.sri.com/tool-paper.pdf` (2006).

[13] S. Falke, D. Kapur, C. Sinz, Termination analysis of C programs using compiler intermediate languages, in: Proc. RTA '11, LIPIcs 10, 2011, pp. 41–50.

[14] S. Falke, D. Kapur, C. Sinz, Termination analysis of imperative programs using bitvector arithmetic, in: Proc. VSTTE '12, LNCS 7152, 2012, pp. 261–277.

[15] A. Flores-Montoya, R. Hähnle, Resource analysis of complex programs with cost equations, in: Proc. APLAS '14, LNCS 8858, 2014, pp. 275–295.

[16] A. Flores-Montoya, Upper and lower amortized cost bounds of programs expressed as cost relations, in: Proc. FM '16, LNCS 9995, 2016, pp. 254–273.

[17] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, R. Thiemann, Analyzing program termination and complexity automatically with AProVE, Journal of Automated Reasoning 58 (2017) 3–31.

[18] S. Gulwani, SPEED: Symbolic complexity bound analysis, in: Proc. CAV '09, LNCS 5643, 2009, pp. 51–62.

[19] M. Heizmann, J. Hoenicke, J. Leike, A. Podelski, Linear ranking for linear lasso programs, in: Proc. ATVA '13, LNCS 8172, 2013, pp. 365–380.

[20] J. Hensel, J. Giesl, F. Frohn, T. Ströder, Proving termination of programs with bitvector arithmetic by symbolic execution, in: Proc. SEFM '16, LNCS 9763, 2016, pp. 234–252.

[21] J. Hensel, F. Emrich, F. Frohn, T. Ströder, J. Giesl, AProVE: Proving and disproving termination of memory-manipulating C programs (competition contribution), in: Proc. TACAS '17, LNCS 10206, 2017, pp. 350–354.

[22] J. Hoffmann, A. Das, S.-C. Weng, Towards automatic resource bound analysis for OCaml, in: Proc. POPL '17, 2017, pp. 359–373.

[23] D. Kroening, N. Sharygina, A. Tsitovich, C. Wintersteiger, Termination analysis with compositional transition invariants, in: Proc. CAV '10, LNCS 6174, 2010, pp. 89–103.

[24] C. Lattner, V. S. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in: Proc. CGO '04, 2004, pp. 75–88.

[25] M. Sinn, F. Zuleger, H. Veith, Difference constraints: An adequate abstraction for complexity analysis of imperative programs, in: Proc. FMCAD '15, 2015, pp. 144–151.

[26] T. Ströder, C. Aschermann, F. Frohn, J. Hensel, J. Giesl, AProVE: Termination and memory safety of C programs (competition contribution), in: Proc. TACAS '15, LNCS 9035, 2015, pp. 417–419.

[27] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, C. Aschermann, Automatically proving termination and memory safety for programs with pointer arithmetic, Journal of Automated Reasoning 58 (2017) 33–65.

Appendix A. Separation Logic Semantics of Abstract States

As mentioned in Sect. 2, to formalize the semantics of an abstract state a, in [27] we introduced a separation logic formula $\langle a \rangle_{SL}$, which extends $\langle a \rangle_{FO}$ by information about the memory. We consider a fragment of separation logic which augments first-order logic by a predicate symbol "$\hookrightarrow$" for "points-to" information and by the connective "$*$" for the separating conjunction. As usual, $\varphi_1 * \varphi_2$ means that φ_1 and φ_2 hold for disjoint parts of the memory.

In $\langle a \rangle_{SL}$, we combine the elements of AL with the separating conjunction "$*$" to express that different allocated memory blocks are disjoint. 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 9 (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 $\langle v_1 \hookrightarrow_{n+8} v_3 \rangle_{SL} = (v_1 \hookrightarrow (v_3 \bmod 2^8)) \wedge \langle\, (v_1 + 1) \hookrightarrow_n (v_3 \operatorname{div} 2^8) \,\rangle_{SL}$.

Then a state $a = (p, LV, KB, AL, PT)$ is represented in separation logic by[11]

$$\langle a \rangle_{SL} = \langle a \rangle_{FO} \;\wedge\; (\textstyle\ast_{\varphi \in AL}\; \langle \varphi \rangle_{SL}) \;\wedge\; (\bigwedge\nolimits_{\varphi \in PT}\; \langle \varphi \rangle_{SL}).$$

We use *interpretations* (as, mem) for the semantics of separation logic. To deal with symbolic variables in formulas, we use *instantiations*. Let $\mathcal{T}(\mathcal{V}_{sym})$ be the set of all arithmetic terms containing only variables from $\mathcal{V}_{sym}$. Any function $\sigma : \mathcal{V}_{sym} \to \mathcal{T}(\mathcal{V}_{sym})$ is called an instantiation and as in Sect. 2, an instantiation is *concrete* iff $\sigma(v) \in \mathbb{Z}$ for all $v \in \mathcal{V}_{sym}$. Again, we write "$\rightharpoonup$" for partial functions.

[11] Here, we assume the empty separating conjunction to be *true*, i.e., if $AL = \varnothing$ then $\ast_{\varphi \in AL}\; \langle \varphi \rangle_{SL} = true$.

Definition 10 (Semantics of Separation Logic). *Let $as : \mathcal{V}_{\mathcal{P}} \to \mathbb{Z}$ be an assignment, $mem : \mathbb{N}_{>0} \rightharpoonup \{0, \ldots, \mathsf{umax}_8\}$, and φ be a formula. Let $as(\varphi)$ result from replacing all local variables $\mathtt{x}$ in φ by the value $as(\mathtt{x})$. By construction, local variables $\mathtt{x}$ 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 $v_1, \ldots, v_n$ in ψ are implicitly universally quantified, i.e., $mem \models \psi$ iff $mem \models \forall v_1, \ldots, v_n. \psi$. 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 $*$ for variable-free formulas. For $n_1, n_2 \in \mathbb{Z}$, let $mem \models n_1 \hookrightarrow n_2$ hold iff $mem(n_1) = n_2$.[12] *The semantics of $*$ is defined as usual in separation logic: For two partial functions $mem_1, mem_2 : \mathbb{N}_{>0} \rightharpoonup \mathbb{Z}$, we write $mem_1 \perp mem_2$ to indicate that the domains of mem_1 and mem_2 are disjoint. If $mem_1 \perp mem_2$, then $mem_1 \uplus mem_2$ denotes the union of mem_1 and mem_2. Now $mem \models \varphi_1 * \varphi_2$ holds iff there exist $mem_1 \perp mem_2$ such that $mem = mem_1 \uplus mem_2$ where $mem_1 \models \varphi_1$ and $mem_2 \models \varphi_2$.*

Appendix 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.

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

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

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

[12] 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.

Thus, we have:

$$\begin{array}{rcrcll} t \leq u & \Rightarrow & t-\ell & \leq & u-\ell & \\ & \Rightarrow & (t-\ell) \mod 2^n & \leq & u-\ell+2^n & \text{by (B.1)} \\ & \Rightarrow & ((t-\ell) \mod 2^n)+\ell & \leq & u+2^n & \\ & \Rightarrow & \multicolumn{3}{l}{\mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max}) \text{ holds}} \end{array}$$

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

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

We obtain:

$$\begin{array}{rcrcll} t \geq u+1 & \Rightarrow & t-\ell & \geq & u-\ell+1 & \\ & \Rightarrow & (t-\ell) \mod 2^n & \geq & u-\ell+1+2^n & \text{by (B.2)} \\ & \Rightarrow & ((t-\ell) \mod 2^n)+\ell & \geq & u+1+2^n & \\ & \Rightarrow & \multicolumn{3}{l}{\mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max}) \text{ does not hold}} \end{array}$$

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.,

$$0 \leq \mathsf{max}-\ell < 2^n. \tag{B.3}$$

In addition, we have

$$\mathsf{max} = \mathsf{min}+2^n-1 \leq u+2^n-1. \tag{B.4}$$

Here, we obtain:

$$\begin{array}{rcrcll} t \leq \mathsf{max} & \Rightarrow & t-\ell & \leq & \mathsf{max}-\ell & \\ & \Rightarrow & (t-\ell) \mod 2^n & \leq & \mathsf{max}-\ell & \text{by (B.3)} \\ & \Rightarrow & ((t-\ell) \mod 2^n)+\ell & \leq & \mathsf{max} & \\ & \Rightarrow & ((t-\ell) \mod 2^n)+\ell & \leq & u+2^n & \text{by (B.4)} \\ & \Rightarrow & \multicolumn{3}{l}{\mathsf{inBounds}(t, \mathsf{min}, u, \ell, \mathsf{max}) \text{ holds}} \end{array}$$

□

Proof of Thm. 8. The proof of Thm. 8 is identical to the proofs of Thm. 10 and 13 in [27]. 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 by some abstract state a in the symbolic execution graph $\mathcal{G}$, then every LLVM evaluation of c corresponds to a path in the graph. More precisely, for every LLVM evaluation step $c \to_{\mathsf{LLVM}} \overline{c}$ there is a path from a to an abstract state $\overline{a}$ in $\mathcal{G}$ such that $\overline{c}$ is represented by $\overline{a}$.

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 [27] directly carries over to the present setting. □