Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic

**Thomas Ströder · Jürgen Giesl ·
Marc Brockschmidt · Florian Frohn ·
Carsten Fuhs · Jera Hensel ·
Peter Schneider-Kamp · Cornelius Aschermann**

Abstract While automated verification of imperative programs has been studied intensively, proving termination of programs with explicit pointer arithmetic fully automatically was still an open problem. To close this gap, we introduce a novel abstract domain that can track allocated memory in detail. We use it to automatically construct a *symbolic execution graph* that over-approximates all possible runs of a program and that can be used to prove memory safety. This graph is then transformed into an *integer transition system*, whose termination can be proved by standard techniques. We implemented this approach in the automated termination prover AProVE and demonstrate its capability of analyzing C programs with pointer arithmetic that existing tools cannot handle.

Keywords LLVM · C programs · Termination · Memory Safety · Symbolic Execution

1 Introduction

Consider the following standard C implementation of `strlen` [62, 72], computing the length of the string at the pointer `str`. In C, strings are usually represented as a pointer `str` to the heap, where all following memory cells up to the first one that contains the value 0 are allocated memory and form the value of the string.

```
int strlen(char* str) {char* s = str; while(*s) s++; return s-str;}
```

To analyze algorithms on such data, one has to handle the interplay between addresses and the values they point to. In C, a violation of *memory safety* (e.g., dereferencing `NULL`, accessing an array outside its bounds, etc.) leads to undefined behavior, which may also

Supported by Deutsche Forschungsgemeinschaft (DFG) grant GI 274/6-1, Research Training Group 1298 (*AlgoSyn*), and the Danish Council for Independent Research, Natural Sciences.

Thomas Ströder · Jürgen Giesl · Florian Frohn · Jera Hensel · Cornelius Aschermann
LuFG Informatik 2, RWTH Aachen University, Germany

Marc Brockschmidt
Microsoft Research Cambridge, UK

Carsten Fuhs
Dept. of Computer Science and Information Systems, Birkbeck, University of London, UK

Peter Schneider-Kamp
Dept. of Mathematics and Computer Science, University of Southern Denmark, Denmark

include non-termination. Thus, to prove termination of C programs with low-level memory access, one must also ensure memory safety. The `strlen` algorithm is memory safe and terminates, because there is some address `end` $\geq$ `str` (an *integer property* of `end` and `str`) such that `*end` is 0 (a *pointer property* of `end`) and all addresses `str` $\leq$ `s` $\leq$ `end` are allocated. Other typical programs with pointer arithmetic operate on arrays (which are just sequences of memory cells in C). In this paper, we present a novel approach to prove memory safety and termination of algorithms on integers and pointers automatically. Our abstract domain is tailored to track both integer properties which relate allocated memory addresses with each other, as well as pointer properties about the data stored at such addresses.

To avoid handling the intricacies of C, we analyze programs in the platform-independent intermediate representation (IR) of the LLVM compilation framework [51,53]. Our approach works in three steps: First, a *symbolic execution graph* is created that represents an over-approximation of all possible program runs. We present our abstract domain based on *separation logic* [61] in Sect. 2 and the automated generation of such graphs in Sect. 3. In Sect. 4 we show the correctness of our construction. In this first step from LLVM to the symbolic execution graph, we handle all issues related to memory, and in particular we prove memory safety of our input program. In Sect. 5, we describe the second step of our approach, in which we generate an *integer transition system* (ITS) from the symbolic execution graph, encoding the essential information needed to show termination. In the last step, existing techniques for integer programs are used to prove termination of the resulting ITS. In Sect. 6, we compare our approach with related work and show that our implementation in the termination prover AProVE proves memory safety and termination of typical pointer algorithms that could not be handled by other tools before.

A preliminary version of parts of this paper was published in [67]. The present paper extends [67] by the following new contributions:

- We lift the restriction of analyzing only programs with exactly one function to non-recursive programs with several functions.
- We show how to consider alignment information in the abstract domain. In [67], we just assumed a 1 byte data alignment for all types.
- In [67], we only handled memory allocation using the LLVM instruction `alloca`. In this paper, we extend our abstract domain and our symbolic execution rules to handle the external functions `malloc` and `free`. This allows us to model memory safety more precisely. Up to now, we could only prove absence of accesses to unallocated memory, whereas now, we can also show that `free` is only called for addresses that have been returned by `malloc` and that have not been released already. Note that if memory is not released by the end of the program, then we do not consider this as a violation of memory safety, because it does not lead to undefined behavior.
- We added more symbolic execution rules for LLVM instructions, and give a detailed overview of our limitations in Sect. 6.
- To represent all possible program runs by a finite symbolic execution graph, it is crucial to *merge* abstract program states that visit the same program position. We have substantially improved the merging heuristic of [67] in order to also analyze programs where termination or memory safety depend on invariants relating different areas of allocated memory. Such reasoning is required for programs like the `strcpy` function from the standard C library. Our symbolic execution can now handle such programs automatically, whereas [67] fails to prove memory safety (and hence also termination).
- We prove the soundness of our approach w.r.t. the formal LLVM semantics from [73], and provide all proofs in the paper.

2 Abstract Domain for Symbolic Execution

In this section, we introduce concrete LLVM states and *abstract* states that represent *sets* of concrete states. These states will be needed for symbolic execution in Sect. 3.

To simplify the presentation, we restrict ourselves to 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. Like many other approaches to termination analysis, we disregard integer overflows and assume that variables are only instantiated with signed integers appropriate for their type.

We consider the `strlen` function from Sect. 1. In the corresponding LLVM code,[1] `str` has the type `i8*`, since it is a pointer to the string's first character (of type `i8`). The program is split into the *basic blocks* `entry`, `loop`, and `done`. We will explain this LLVM code in detail when constructing the symbolic execution graph in Sect. 3.

```
define i32 @strlen(i8* str) {
entry: 0: c0 = load i8* str
       1: c0zero = icmp eq i8 c0, 0
       2: br i1 c0zero, label done, label loop
loop:  0: olds = phi i8* [str,entry],[s,loop]
       1: s = getelementptr i8* olds, i32 1
       2: c = load i8* s
       3: czero = icmp eq i8 c, 0
       4: br i1 czero, label done, label loop
done:  0: sfin = phi i8* [str,entry],[s,loop]
       1: sfinint = ptrtoint i8* sfin to i32
       2: strint = ptrtoint i8* str to i32
       3: size = sub i32 sfinint, strint
       4: ret i32 size                         }
```

An LLVM state consists of a call stack, a knowledge base with information about the values of symbolic variables, and two sets which describe memory allocations and the contents of memory. The call stack is a sequence of stack frames, where each stack frame contains information local to its corresponding function. In particular, a stack frame contains the current *program position* which is represented by a pair $(\mathtt{b}, j)$. Here, `b` is the name of the current basic block and j is the index of the next instruction. So if *Blks* is the set of all basic blocks, then the set of program positions is $\mathit{Pos} = \mathit{Blks} \times \mathbb{N}$. To ease the formalization, we assume that different functions do not have basic blocks with the same names. Moreover, a stack frame also contains information on the current values of the local program variables. We represent an assignment to the *l*ocal *v*ariables $\mathcal{V}_\mathcal{P}$ (e.g., $\mathcal{V}_\mathcal{P} = \{\mathtt{str}, \mathtt{c0}, \ldots\}$) in the i-th stack frame as a partial function $LV_i : \mathcal{V}_\mathcal{P} \rightharpoonup \mathcal{V}_{sym}$ (where "$\rightharpoonup$" denotes partial functions). We use an infinite set of symbolic variables $\mathcal{V}_{sym}$ with $\mathcal{V}_{sym} \cap \mathcal{V}_\mathcal{P} = \{\}$ instead of concrete integers. In this way, our states can represent not only *concrete* execution states, where all symbolic variables $v \in \mathcal{V}_{sym}$ are constrained to a concrete fixed number in $\mathbb{Z}$, but also *abstract* states, where v can stand for several possible values. Such states will be needed for symbolic execution. To ease the generalization of states in Sect. 3.3, we require that all LV_i occurring in a call stack are injective and have pairwise disjoint ranges. Let $\mathcal{V}_{sym}(LV_i) \subseteq \mathcal{V}_{sym}$ be the set of all symbolic variables v where there exists some $\mathtt{x} \in \mathcal{V}_\mathcal{P}$ with $LV_i(\mathtt{x}) = v$.

In addition to the values of local variables, each stack frame also contains an *a*llocation *l*ist AL_i. This list contains expressions of the form $[\![v_1, v_2]\!]$ for $v_1, v_2 \in \mathcal{V}_{sym}$, which indicate that $v_1 \leq v_2$ and that all addresses between v_1 and v_2 have been allocated by an `alloca` instruction. This information is stored in the stack frames, as memory allocated by `alloca` in a function is automatically released when the control flow returns from that function.

A program position, a variable assignment and an allocation list form a stack frame FR, and we represent call stacks as sequences $[FR_1, \ldots, FR_n]$ of such stack frames, where the i-th stack frame has the form $FR_i = (p_i, LV_i, AL_i)$. The topmost frame is FR_1, and we use "$\cdot$" to decompose call stacks, i.e., $[FR_1, \ldots, FR_n] = FR_1 \cdot [FR_2, \ldots, FR_n]$. A new stack frame

[1] This LLVM program corresponds to the code obtained from `strlen` with the Clang compiler [23]. To ease readability, we wrote variables without "`%`" in front (i.e., we wrote "`str`" instead of "`%str`" as in proper LLVM) and added line numbers.

is added in front of the sequence whenever a function is called, and removed when control returns from it. For any call stack $CS = [FR_1, \ldots, FR_n]$ where each stack frame FR_i uses the partial function LV_i for the local variables, let $\mathcal{V}_{sym}(CS)$ consist of $\mathcal{V}_{sym}(LV_1) \cup \ldots \cup \mathcal{V}_{sym}(LV_n)$ and of all symbolic variables occurring in $AL_1, \ldots,$ or AL_n.

The second component of our LLVM states is the *knowledge base* $KB \subseteq QF\_IA(\mathcal{V}_{sym})$, a set of quantifier-free first-order formulas that express integer arithmetic properties of $\mathcal{V}_{sym}$. For concrete states, the knowledge base constrains $\mathcal{V}_{sym}(CS)$ in such a way that their values are uniquely determined, whereas for abstract states several values are possible.

Many of the rules for symbolic execution in Sect. 3 have conditions where one has to check validity of formulas obtained from the knowledge base of the current state. In principle, any SMT solver can be used for this check. Most of these formulas only use *linear* integer arithmetic, but for programs with non-linear expressions (like `x * y`), the resulting formulas can also contain non-linear arithmetic. As validity is not decidable for non-linear integer arithmetic, the power of the SMT solver influences the power of our analysis, since symbolic execution rules can only be applied if the proof for their applicability conditions succeeds.

The third component is the global allocation list AL. It is used to model memory allocated by `malloc`, where allocated parts of the memory are again represented by expressions of the form $[\![v_1, v_2]\!]$. In contrast to `alloca`, memory allocated by `malloc` needs to be released explicitly by the programmer. In this paper, we assume that reading from memory locations that are currently allocated but not initialized, yields an arbitrary fixed value. To remove this assumption, a structure similar to AL could be used to track initialized memory regions.

As the fourth and final component, PT is a set of "points-to" atoms $v_1 \hookrightarrow_{\texttt{ty}} v_2$ where $v_1, v_2 \in \mathcal{V}_{sym}$ and `ty` is an LLVM type. This means that the value v_2 of type `ty` is stored at the address v_1. Let $size(\texttt{ty})$ be the number of bytes required for values of type `ty` (e.g., $size(\texttt{i8}) = 1$ and $size(\texttt{i32}) = 4$). As each memory cell stores one byte, $v_1 \hookrightarrow_{\texttt{i32}} v_2$ means that v_2 is stored in the four cells at the addresses $v_1, \ldots, v_1 + 3$. The size of a pointer type `ty*` is determined by the data layout string in the beginning of an LLVM program. On 64-bit machine architectures, we usually have $size(\texttt{ty*}) = 8$, and on 32-bit architectures we usually have $size(\texttt{ty*}) = 4$. In the following let us consider some fixed value for $size(\texttt{ty*})$.

Finally, to model possible violations of memory safety, we introduce a special error state ERR. In particular, this state is reached when accessing non-allocated memory. The following definition introduces our notion of (possibly abstract) LLVM states formally.

Definition 1 (LLVM States) LLVM *states* have the form (CS, KB, AL, PT) where $CS \in (Pos \times (\mathcal{V}_{\mathcal{P}} \rightharpoonup \mathcal{V}_{sym}) \times \{[\![v_1, v_2]\!] \mid v_1, v_2 \in \mathcal{V}_{sym}\})^*$, $KB \subseteq QF\_IA(\mathcal{V}_{sym})$, $AL \subseteq \{[\![v_1, v_2]\!] \mid v_1, v_2 \in \mathcal{V}_{sym}\}$, and $PT \subseteq \{(v_1 \hookrightarrow_{\texttt{ty}} v_2) \mid v_1, v_2 \in \mathcal{V}_{sym},\ \texttt{ty} \text{ is an LLVM type}\}$. Additionally, there is a state ERR for possible memory safety violations. For a state $a = (CS, KB, AL, PT)$, let $\mathcal{V}_{sym}(a)$ consist of $\mathcal{V}_{sym}(CS)$ and of all symbolic variables occurring in KB, AL, or PT.

In a call stack $CS = [(p_1, LV_1, AL_1), \ldots, (p_n, LV_n, AL_n)]$, we often identify the mapping LV_i with the set of equations $\{\texttt{x}_i = LV_i(\texttt{x}) \mid \texttt{x} \in \mathcal{V}_{\mathcal{P}}, LV_i(\texttt{x}) \text{ is defined}\}$ and extend LV_i to a function from $\mathcal{V}_{\mathcal{P}} \uplus \mathbb{Z}$ to $\mathcal{V}_{sym} \uplus \mathbb{Z}$ by defining $LV_i(n) = n$ for all $n \in \mathbb{Z}$. We also often identify CS with the set of equations $\bigcup_{1 \leq i \leq n} \{\texttt{x}_i = LV_i(\texttt{x}) \mid \texttt{x} \in \mathcal{V}_{\mathcal{P}}, LV_i(\texttt{x}) \text{ is defined}\}$. Let $\mathcal{V}_{\mathcal{P}}^{fr} = \{\texttt{x}_i \mid \texttt{x} \in \mathcal{V}_{\mathcal{P}}, i \in \mathbb{N}_{>0}\}$ be the set of all these indexed variables that we use to represent stack frames. Moreover, we write AL^* for the union of the global allocation list with the allocation lists in the individual stack frames, i.e., $AL^* = AL \cup AL_1 \cup \ldots \cup AL_n$. Thus, AL^* represents all currently allocated memory (by `alloca` or `malloc`) in the current state. We say that a state (CS, KB, AL, PT) is *well formed* iff for every "points-to" information $v \hookrightarrow_{\texttt{ty}} w \in PT$, there is an allocated area $[\![v_1, v_2]\!]$ in AL^* such that $\models KB \Rightarrow v_1 \leq v \wedge v \leq v_2$. So PT only contains information about addresses that are known to be allocated.

As an example, consider the following abstract state for our `strlen` program:

$$(\ [(\,(\texttt{entry},0),\ \ \{\texttt{str}_1 = u_{\texttt{str}}\},\ \ \{\}\,)],\ \ \{z=0\},\ \ \{[\![u_{\texttt{str}}, v_{end}]\!]\},\ \ \{v_{end} \hookrightarrow_{\texttt{i8}} z\}\ ) \qquad (\dagger)$$

It represents states at the beginning of the `entry` block, where $CS = [((\texttt{entry},0), LV_1, \{\})]$ with $LV_1(\texttt{str}) = u_{\texttt{str}}$ and no memory was allocated by `alloca`. Due to an earlier call of `malloc`, the memory cells between $LV_1(\texttt{str}) = u_{\texttt{str}}$ and v_{end} are allocated on the heap, and the value at the address v_{end} is z (where the knowledge base is $\{z = 0\}$).

To define the semantics of abstract states a, we introduce the formulas $\langle a\rangle_{SL}$ and $\langle a\rangle_{FO}$. Here, $\langle a\rangle_{SL}$ is a formula from a fragment of *separation logic* [61] that defines which concrete states are represented by a. The first-order formula $\langle a\rangle_{FO}$ is a weakened version of $\langle a\rangle_{SL}$, used for the automation of our approach. We use it to construct symbolic execution graphs, as it allows us to apply standard SMT solving [59] for all reasoning. We also use $\langle a\rangle_{FO}$ for the subsequent generation of integer transition systems from symbolic execution graphs.

The formula $\langle a\rangle_{FO}$ contains KB, and in addition, it expresses that the pairs $[\![v_1, v_2]\!]$ in allocation lists represent disjoint intervals. Moreover, two values at the same address must be equal and two addresses must be different if they point to different values in PT. Finally, all addresses are positive numbers.

Definition 2 (Representing States by *FO* Formulas) The set $\langle a\rangle_{FO}$ is the smallest set with

$$\begin{aligned}\langle a\rangle_{FO} = KB \cup \{&1 \le v_1 \wedge v_1 \le v_2 \mid [\![v_1, v_2]\!] \in AL^*\} \cup\\ \{&v_2 < w_1 \vee w_2 < v_1 \mid [\![v_1, v_2]\!], [\![w_1, w_2]\!] \in AL^*,\ (v_1,v_2) \neq (w_1,w_2)\} \cup\\ \{&v_2 = w_2 \mid (v_1 \hookrightarrow_{\texttt{ty}} v_2), (w_1 \hookrightarrow_{\texttt{ty}} w_2) \in PT \text{ and } \models \langle a\rangle_{FO} \Rightarrow v_1 = w_1\} \cup\\ \{&v_1 \neq w_1 \mid (v_1 \hookrightarrow_{\texttt{ty}} v_2), (w_1 \hookrightarrow_{\texttt{ty}} w_2) \in PT \text{ and } \models \langle a\rangle_{FO} \Rightarrow v_2 \neq w_2\} \cup\\ \{&v_1 > 0 \mid (v_1 \hookrightarrow_{\texttt{ty}} v_2) \in PT\}.\end{aligned}$$

Now we formally define "*concrete* states" as abstract states of a particular form. A concrete state c *uniquely* describes the call stack and the contents of the memory. So we require that (a) $\langle c\rangle_{FO}$ is satisfiable to ensure that c actually *can* represent something, and that (b) c has unique values for the contents of all allocated addresses. Here, we represent memory data byte-wise, and since LLVM represents values in two's complement, each byte stores a value from $[-2^7, 2^7-1]$. This byte-wise representation of the memory enforces a uniform representation of concrete states, and thus (c) we allow only statements of the form $w_1 \hookrightarrow_{\texttt{i8}} w_2$ in PT for concrete states. Moreover, this restriction ensures that concrete states are really legal states. (Otherwise, we would have to check whether two statements $w_1 \hookrightarrow_{\texttt{ty}_1} w_2$ and $w_1 \hookrightarrow_{\texttt{ty}_2} w_3$ with $\texttt{ty}_1 \neq \texttt{ty}_2$ in the same state are compatible or whether they contradict each other.) Finally, (d) all occurring symbolic variables must have unique values.

Definition 3 (Concrete States) Let $c = (CS, KB, AL, PT)$ be an LLVM state. We call c a *concrete state* iff c is well formed and all of the following conditions hold:

(a) $\langle c\rangle_{FO}$ is satisfiable,

(b) for all $[\![v_1, v_2]\!] \in AL^*$ and for all integers n with $\models \langle c\rangle_{FO} \Rightarrow v_1 \le n \wedge n \le v_2$, there exists $(w_1 \hookrightarrow_{\texttt{i8}} w_2) \in PT$ 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 $k \in [-2^7, 2^7-1]$,

(c) there is no $w_1 \hookrightarrow_{\texttt{ty}} w_2 \in PT$ for $\texttt{ty} \neq \texttt{i8}$,

(d) for all $v \in \mathcal{V}_{sym}(c)$ there exists an $n \in \mathbb{Z}$ such that $\models \langle c\rangle_{FO} \Rightarrow v = n$.

Moreover, *ERR* is also a concrete state.

A state $a \neq ERR$ always stands for a memory-safe state where exactly the addresses in AL^* are allocated. Let $\to_{\mathsf{LLVM}}$ be LLVM's evaluation relation on concrete states, i.e., $c \to_{\mathsf{LLVM}} \overline{c}$ holds iff c evaluates to $\overline{c}$ by executing one LLVM instruction. Similarly, $c \to_{\mathsf{LLVM}} ERR$

means that the evaluation step performs an operation that may lead to undefined behavior. An LLVM program is *memory safe* for $c \neq ERR$ iff there is no evaluation $c \to^{+}_{\mathsf{LLVM}} ERR$, where $\to^{+}_{\mathsf{LLVM}}$ is the transitive closure of $\to_{\mathsf{LLVM}}$.

As mentioned, in addition to $\langle a \rangle_{FO}$, we also introduce a separation logic formula $\langle a \rangle_{SL}$ for every state a. We consider a fragment of separation logic which extends first-order logic by a predicate symbol "$\hookrightarrow$" for "points-to" information and by the connective "$*$" for separating conjunction. As usual, $\varphi_1 * \varphi_2$ means that φ_1 and φ_2 hold for disjoint parts of the memory. The semantics of separation logic can then be defined using *interpretations* of the form (as, mem) which represent the values of the program variables and the heap. The (partial) *assignment* function $as : \mathcal{V}_{\mathcal{P}}^{fr} \rightharpoonup \mathbb{Z}$ is used to describe the values of the program variables (more precisely, as operates on variables of the form $\mathtt{x}_i$ to represent the variable $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ occurring in the i-th stack frame). Moreover, a partial function $mem : \mathbb{N}_{>0} \rightharpoonup \{0, \ldots, 2^8 - 1\}$ with finite domain describes the *memory* contents at allocated addresses (as unsigned bytes).

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. Thus, σ does not instantiate $\mathcal{V}_{\mathcal{P}}^{fr}$. 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 $\sigma(v)$. An instantiation is called *concrete* iff $\sigma(v) \in \mathbb{Z}$ for all $v \in \mathcal{V}_{sym}$.

Definition 4 (Semantics of Separation Logic) Let $as : \mathcal{V}_{\mathcal{P}}^{fr} \rightharpoonup \mathbb{Z}$, $mem : \mathbb{N}_{>0} \rightharpoonup \{0, \ldots, 2^8 - 1\}$, and let φ be a formula such that as is defined on all variables from $\mathcal{V}_{\mathcal{P}}^{fr}$ that occur in φ. Let $as(\varphi)$ result from replacing all $\mathtt{x}_i$ in φ by $as(\mathtt{x}_i)$. Note that by construction, local variables $\mathtt{x}_i$ 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 $\mathcal{V}_{sym}$ (this is needed for Sect. 3). 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 $mem(n_1) = n_2$.[2] 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 $mem_1 \bot mem_2$ to indicate that the domains of mem_1 and mem_2 are disjoint. If $mem_1 \bot mem_2$, then $mem_1 \uplus mem_2$ denotes the union of mem_1 and mem_2. Now $mem \models \varphi_1 * \varphi_2$ holds iff there exist $mem_1 \bot mem_2$ such that $mem = mem_1 \uplus mem_2$ where $mem_1 \models \varphi_1$ and $mem_2 \models \varphi_2$. As usual, "$\models \varphi$" means that φ is a tautology, i.e., that $(as, mem) \models \varphi$ holds for any interpretation (as, mem).

To formalize the semantics of an abstract state a, i.e., to define which concrete states are represented by a, we now define $\langle a \rangle_{SL}$. In $\langle a \rangle_{SL}$, we combine the elements of AL^* with the separating conjunction "$*$" to express that different allocated memory blocks are disjoint. We have to include an additional separated conjunct *true* to represent further allocations that we do not know of. In contrast, the elements of PT are combined by the ordinary conjunction "$\wedge$". So $(v_1 \hookrightarrow_{\mathtt{ty}} v_2) \in PT$ does not imply that v_1 is different from other addresses occurring in PT. Similarly, we also combine the two formulas resulting from AL^* and PT by "$\wedge$", as both express different properties of the same memory addresses.

Definition 5 (Representing States by *SL* Formulas) For $v_1, v_2 \in \mathcal{V}_{sym}$, let $\langle [\![v_1, v_2]\!] \rangle_{SL} =$

[2] 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.

$$1 \leq v_1 \wedge v_1 \leq v_2 \wedge (\forall x.\exists y.\ (v_1 \leq x \leq v_2) \Rightarrow (x \hookrightarrow y)).$$

Reflecting two's complement representation, for any LLVM type `ty`, we define $\langle v_1 \hookrightarrow_{\mathtt{ty}} v_2 \rangle_{SL} =$

$$v_1 > 0 \wedge \langle v_1 \hookrightarrow_{size(\mathtt{ty})} v_3 \rangle_{SL} \wedge (v_2 \geq 0 \Rightarrow v_3 = v_2) \wedge (v_2 < 0 \Rightarrow v_3 = v_2 + 2^{8 \cdot size(\mathtt{ty})}),$$

where $v_3 \in \mathcal{V}_{sym}$ is fresh. We assume a little-endian data layout (where the least significant byte is stored in the lowest address).[3] Here, we let $\langle v_1 \hookrightarrow_0 v_3 \rangle_{SL} = true$ and $\langle v_1 \hookrightarrow_{n+1} v_3 \rangle_{SL} = (v_1 \hookrightarrow (v_3 \bmod 2^8)) \wedge \langle (v_1 + 1) \hookrightarrow_n (v_3 \operatorname{div} 2^8) \rangle_{SL}$.

Let $a = (CS, KB, AL, PT)$ be an abstract state. It is represented in separation logic by[4]

$$\langle a \rangle_{SL} = CS \wedge KB \wedge (true * (\ast_{\varphi \in AL^*} \ \langle \varphi \rangle_{SL})) \wedge (\textstyle\bigwedge_{\varphi \in PT} \ \langle \varphi \rangle_{SL})$$

Clearly, we have $\models \langle a \rangle_{SL} \Rightarrow \langle a \rangle_{FO}$ for any abstract state a. So $\langle a \rangle_{FO}$ only contains first-order information that holds in every concrete state represented by a.

Now we can define which concrete states are represented by an abstract state. Note that due to Def. 3, we can extract an interpretation (as^c, mem^c) from every concrete state $c \neq ERR$. Then we define that a (well-formed) abstract state *a represents* all those concrete states c where (as^c, mem^c) is a model of some (concrete) instantiation of a.

Definition 6 (Representing Concrete by Abstract States) Let $c = (CS^c, KB^c, AL^c, PT^c)$ be a concrete state where CS^c uses the functions $LV_1^c, \ldots, LV_n^c$. For every $\mathtt{x} \in \mathcal{V}_\mathcal{P}$ where $LV_i^c(\mathtt{x})$ is defined, let $as^c(\mathtt{x}_i) = n$ for the number $n \in \mathbb{Z}$ with $\models \langle c \rangle_{FO} \Rightarrow LV_i^c(\mathtt{x}) = n$.

For $n \in \mathbb{N}_{>0}$, the function $mem^c(n)$ is defined iff there exists a $w_1 \hookrightarrow_{\mathtt{i8}} w_2 \in PT^c$ such that $\models \langle c \rangle_{FO} \Rightarrow w_1 = n$. Let $\models \langle c \rangle_{FO} \Rightarrow w_2 = k$ for $k \in [-2^7, 2^7 - 1]$. Then we have $mem^c(n) = k$ if $k \geq 0$ and $mem^c(n) = k + 2^8$ if $k < 0$.

We say that an abstract state $a = ([(p_1, LV_1^a, AL_1^a), \ldots, (p_n, LV_n^a, AL_n^a)], KB^a, AL^a, PT^a)$ *represents* a concrete state $c = ([(p_1, LV_1^c, AL_1^c), \ldots, (p_n, LV_n^c, AL_n^c)], KB^c, AL^c, PT^c)$ 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 (†) from the `strlen` program represents all concrete states $c = ([((\mathtt{entry}, 0), LV_1, \{\})], KB, AL, PT)$ where mem^c stores a string at the address $as^c(\mathtt{str}_1)$.[5]

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 given program. For this, we present symbolic execution rules for some of the most important LLVM instructions. We start with the rules for the LLVM instructions in our `strlen` example in Sect. 3.1. In Sect. 3.2, we present rules for a more advanced example including memory allocation and function calls.

While there already exist approaches for symbolic execution of C or LLVM (e.g., in the tools KLEE [18] and Ufo [1]), our new abstract domain is particularly suitable for tracking explicit information about memory allocations and the contents of memory, allowing a fully

[3] A corresponding representation could also be defined for big-endian layout. This layout information is necessary to decide which concrete states are represented by abstract states, but it is not used when constructing symbolic execution graphs (i.e., our remaining approach is independent of such layout information).

[4] We identify *sets* of first-order formulas $\{\varphi_1, ..., \varphi_n\}$ with their conjunction $\varphi_1 \wedge ... \wedge \varphi_n$. Thus, CS is identified with the set resp. with the conjunction of the equations $\bigcup_{1 \leq i \leq n} \{\mathtt{x}_i = LV_i(\mathtt{x}) \mid \mathtt{x} \in \mathcal{V}_\mathcal{P}, LV_i(\mathtt{x}) \text{ is defined}\}$. Moreover, we wrote $(true * (\ast_{\varphi \in AL^*} \ \langle \varphi \rangle_{SL}))$ to ensure that this part of the formula is *true* if $AL^* = \varnothing$.

[5] The reason is that then there is an address $end \in \mathbb{N}_{>0}$ with $end \geq as^c(\mathtt{str}_1)$ such that $mem^c(end) = 0$ and mem^c is defined for all numbers between $as^c(\mathtt{str}_1)$ and end. Hence if a is the state in (†), then $mem^c \models \sigma(\langle a \rangle_{SL})$ holds for any instantiation σ with $\sigma(u_{\mathtt{str}}) = as^c(\mathtt{str}_1)$, $\sigma(v_{end}) = end$, and $\sigma(z) = 0$.

automated analysis of programs with direct memory access and pointer arithmetic. Most other existing tools cannot successfully analyze termination of such programs fully automatically without the specification of invariants by the user. In particular, we also have rules for refining and generalizing abstract states. This is needed to obtain *finite* symbolic execution graphs that represent all possible executions. We present our algorithm to generalize states in Sect. 3.3.

3.1 Basic Symbolic Execution Rules

Our analysis starts with the set of initial states that one wants to analyze for termination, e.g., all states where `str` points to a *string*. So in our example, we start with the abstract state (†). Fig. 1 depicts the symbolic execution graph for `strlen`. Here, we omitted the component $AL = \{[\![u_{\texttt{str}}, v_{end}]\!]\}$ for the global allocation list, which stays the same in all states in this example. We also abbreviated parts of *CS*, *KB*, and *PT* by "...". Instead of $v_{end} \hookrightarrow_{\texttt{i8}} z$ and $z = 0$, we directly wrote $v_{end} \hookrightarrow 0$, etc.

The function `strlen` starts with loading the character at address `str` to `c0`. Let $p\!:\!ins$ denote that *ins* is the instruction at position p. Our first rule handles the case p:"`x = load ty* ad`", i.e., the value of type `ty` at the address `ad` is assigned to the variable `x`. In our rules, let a always denote the state *before* the execution step (i.e., above the horizontal line of the rule). Moreover, we write $\langle a \rangle$ instead of $\langle a \rangle_{FO}$. As each memory cell stores one byte, in the `load`-rule we first have to check whether the addresses $\texttt{ad}, \ldots, \texttt{ad} + size(\texttt{ty}) - 1$ are allocated, i.e., whether there is a $[\![v_1, v_2]\!] \in AL^*$ such that $\langle a \rangle \Rightarrow (v_1 \leq LV_1(\texttt{ad}) \wedge LV_1(\texttt{ad}) + size(\texttt{ty}) - 1 \leq v_2)$ is valid. Then, we reach a new state where the previous position $p = (\texttt{b}, i)$ is updated to the position $p^+ = (\texttt{b}, i+1)$ of the next instruction in the same basic block, and we set $LV_1(\texttt{x}) = w$ for a fresh $w \in \mathcal{V}_{sym}$. Here we write $LV_1[\texttt{x} := w]$ for the function where $(LV_1[\texttt{x} := w])(\texttt{x}) = w$ and for $\texttt{y} \neq \texttt{x}$, we have $(LV_1[\texttt{x} := w])(\texttt{y}) = LV_1(\texttt{y})$. Moreover, we add $LV_1(\texttt{ad}) \hookrightarrow_{\texttt{ty}} w$ to *PT*. Thus, if *PT* already contained a formula $LV_1(\texttt{ad}) \hookrightarrow_{\texttt{ty}} u$, then $\langle a \rangle$ implies $w = u$. We used this rule to obtain *B* from *A* in Fig. 1.

In memory access instructions like `load`, one can also specify an optional *alignment* `al` which indicates that the respective addresses are divisible by `al`. This alignment information is generated by the LLVM code emitter (e.g., by the compiler from C to LLVM). It is a hint to the code generator (which transforms LLVM code into machine code) that the address will be at the specified alignment. The code generator may use this information for optimizations.

Note in the rules that LV_1 is a partial function, i.e., LV_1 may not be defined for all $\texttt{x} \in \mathcal{V}_{\mathcal{P}}$. But according to [53], in well-formed LLVM programs all uses of a variable are dominated by its definition. So $LV_1(\texttt{x})$ is always defined when we read from `x` during symbolic execution.

`load` **from allocated memory** (p : **"**`x = load ty* ad [, align al]`**" with** $\texttt{x}, \texttt{ad} \in \mathcal{V}_{\mathcal{P}}$**,** $\texttt{al} \in \mathbb{N}$)

$$\frac{((p,\ LV_1,\ AL_1) \cdot CS,\ KB,\ AL,\ PT)}{((p^+,\ LV_1[\texttt{x} := w],\ AL_1) \cdot CS,\ KB,\ AL,\ PT \cup \{LV_1(\texttt{ad}) \hookrightarrow_{\texttt{ty}} w\})} \quad \text{if}$$

- there is $[\![v_1, v_2]\!] \in AL^*$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV_1(\texttt{ad}) \wedge LV_1(\texttt{ad}) + size(\texttt{ty}) - 1 \leq v_2)$,
- $\models \langle a \rangle \Rightarrow (LV_1(\texttt{ad}) \bmod \texttt{al} = 0)$, if an alignment $\texttt{al} \geq 1$ is specified,
- $w \in \mathcal{V}_{sym}$ is fresh

In a similar way, we formulate a rule for instructions that `store` a value at some address in the memory. The instruction "`store ty` t`, ty* ad`" stores the value t of type `ty` at the address `ad`. Again, we check whether $LV_1(\texttt{ad}), \ldots, LV_1(\texttt{ad}) + size(\texttt{ty}) - 1$ are addresses in an allocated part of the memory. The information that `ad` now points to t is added to the set *PT*. All other information in *PT* that is not influenced by this change is kept.[6]

[6] For any terms, "$[\![t_1, t_2]\!] \perp [\![\overline{t_1}, \overline{t_2}]\!]$" is a shorthand for $t_2 < \overline{t_1} \vee \overline{t_2} < t_1$.

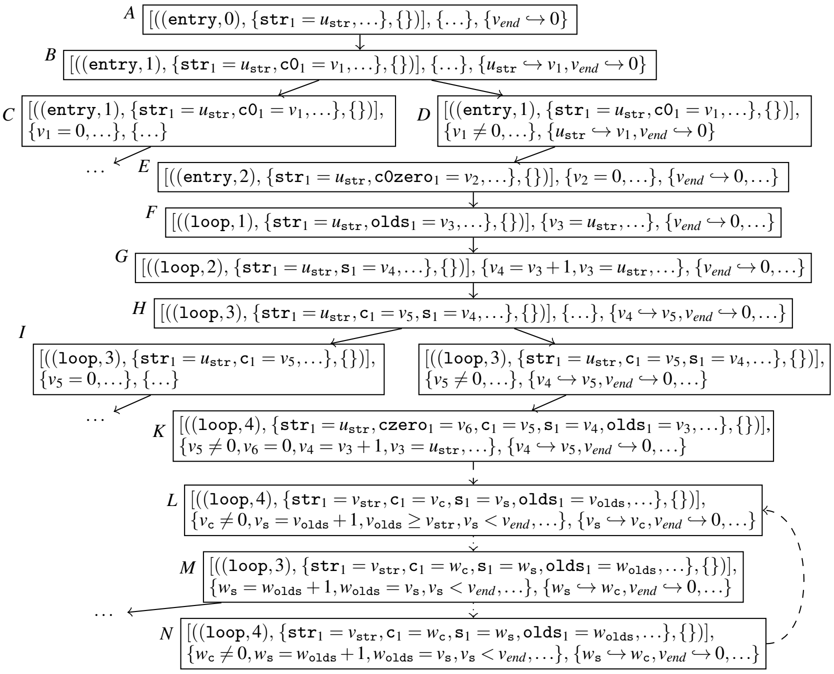
A
$[((\texttt{entry},0),\{\texttt{str}_1=u_{\texttt{str}},\ldots\},\{\})],\{\ldots\},\{v_{end}\hookrightarrow 0\}$
B
$[((\texttt{entry},1),\{\texttt{str}_1=u_{\texttt{str}},\texttt{c0}_1=v_1,\ldots\},\{\})],\{\ldots\},\{u_{\texttt{str}}\hookrightarrow v_1,v_{end}\hookrightarrow 0\}$
C
$[((\texttt{entry},1),\{\texttt{str}_1=u_{\texttt{str}},\texttt{c0}_1=v_1,\ldots\},\{\})],$ $\{v_1=0,\ldots\},\{\ldots\}$
D
$[((\texttt{entry},1),\{\texttt{str}_1=u_{\texttt{str}},\texttt{c0}_1=v_1,\ldots\},\{\})],$ $\{v_1\neq 0,\ldots\},\{u_{\texttt{str}}\hookrightarrow v_1,v_{end}\hookrightarrow 0\}$
...
E
$[((\texttt{entry},2),\{\texttt{str}_1=u_{\texttt{str}},\texttt{c0zero}_1=v_2,\ldots\},\{\})],\{v_2=0,\ldots\},\{v_{end}\hookrightarrow 0,\ldots\}$
F
$[((\texttt{loop},1),\{\texttt{str}_1=u_{\texttt{str}},\texttt{olds}_1=v_3,\ldots\},\{\})],\{v_3=u_{\texttt{str}},\ldots\},\{v_{end}\hookrightarrow 0,\ldots\}$
G
$[((\texttt{loop},2),\{\texttt{str}_1=u_{\texttt{str}},\texttt{s}_1=v_4,\ldots\},\{\})],\{v_4=v_3+1,v_3=u_{\texttt{str}},\ldots\},\{v_{end}\hookrightarrow 0,\ldots\}$
H
$[((\texttt{loop},3),\{\texttt{str}_1=u_{\texttt{str}},\texttt{c}_1=v_5,\texttt{s}_1=v_4,\ldots\},\{\})],\{\ldots\},\{v_4\hookrightarrow v_5,v_{end}\hookrightarrow 0,\ldots\}$
I
$[((\texttt{loop},3),\{\texttt{str}_1=u_{\texttt{str}},\texttt{c}_1=v_5,\ldots\},\{\})],$ $\{v_5=0,\ldots\},\{\ldots\}$
J
$[((\texttt{loop},3),\{\texttt{str}_1=u_{\texttt{str}},\texttt{c}_1=v_5,\texttt{s}_1=v_4,\ldots\},\{\})],$ $\{v_5\neq 0,\ldots\},\{v_4\hookrightarrow v_5,v_{end}\hookrightarrow 0,\ldots\}$
...
K
$[((\texttt{loop},4),\{\texttt{str}_1=u_{\texttt{str}},\texttt{czero}_1=v_6,\texttt{c}_1=v_5,\texttt{s}_1=v_4,\texttt{olds}_1=v_3,\ldots\},\{\})],$ $\{v_5\neq 0,v_6=0,v_4=v_3+1,v_3=u_{\texttt{str}},\ldots\},\{v_4\hookrightarrow v_5,v_{end}\hookrightarrow 0,\ldots\}$
L
$[((\texttt{loop},4),\{\texttt{str}_1=v_{\texttt{str}},\texttt{c}_1=v_{\texttt{c}},\texttt{s}_1=v_{\texttt{s}},\texttt{olds}_1=v_{\texttt{olds}},\ldots\},\{\})],$ $\{v_{\texttt{c}}\neq 0,v_{\texttt{s}}=v_{\texttt{olds}}+1,v_{\texttt{olds}}\geq v_{\texttt{str}},v_{\texttt{s}}<v_{end},\ldots\},\{v_{\texttt{s}}\hookrightarrow v_{\texttt{c}},v_{end}\hookrightarrow 0,\ldots\}$
M
$[((\texttt{loop},3),\{\texttt{str}_1=v_{\texttt{str}},\texttt{c}_1=w_{\texttt{c}},\texttt{s}_1=w_{\texttt{s}},\texttt{olds}_1=w_{\texttt{olds}},\ldots\},\{\})],$ $\{w_{\texttt{s}}=w_{\texttt{olds}}+1,w_{\texttt{olds}}=v_{\texttt{s}},v_{\texttt{s}}<v_{end},\ldots\},\{w_{\texttt{s}}\hookrightarrow w_{\texttt{c}},v_{end}\hookrightarrow 0,\ldots\}$
...
N
$[((\texttt{loop},4),\{\texttt{str}_1=v_{\texttt{str}},\texttt{c}_1=w_{\texttt{c}},\texttt{s}_1=w_{\texttt{s}},\texttt{olds}_1=w_{\texttt{olds}},\ldots\},\{\})],$ $\{w_{\texttt{c}}\neq 0,w_{\texttt{s}}=w_{\texttt{olds}}+1,w_{\texttt{olds}}=v_{\texttt{s}},v_{\texttt{s}}<v_{end},\ldots\},\{w_{\texttt{s}}\hookrightarrow w_{\texttt{c}},v_{end}\hookrightarrow 0,\ldots\}$

Fig. 1 Symbolic execution graph for `strlen`

`store` **to allocated memory** (p : **"**`store ty` t`, ty* ad [, align al]`**",** $t\in\mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**,** $\texttt{ad}\in\mathcal{V}_\mathcal{P}$**,** $\texttt{al}\in\mathbb{N}$**)**

$$\frac{((p,\ LV_1,\ AL_1)\cdot CS,\ KB,\ AL,\ PT)}{((p^+,\ LV_1,\ AL_1)\cdot CS,\ KB\cup\{w=LV_1(t)\},\ AL,\ PT'\cup\{LV_1(\texttt{ad})\hookrightarrow_{\texttt{ty}} w\})}\quad\text{if}$$

- there is $[\![v_1,\,v_2]\!]\in AL^*$ with $\models\langle a\rangle\Rightarrow(v_1\leq LV_1(\texttt{ad})\wedge LV_1(\texttt{ad})+size(\texttt{ty})-1\leq v_2)$,
- $PT'=\{(w_1\hookrightarrow_{\texttt{sy}} w_2)\in PT\mid\ \models\langle a\rangle\Rightarrow(\,[\![LV_1(\texttt{ad}),LV_1(\texttt{ad})+size(\texttt{ty})-1]\!]\perp[\![w_1,\,w_1+size(\texttt{sy})-1]\!]\,)\}$,
- $\models\langle a\rangle\Rightarrow(LV_1(\texttt{ad})\ \mathrm{mod}\ \texttt{al}=0)$, if an alignment $\texttt{al}\geq 1$ is specified,
- $w\in\mathcal{V}_{sym}$ is fresh

If `load` or `store` accesses a non-allocated address or if the address does not correspond to the specified alignment, then memory safety is violated and we reach the *ERR* state.

`load` **or** `store` **on unallocated memory** (p : **"**`x = load ty* ad [, align al]`**" with** $\texttt{x},\texttt{ad}\in\mathcal{V}_\mathcal{P}$ **and** $\texttt{al}\in\mathbb{N}$**, or** p : **"**`store ty` t`, ty* ad [, align al]`**" with** $t\in\mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**,** $\texttt{ad}\in\mathcal{V}_\mathcal{P}$**, and** $\texttt{al}\in\mathbb{N}$**)**

$$\frac{((p,\ LV_1,\ AL_1)\cdot CS,\ KB,\ AL,\ PT)}{ERR}\quad\text{if}$$

there is no $[\![v_1,\,v_2]\!]\in AL^*$ with $\models\langle a\rangle\Rightarrow(v_1\leq LV_1(\texttt{ad})\wedge LV_1(\texttt{ad})+size(\texttt{ty})-1\leq v_2)$

`load` **or** `store` **with unsafe alignment** (p : **"**`x = load ty* ad, align al`**" with** $\texttt{x},\texttt{ad}\in\mathcal{V}_\mathcal{P}$ **and** $\texttt{al}\in\mathbb{N}_{>0}$**, or** p : **"**`store ty` t`, ty* ad, align al`**" with** $t\in\mathcal{V}_\mathcal{P}\cup\mathbb{Z}$**,** $\texttt{ad}\in\mathcal{V}_\mathcal{P}$**, and** $\texttt{al}\in\mathbb{N}_{>0}$**)**

$$\frac{((p,\ LV_1,\ AL_1)\cdot CS,\ KB,\ AL,\ PT)}{ERR}\quad\text{if}\not\models\langle a\rangle\Rightarrow(LV_1(\texttt{ad})\ \mathrm{mod}\ \texttt{al}=0)$$

The instructions `icmp` and `br` in `strlen`'s `entry` block check if the first character `c0` is `0`. In that case, we have reached the end of the string and jump to the block `done`. Thus, we now introduce rules for integer comparison. For "`x = icmp eq ty` t_1`,` t_2", we check if the state contains enough information to decide whether the values t_1 and t_2 of type `ty` are equal. In that case, the value `1` resp. `0` (i.e., *true* resp. *false*) is assigned to `x`.

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

$$\frac{((p, LV_1, AL_1) \cdot CS,\ KB,\ AL,\ PT)}{((p^+, LV_1[\mathtt{x} := w],\ AL_1) \cdot CS,\ KB \cup \{w = 1\},\ AL,\ PT)} \quad \begin{array}{l}\text{if} \models \langle a \rangle \Rightarrow (LV_1(t_1) = LV_1(t_2))\\ \text{and } w \in \mathcal{V}_{sym} \text{ is fresh}\end{array}$$

$$\frac{((p, LV_1, AL_1) \cdot CS,\ KB,\ AL,\ PT)}{((p^+, LV_1[\mathtt{x} := w],\ AL_1) \cdot CS,\ KB \cup \{w = 0\},\ AL,\ PT)} \quad \begin{array}{l}\text{if} \models \langle a \rangle \Rightarrow (LV_1(t_1) \neq LV_1(t_2))\\ \text{and } w \in \mathcal{V}_{sym} \text{ is fresh}\end{array}$$

Other integer comparisons (for $<$, $\leq$, $\ldots$) are handled analogously. Note that LLVM always represents integers in two's complement, as does the knowledge base in our states. However, some instructions explicitly consider values in an unsigned way, and this needs to be reflected in our evaluation rules. As an example, suppose that $\models \langle a \rangle \Rightarrow v = -2^7 \wedge w = 2^7 - 1$. Then signed comparison yields $v < w$, but unsigned comparison yields $v > w$, because v is stored as $(1\,0\,0\,0\,0\,0\,0\,0)$, whereas w is stored as $(0\,1\,1\,1\,1\,1\,1\,1)$. So for an unsigned comparison, we check whether the two values to be compared are either both positive or both negative, i.e., have the same sign. In this case, the comparison on the unsigned interpretation coincides with the signed comparison. For different signs, negative numbers (like $v = -2^7$) are always *greater* than positive ones (like $w = 2^7 - 1$). As an example, the following rule illustrates the affirmative case ($w = 1$) of unsigned less-or-equal (`ule`).

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

$$\frac{((p, LV_1, AL_1) \cdot CS,\ KB,\ AL,\ PT)}{((p^+, LV_1[\mathtt{x} := w],\ AL_1) \cdot CS,\ KB \cup \{w = 1\},\ AL,\ PT)}$$

if $\models \langle a \rangle \Rightarrow (LV_1(t_1) \leq LV_1(t_2)) \wedge (sgn(LV_1(t_1)) = sgn(LV_1(t_2))) \quad \vee \quad (LV_1(t_1) \geq 0) \wedge (LV_1(t_2) < 0)$
and $w \in \mathcal{V}_{sym}$ is fresh

The rules for `icmp` are only applicable if *KB* contains enough information to evaluate the respective condition. Otherwise, a case analysis needs to be performed, i.e., one has to *refine* the abstract state by extending its knowledge base. This is done by the following rule, which transforms an abstract state into *two* new ones.[7]

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

$$\frac{((p,\ LV_1,\ AL_1) \cdot CS,\ KB,\ AL,\ PT)}{((p,\ LV_1,\ AL_1) \cdot CS,\ KB \cup \{\varphi\},\ AL,\ PT) \mid ((p,\ LV_1,\ AL_1) \cdot CS,\ KB \cup \{\neg\varphi\},\ AL,\ PT)}$$

if $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$ and φ is $LV_1(t_1) = LV_1(t_2)$

In state B of Fig. 1, we evaluate "`c0zero = icmp eq i8 c0, 0`", i.e., we check if the first character `c0` of the string `str` is `0`. Since this cannot be inferred from B's knowledge base, we refine B to the successor states C and D and call the edges from B to C and D *refinement edges*. In D, we have $\mathtt{c0} = v_1$ and $v_1 \neq 0$. Thus, the `icmp`-rule yields E where $\mathtt{c0zero} = v_2$ and $v_2 = 0$. We do not display the successors of C that lead to a program end.

The next instruction in our example is "`br i1 c0zero, label done, label loop`", a conditional jump (or branch) to another block. Let us first consider a similar, but simpler case. The instruction "`br label b`$_{next}$" means that the execution has to continue with the first instruction in the block `b`$_{next}$. When execution moves from one block to another, in the

[7] Analogous refinement rules can also be used for other conditional LLVM instructions, e.g., conditional jumps with `br` or other cases of `icmp`.

new target block one first evaluates the phi instructions that may be present at its beginning. These instructions are needed due to the static single assignment form of LLVM and initialize the variables in the target block depending on from which block we are entering the target block. Such phi instructions may only occur at the beginning of a block, i.e., every block starts with a (possibly empty) sequence of phi instructions. A phi instruction has the form "x = phi ty [t_1,b$_1$],…,[t_n,b$_n$]", meaning that if the previous block was b$_j$, then the value t_j is assigned to x. All $t_1,\ldots,t_n$ must have type ty. A peculiarity of phi instructions is that all phi instructions in the same block are executed atomically together. So all local variables occurring in $t_1,\ldots,t_n$ still have the values that they had *before* entering the new target block.

To handle phi in combination with the br instruction at the end of the previous block, we introduce an auxiliary function *firstNonPhi*. For any block b, *firstNonPhi*(b) is the index of the first non-phi instruction in b. Moreover, we define the function *computePhi* to implement the parallel execution of all phi statements "x^1 = phi ty^1 [t^1_1,b$^1_1$], …, [$t^1_{n^1}$,b$^1_{n^1}$]", …, "x^m = phi ty^m [t^m_1,b$^m_1$], …, [$t^m_{n^m}$,b$^m_{n^m}$]" at the start of the block b$_{next}$. Its arguments are the current values LV of the local variables, the current block b$_j$, and the target block b$_{next}$, and it returns a pair $(LV', KB_{\texttt{phi}})$, where LV' reflects the updated local variables and $KB_{\texttt{phi}}$ contains information on the new symbolic variables introduced in LV':

$$\mathit{computePhi}(LV, \texttt{b}_j, \texttt{b}_{next}) = (LV[\texttt{x}^1 := w^1, \ldots, \texttt{x}^m := w^m],\ \ \{w^1 = LV(t^1_j), \ldots, w^m = LV(t^m_j)\}),$$

where $w^1,\ldots,w^m \in \mathcal{V}_{sym}$ are fresh. Now we can define a rule that allows us to perform an unconditional jump with br to a block b$_{next}$ and that executes b$_{next}$'s phi instructions.

unconditional br (*p* : "br label b$_{next}$" **with** b$_{next}$ $\in$ *Blks*)

$$\frac{(((\texttt{b}, i),\ LV_1,\ AL_1) \cdot CS,\ KB,\ AL,\ PT)}{(((\texttt{b}_{next}, j),\ LV'_1,\ AL_1) \cdot CS,\ KB \cup KB_{\texttt{phi}},\ AL,\ PT)}$$

if $(LV'_1, KB_{\texttt{phi}}) = \mathit{computePhi}(LV_1, \texttt{b}, \texttt{b}_{next})$ and $j = \mathit{firstNonPhi}(\texttt{b}_{next})$

For conditional branches "br i1 t, label b$_1$, label b$_2$", one has to check whether the current state contains enough information to conclude that t is 1 (i.e., *true*) or 0 (i.e., *false*). Then the evaluation continues after the phi instructions of block b$_1$ resp. b$_2$.

conditional br (*p* : "br i1 t, label b$_1$, label b$_2$" **with** $t \in \mathcal{V}_{\mathcal{P}} \cup \{0,1\}$ **and** b$_1$, b$_2$ $\in$ *Blks*)

$$\frac{(((\texttt{b}, i),\ LV_1,\ AL_1) \cdot CS,\ KB,\ AL,\ PT)}{(((\texttt{b}_1, j_1),\ LV'_1,\ AL_1) \cdot CS,\ KB \cup KB_{\texttt{phi}},\ AL,\ PT)}$$

if $\models \langle a \rangle \Rightarrow (LV_1(t) = 1)$, $(LV'_1, KB_{\texttt{phi}}) = \mathit{computePhi}(LV_1, \texttt{b}, \texttt{b}_1)$, $j_1 = \mathit{firstNonPhi}(\texttt{b}_1)$

$$\frac{(((\texttt{b}, i),\ LV_1,\ AL_1) \cdot CS,\ KB,\ AL,\ PT)}{(((\texttt{b}_2, j_2),\ LV'_1,\ AL_1) \cdot CS,\ KB \cup KB_{\texttt{phi}},\ AL,\ PT)}$$

if $\models \langle a \rangle \Rightarrow (LV_1(t) = 0)$, $(LV'_1, KB_{\texttt{phi}}) = \mathit{computePhi}(LV_1, \texttt{b}, \texttt{b}_2)$, $j_2 = \mathit{firstNonPhi}(\texttt{b}_2)$

With the br instruction, one now jumps to the loop block in State F. Note that we simplified the equalities resulting from *computePhi* in F, to avoid renaming in the presentation.

The strlen function traverses the string using a pointer s, and the loop terminates when s eventually reaches the last memory cell of the string (containing 0). Then one jumps to done, converts the pointers s and str to integers, and returns their difference. To perform the required pointer arithmetic, "bd = getelementptr ty* ad,in t" increases ad by the size of t elements of type ty (i.e., by $\mathit{size}(\texttt{ty}) \cdot t$) and assigns this address to bd.[8]

[8] Since we do not consider struct data structures in this paper, we disregard getelementptr instructions with more than two parameters. Note that getelementptr instructions with just two parameters suffice for several levels of de-referencing (where memory has to be accessed after each getelementptr instruction).

`getelementptr` (p : "`bd = getelementptr ty* ad, i`n t", $\texttt{ad}, \texttt{bd} \in \mathcal{V}_\mathcal{P}$, $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{((p,\ LV_1,\ AL_1)\cdot CS,\ KB,\ AL,\ PT)}{((p^+,\ LV_1[\texttt{bd}_1 := w],\ AL_1)\cdot CS,\ KB\cup\{w = LV_1(\texttt{ad}) + size(\texttt{ty})\cdot LV_1(t)\},\ AL,\ PT)} \quad \text{if } w \in \mathcal{V}_{sym} \text{ is fresh}$$

In Fig. 1, this rule is used for the step from F to G, which implies $\texttt{s} = \texttt{str} + 1$. In the step to H, the character at address `s` is loaded to `c`. To ensure memory safety, the `load`-rule checks that `s` is in an allocated part of the memory (i.e., that $u_\texttt{str} \leq u_\texttt{str} + 1 \leq v_{end}$). This holds because $\langle G \rangle$ implies $u_\texttt{str} \leq v_{end}$ and $u_\texttt{str} \neq v_{end}$ (as $u_\texttt{str} \hookrightarrow v_1, v_{end} \hookrightarrow 0 \in PT$ and $v_1 \neq 0 \in KB$). Finally, we check whether `c` is `0`. We again perform a refinement which yields the states I and J. State J corresponds to the case $\texttt{c} \neq \texttt{0}$ and thus, we obtain $\texttt{czero} = \texttt{0}$ in K.

Finally, we present rules for the instructions `ptrtoint` and `sub` that are used in the block `done` of the `strlen` example. The `ptrtoint` instruction simply converts pointers to integers and is needed to perform subsequent arithmetic operations on them (e.g., to subtract one address from another in the `strlen` algorithm). In a similar way, we also have rules to handle other LLVM instructions for casting between pointers and different types of integers.

`ptrtoint` (p : "`x = ptrtoint ty* ad to i`n" **with** $\texttt{x}, \texttt{ad} \in \mathcal{V}_\mathcal{P}$)

$$\frac{((p,\ LV_1,\ AL_1)\cdot CS,\ KB,\ AL,\ PT)}{((p^+,\ LV_1[\texttt{x} := w],\ AL_1)\cdot CS,\ KB\cup\{w = LV_1(\texttt{ad})\},\ AL,\ PT)} \quad \text{if } w \in \mathcal{V}_{sym} \text{ is fresh}$$

In `sub` instructions of the form "`x = sub ty` t_1`,` t_2", both t_1 and t_2 must have the type `ty` and the variable `x` also gets this type. We use similar rules to handle other LLVM instructions for other arithmetic, Boolean, and bit manipulation operations.

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

$$\frac{((p,\ LV_1,\ AL_1)\cdot CS,\ KB,\ AL,\ PT)}{((p^+,\ LV_1[\texttt{x} := w],\ AL_1)\cdot CS,\ KB\cup\{w = LV_1(t_1) - LV_1(t_2)\},\ AL,\ PT)} \quad \text{if } w \in \mathcal{V}_{sym} \text{ is fresh}$$

3.2 Advanced Symbolic Execution Rules

Now we also present rules that allow allocation of memory, function calls, and manipulation of larger memory chunks. We start with a rule for the `alloca` statement. The instruction "`x = alloca ty, i`n t" allocates memory for t elements of the type `ty`. Here, `x` is an identifier from $\mathcal{V}_\mathcal{P}$ of type `ty*` and t is either an identifier or a natural number. Thus, a new interval is allocated (i.e., the allocation list AL_1 of the current stack frame is extended by $[\![v_1,\ v_2]\!]$ for fresh symbolic variables v_1, v_2) and KB is extended by $v_2 = v_1 + size(\texttt{ty}) \cdot LV_1(t) - 1$. Moreover, the address of the first memory cell in the newly allocated block is assigned to `x`. Thus, we update LV_1 by $\texttt{x} = v_1$. Again, the code emitter may have added an alignment `al`. In contrast to `load` and `store`, it is not designed as a hint for the code generator but as a requirement that the result of the allocation must be at least `al`-aligned. If no alignment is specified or $\texttt{al} = 0$, one uses the alignment $\text{align}(\texttt{ty})$ specified by the ABI (application binary interface) of the target machine and operating system. The code emitter writes information on the ABI alignment of pointers and the most common integer, vector, and floating point types in the header of the LLVM program. For all remaining types, the ABI alignment is computed from these given alignments. Allocating 0 bytes results in undefined behavior, which may therefore violate memory safety and affect the termination behavior.

`alloca` (p: "`x = alloca ty, i`n t `[, align al]`" **with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$**,** $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$**, and** $\mathtt{al} \in \mathbb{N}$)

$$\frac{((p,\, LV_1,\, AL_1)\cdot CS,\, KB,\, AL,\, PT)}{ERR} \quad \text{if} \not\models \langle a\rangle \Rightarrow (LV_1(t) > 0)$$

$$\frac{((p,\, LV_1,\, AL_1)\cdot CS,\, KB,\, AL,\, PT)}{((p^+,\, LV_1[\mathtt{x} := v_1],\, AL_1 \cup \{[\![v_1,\, v_2]\!]\})\cdot CS,\, KB' \cup \{v_2 = v_1 + size(\mathtt{ty})\cdot LV_1(t) - 1\},\, AL,\, PT)} \quad \text{if}$$

- we have $\models \langle a\rangle \Rightarrow (LV_1(t) > 0)$,
- $KB' = KB \cup \{v_1 \bmod c = 0\}$, where $c = \mathtt{al}$, if $\mathtt{al} \geq 1$ is specified, or else $c = \text{align}(\mathtt{ty})$,
- $v_1, v_2 \in \mathcal{V}_{sym}$ are fresh

Note that `alloca` is used to allocate memory on the stack, whereas `malloc` and `free` allocate and release memory on the heap. The latest versions of LLVM do not have built-in `malloc` or `free` instructions anymore, but one has to call them as external functions (provided by the standard C library). For LLVM programs that call `malloc` or `free`, we use the following two inference rules. The rule for `malloc` mainly differs from the rule for `alloca` by placing the newly allocated memory region into the global allocation list instead of the allocation list of the current stack frame. Here, "`x = call i8* @malloc(i`n t`)`" allocates t bytes and the address of the first memory cell in this block is assigned to `x`. Depending on the processor architecture of the target machine, the allocated memory is 8-byte or 16-byte aligned. Our rule for `malloc` currently does not take into account that `malloc` may also return `NULL` without allocating any memory. However, we could easily support this by introducing a corresponding second successor state for this possible outcome.

`malloc` (p: "`x = call i8* @malloc(i`n t`)`" **with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ **and** $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{((p,\, LV_1,\, AL_1)\cdot CS,\, KB,\, AL,\, PT)}{((p^+,\, LV_1[\mathtt{x} := v_1],\, AL_1)\cdot CS,\, KB' \cup \{v_2 = v_1 + LV_1(t) - 1\},\, AL \cup \{[\![v_1,\, v_2]\!]\},\, PT)} \quad \text{if}$$

- we have $\models \langle a\rangle \Rightarrow (LV_1(t) > 0)$,
- $KB' = KB \cup \{v_1 \bmod c = 0\}$, where $c = 8$ for 32-bit platforms and $c = 16$ for 64-bit platforms,
- $v_1, v_2 \in \mathcal{V}_{sym}$ are fresh

LLVM does not explicitly distinguish between the heap and stack, but applies the same memory model for both (using `load` and `store`). However, memory acquired by `alloca` is automatically released at the end of the function in which it was allocated, while memory acquired by `malloc` has to be released explicitly by calling `free`. The instruction "`call void @free(i8*` t`)`" releases the allocated memory block starting at the address t. Moreover, it deletes those entries from PT which are known to correspond to this memory block. Calling `free` on `NULL` does not change the state. If `free` is called with an address that is neither the beginning of an allocated memory block in the global allocation list (of memory allocated by `malloc`) nor `NULL`, then memory safety is violated and we reach the state ERR.

`free` (p: "`call void @free(i8*` t`)`" **with** $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{((p,\, LV_1,\, AL_1)\cdot CS,\, KB,\, AL \uplus \{[\![v_1,\, v_2]\!]\},\, PT)}{((p^+,\, LV_1,\, AL_1)\cdot CS,\, KB,\, AL,\, PT')} \quad \text{if}$$

- $v_1, v_2 \in \mathcal{V}_{sym}$,
- $\models \langle a\rangle \Rightarrow (LV_1(t) = v_1)$,
- PT' results from PT by removing all $v \hookrightarrow_{\mathtt{ty}} w$ where $\models \langle a\rangle \Rightarrow v_1 \leq v \wedge v \leq v_2$

$$\frac{((p,\, LV_1,\, AL_1)\cdot CS,\, KB,\, AL,\, PT)}{((p^+,\, LV_1,\, AL_1)\cdot CS,\, KB,\, AL,\, PT)} \quad \text{if} \models \langle a\rangle \Rightarrow (LV_1(t) = 0)$$

$$\frac{((p,\, LV_1,\, AL_1)\cdot CS,\, KB,\, AL,\, PT)}{ERR} \quad \text{if}$$

- $\not\models \langle a\rangle \Rightarrow (LV_1(t) = 0)$,
- there is no $[\![v_1,\, v_2]\!] \in AL$ with $\models \langle a\rangle \Rightarrow (LV_1(t) = v_1)$

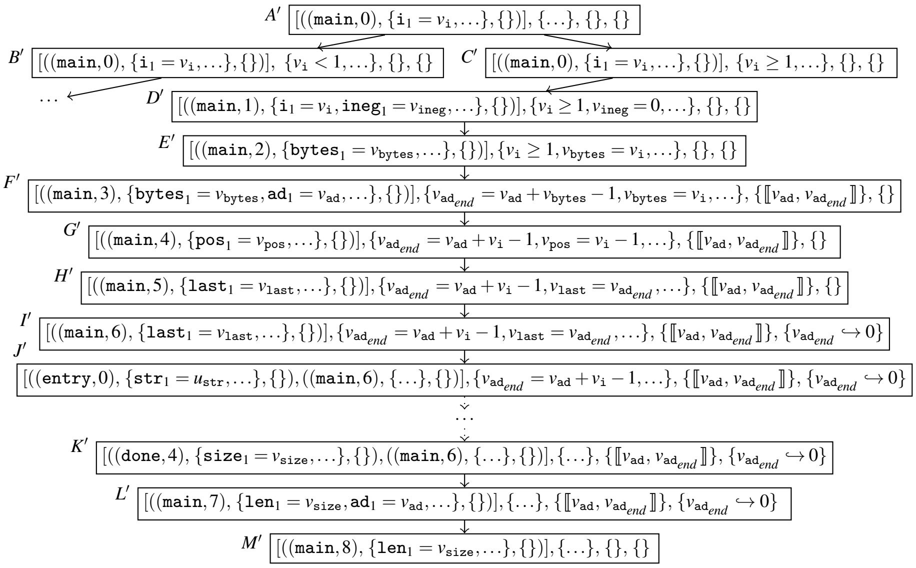
A' $[((\mathtt{main},0), \{\mathtt{i}_1 = v_{\mathtt{i}},\ldots\},\{\})], \{\ldots\}, \{\}, \{\}$
B' $[((\mathtt{main},0), \{\mathtt{i}_1 = v_{\mathtt{i}},\ldots\},\{\})], \{v_{\mathtt{i}} < 1,\ldots\}, \{\}, \{\}$
C' $[((\mathtt{main},0), \{\mathtt{i}_1 = v_{\mathtt{i}},\ldots\},\{\})], \{v_{\mathtt{i}} \geq 1,\ldots\}, \{\}, \{\}$
$\ldots$
D' $[((\mathtt{main},1), \{\mathtt{i}_1 = v_{\mathtt{i}}, \mathtt{ineg}_1 = v_{\mathtt{ineg}},\ldots\},\{\})], \{v_{\mathtt{i}} \geq 1, v_{\mathtt{ineg}} = 0,\ldots\}, \{\}, \{\}$
E' $[((\mathtt{main},2), \{\mathtt{bytes}_1 = v_{\mathtt{bytes}},\ldots\},\{\})], \{v_{\mathtt{i}} \geq 1, v_{\mathtt{bytes}} = v_{\mathtt{i}},\ldots\}, \{\}, \{\}$
F' $[((\mathtt{main},3), \{\mathtt{bytes}_1 = v_{\mathtt{bytes}}, \mathtt{ad}_1 = v_{\mathtt{ad}},\ldots\},\{\})], \{v_{\mathtt{ad}_{end}} = v_{\mathtt{ad}} + v_{\mathtt{bytes}} - 1, v_{\mathtt{bytes}} = v_{\mathtt{i}},\ldots\}, \{[\![v_{\mathtt{ad}}, v_{\mathtt{ad}_{end}}]\!]\}, \{\}$
G' $[((\mathtt{main},4), \{\mathtt{pos}_1 = v_{\mathtt{pos}},\ldots\},\{\})], \{v_{\mathtt{ad}_{end}} = v_{\mathtt{ad}} + v_{\mathtt{i}} - 1, v_{\mathtt{pos}} = v_{\mathtt{i}} - 1,\ldots\}, \{[\![v_{\mathtt{ad}}, v_{\mathtt{ad}_{end}}]\!]\}, \{\}$
H' $[((\mathtt{main},5), \{\mathtt{last}_1 = v_{\mathtt{last}},\ldots\},\{\})], \{v_{\mathtt{ad}_{end}} = v_{\mathtt{ad}} + v_{\mathtt{i}} - 1, v_{\mathtt{last}} = v_{\mathtt{ad}_{end}},\ldots\}, \{[\![v_{\mathtt{ad}}, v_{\mathtt{ad}_{end}}]\!]\}, \{\}$
I' $[((\mathtt{main},6), \{\mathtt{last}_1 = v_{\mathtt{last}},\ldots\},\{\})], \{v_{\mathtt{ad}_{end}} = v_{\mathtt{ad}} + v_{\mathtt{i}} - 1, v_{\mathtt{last}} = v_{\mathtt{ad}_{end}},\ldots\}, \{[\![v_{\mathtt{ad}}, v_{\mathtt{ad}_{end}}]\!]\}, \{v_{\mathtt{ad}_{end}} \hookrightarrow 0\}$
J' $[((\mathtt{entry},0), \{\mathtt{str}_1 = u_{\mathtt{str}},\ldots\},\{\}), ((\mathtt{main},6), \{\ldots\},\{\})], \{v_{\mathtt{ad}_{end}} = v_{\mathtt{ad}} + v_{\mathtt{i}} - 1,\ldots\}, \{[\![v_{\mathtt{ad}}, v_{\mathtt{ad}_{end}}]\!]\}, \{v_{\mathtt{ad}_{end}} \hookrightarrow 0\}$
$\ldots$
K' $[((\mathtt{done},4), \{\mathtt{size}_1 = v_{\mathtt{size}},\ldots\},\{\}), ((\mathtt{main},6), \{\ldots\},\{\})], \{\ldots\}, \{[\![v_{\mathtt{ad}}, v_{\mathtt{ad}_{end}}]\!]\}, \{v_{\mathtt{ad}_{end}} \hookrightarrow 0\}$
L' $[((\mathtt{main},7), \{\mathtt{len}_1 = v_{\mathtt{size}}, \mathtt{ad}_1 = v_{\mathtt{ad}},\ldots\},\{\})], \{\ldots\}, \{[\![v_{\mathtt{ad}}, v_{\mathtt{ad}_{end}}]\!]\}, \{v_{\mathtt{ad}_{end}} \hookrightarrow 0\}$
M' $[((\mathtt{main},8), \{\mathtt{len}_1 = v_{\mathtt{size}},\ldots\},\{\})], \{\ldots\}, \{\}, \{\}$

Fig. 2 Symbolic execution graph for `main`

To illustrate the rules for allocating and releasing memory, assume that we call the function `strlen` within a main function with a pointer to a memory area allocated by `malloc`. The symbolic execution graph for the corresponding LLVM program is depicted in Fig. 2. The first instruction is `icmp slt`, which checks if the function argument `i` in signed interpretation is less than 1 (`slt`). Since in state A', we do not have any information on `i`, we refine A' to the states B' and C'. C' is then evaluated to D', where the result of the comparison is assigned to `ineg`. Depending on the value of `ineg`, the `select` instruction assigns `1` or `i` to the variable `bytes`. In F', the call of `malloc` has been evaluated: the entry $[\![v_{\mathtt{ad}}, v_{\mathtt{ad}_{end}}]\!]$ is added to the global allocation list and in the knowledge base we keep the relationship between the start address $v_{\mathtt{ad}}$ and the end address $v_{\mathtt{ad}_{end}}$. In M', the allocated memory area is released again, leading to an empty global allocation list and an empty list PT at the end of the program. The transition from I' to J' corresponds to a call of the function `strlen` and the transition from K' to L' corresponds to a return from this function.

```
int main (int i)
  if (i < 1) i = 1;
  char* str = (char*) malloc(i * sizeof(char));
  str[i-1] = '\0';
  int len = strlen(str);
  free(str);
  return len;
```

```
define i32 @main(i32 i) {
main:  0: ineg = icmp slt i32 i, 1
       1: bytes = select i1 ineg, i32 1, i32 i
       2: ad = call i8* @malloc(i32 bytes)
       3: pos = add i32 bytes, -1
       4: last = getelementptr i8* ad, i32 pos
       5: store i8 0, i8* last
       6: len = call i32 @strlen(i8* ad)
       7: call void @free(i8* ad)
       8: ret i32 len}
```

The symbolic execution rules for the `select` instruction are analogous to the rules for `icmp`. The instructions `call` and `ret` for calling and returning from a function are needed when going beyond intraprocedural analysis. The rule for `call` pushes a new frame on the call stack whose position is the entry point of the called function and the argument values are assigned to its parameters. When the `ret` instruction is encountered, the top frame is popped from the stack again. For reasons of space, we only present the rules for non-`void` functions.

call (p : "`x = call ty @function(ty`$_1$ t_1`, ..., ty`$_n$ t_n`)`" **with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}, t_1, \ldots, t_n \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{((p,\, LV_1,\, AL_1) \cdot CS,\, KB,\, AL,\, PT)}{(((\mathtt{function.entry}, 0),\, LV_0,\, \{\}) \cdot (p,\, LV_1,\, AL_1) \cdot CS,\, KB',\, AL,\, PT)} \quad \text{if}$$

- `function.entry` is the entry block of `function`
- `function` is declared as $\mathtt{function}(\mathtt{ty}_1\ \mathtt{u}_1, \ldots, \mathtt{ty}_n\ \mathtt{u}_n)$,
- $w_1, \ldots, w_n \in \mathcal{V}_{sym}$ are fresh,
- $LV_0(\mathtt{u}_1) = w_1, \ldots, LV_0(\mathtt{u}_n) = w_n$, and $LV_0(\mathtt{x})$ is undefined for all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}} \setminus \{\mathtt{u}_1, \ldots, \mathtt{u}_n\}$
- $KB' = KB \cup \{w_1 = LV_1(t_1), \ldots, w_n = LV_1(t_n)\}$,

ret (p_0 : "`ret ty` t"; p_1 : "`x = call ty ...`" **with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}, t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)

$$\frac{((p_0,\, LV_0,\, AL_0) \cdot (p_1,\, LV_1,\, AL_1) \cdot CS,\, KB,\, AL,\, PT)}{((p_1^+,\, LV_1[\mathtt{x} := w],\, AL_1) \cdot CS,\, KB \cup \{w = LV_0(t)\},\, AL,\, PT')} \quad \text{if}$$

- $w \in \mathcal{V}_{sym}$ is fresh,
- PT' results from PT by removing all $v \hookrightarrow_{\mathtt{ty}} w$ where there exists some $[\![v_1,\, v_2]\!] \in AL_0$ with $\models \langle a \rangle \Rightarrow v_1 \leq v \wedge v \leq v_2$

3.3 Generalizing Abstract States

In the `strlen` example and its graph in Fig. 1, after reaching K, one unfolds the loop once more until one reaches a state $\widetilde{K}$ at position $(\mathtt{loop}, 4)$ again, analogous to the first iteration. To obtain *finite* symbolic execution graphs, we *generalize* our states whenever an evaluation visits a program position $(\mathtt{b}, j)$ twice and the domains of the local variable mappings LV_i in the two states are the same. Thus, we have to find a state that is more general than $K = ([(p, LV_1^K, \{\})], KB^K, AL, PT^K)$ and $\widetilde{K} = ([(p, LV_1^{\widetilde{K}}, \{\})], KB^{\widetilde{K}}, AL, PT^{\widetilde{K}})$. For readability, we again write "$\hookrightarrow$" instead of "$\hookrightarrow_{\mathtt{i8}}$". Then $p = (\mathtt{loop}, 4)$, $AL = \{[\![u_{\mathtt{str}},\, v_{end}]\!]\}$, and

$$\begin{aligned}
LV_1^K &= \{\mathtt{str}_1 = u_{\mathtt{str}}, \mathtt{c}_1 = v_5, \mathtt{s}_1 = v_4, \mathtt{olds}_1 = v_3, \ldots\} \\
LV_1^{\widetilde{K}} &= \{\mathtt{str}_1 = u_{\mathtt{str}}, \mathtt{c}_1 = \widetilde{v_5}, \mathtt{s}_1 = \widetilde{v_4}, \mathtt{olds}_1 = \widetilde{v_3}, \ldots\} \\
PT^K &= \{u_{\mathtt{str}} \hookrightarrow v_1, v_4 \hookrightarrow v_5, v_{end} \hookrightarrow z\} \\
PT^{\widetilde{K}} &= \{u_{\mathtt{str}} \hookrightarrow v_1, v_4 \hookrightarrow v_5, \widetilde{v_4} \hookrightarrow \widetilde{v_5}, v_{end} \hookrightarrow z\} \\
KB^K &= \{v_5 \neq 0, v_4 = v_3 + 1, v_3 = u_{\mathtt{str}}, v_1 \neq 0, z = 0, \ldots\} \\
KB^{\widetilde{K}} &= \{\widetilde{v_5} \neq 0, \widetilde{v_4} = \widetilde{v_3} + 1, \widetilde{v_3} = v_4, v_4 = v_3 + 1, v_3 = u_{\mathtt{str}}, v_1 \neq 0, z = 0, \ldots\}.
\end{aligned}$$

Our aim is to construct a new state L that is more general than K and $\widetilde{K}$, but contains enough information for the remaining proof. We now present our heuristic for *merging* states that is used in our implementation.

To merge K and $\widetilde{K}$, we keep those constraints of K that also hold in $\widetilde{K}$. To this end, we proceed in two steps. First, we create a new state $L = ([(p, LV_1^L, \{\})], KB^L, AL^L, PT^L)$ using fresh symbolic variables $v_{\mathtt{x}}$ for all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ where LV_1^K and $LV_1^{\widetilde{K}}$ are defined. This yields

$$LV_1^L = \{\mathtt{str}_1 = v_{\mathtt{str}}, \mathtt{c}_1 = v_{\mathtt{c}}, \mathtt{s}_1 = v_{\mathtt{s}}, \mathtt{olds}_1 = v_{\mathtt{olds}}, \ldots\}.$$

We then create mappings μ_K (resp. $\mu_{\widetilde{K}}$) from L's symbolic variables to their counterparts in K (resp. $\widetilde{K}$), i.e., $\mu_K(v_{\mathtt{x}}) = LV_1^K(\mathtt{x})$ if $LV_1^K(\mathtt{x})$ is defined. In our example, $\mu_K(v_{\mathtt{str}}) = u_{\mathtt{str}}$, $\mu_K(v_{\mathtt{c}}) = v_5$, $\mu_K(v_{\mathtt{s}}) = v_4$, $\mu_K(v_{\mathtt{olds}}) = v_3$, and $\mu_{\widetilde{K}}(v_{\mathtt{str}}) = u_{\mathtt{str}}$, $\mu_{\widetilde{K}}(v_{\mathtt{c}}) = \widetilde{v_5}$, $\mu_{\widetilde{K}}(v_{\mathtt{s}}) = \widetilde{v_4}$, $\mu_{\widetilde{K}}(v_{\mathtt{olds}}) = \widetilde{v_3}$. By injectivity of LV_1^K, we can also define a pseudo-inverse of μ_K that maps K's variables to L by setting $\mu_K^{-1}(LV_1^K(\mathtt{x})) = v_{\mathtt{x}}$ if $LV_1^K(\mathtt{x})$ is defined and $\mu_K^{-1}(v) = v$ for all other $v \in \mathcal{V}_{sym}$ ($\mu_{\widetilde{K}}^{-1}$ is analogous). So symbolic variables in K and $\widetilde{K}$ corresponding to the same program variable are mapped to the same symbolic variable by μ_K^{-1} and $\mu_{\widetilde{K}}^{-1}$.

In a second step, we use the mappings μ_K^{-1} and $\mu_{\widetilde{K}}^{-1}$ to check which constraints of K also hold in $\widetilde{K}$. So we set $AL^L = \mu_K^{-1}(AL) \cap \mu_{\widetilde{K}}^{-1}(AL) = \{[\![v_{\mathtt{str}},\, v_{end}]\!]\}$ and

$$\begin{aligned} PT^L &= \mu_K^{-1}(PT^K) \cap \mu_{\widetilde{K}}^{-1}(PT^{\widetilde{K}}) \\ &= \{v_{\mathtt{str}} \hookrightarrow v_1, v_{\mathtt{s}} \hookrightarrow v_{\mathtt{c}}, v_{end} \hookrightarrow z\} \cap \{v_{\mathtt{str}} \hookrightarrow v_1, v_4 \hookrightarrow v_5, v_{\mathtt{s}} \hookrightarrow v_{\mathtt{c}}, v_{end} \hookrightarrow z\} \\ &= \{v_{\mathtt{str}} \hookrightarrow v_1, v_{\mathtt{s}} \hookrightarrow v_{\mathtt{c}}, v_{end} \hookrightarrow z\}. \end{aligned}$$

Here, v_1 is not changed by μ_K^{-1} and $\mu_{\widetilde{K}}^{-1}$ because it is not assigned to a program variable.

It remains to construct KB^L. Essentially, we would like to take the intersection of those formulas that are implied by the knowledge bases of K and $\widetilde{K}$. So in principle, we would like to define the knowledge base of the new merged state L as follows:

$$KB^L = \{\ \varphi \mid\ \ \models \mu_K^{-1}(\langle K \rangle) \Rightarrow \varphi \text{ and } \models \mu_{\widetilde{K}}^{-1}(\langle \widetilde{K} \rangle) \Rightarrow \varphi\ \} \tag{1}$$

However, with this definition KB^L would be an infinite set of formulas, which is not suitable for automation. Thus, we restrict the definition of KB^L to a finite subset of (1) that can be automatically generated from the knowledge bases of the states K and $\widetilde{K}$. This restriction is only a heuristic and has no impact on the correctness, since it would already be correct to include all formulas of (1) in the knowledge base of the generalized state.

Our heuristic for the restriction of (1) considers K's knowledge base and extends it by certain additional formulas. This leads to a finite set $\langle\!\langle K \rangle\!\rangle$. For these finitely many formulas we then check whether they are also implied by $\langle \widetilde{K} \rangle$ (when renaming variables appropriately).

More precisely, we have $v_3 = u_{\mathtt{str}}$ ("$\mathtt{olds} = \mathtt{str}$") in $\langle K \rangle$, but $\widetilde{v}_3 = v_4$, $v_4 = v_3 + 1$, $v_3 = u_{\mathtt{str}}$ ("$\mathtt{olds} = \mathtt{str} + 1$") in $\langle \widetilde{K} \rangle$. To keep as much information as possible, we rewrite equations to inequations before performing the generalization. So let $\langle\!\langle K \rangle\!\rangle$ result from extending $\langle K \rangle$ by $t_1 \geq t_2$ and $t_1 \leq t_2$ for any equation $t_1 = t_2 \in \langle K \rangle$. In our example, we obtain $v_3 \geq u_{\mathtt{str}} \in \langle\!\langle K \rangle\!\rangle$ ("$\mathtt{olds} \geq \mathtt{str}$"). Moreover, for any $t_1 \neq t_2 \in \langle K \rangle$, we check whether $\langle K \rangle$ implies $t_1 > t_2$ or $t_1 < t_2$, and add the respective inequation to $\langle\!\langle K \rangle\!\rangle$. In this way, one can express sequences of inequations $t_1 \neq t_2$, $t_1 + 1 \neq t_2$, $\ldots$, $t_1 + n \neq t_2$ (where $t_1 \leq t_2$) by a single inequation $t_1 + n < t_2$, which is needed for suitable generalizations afterwards. We use this to derive $v_4 < v_{end} \in \langle\!\langle K \rangle\!\rangle$ ("$\mathtt{s} < v_{end}$") from $v_4 = v_3 + 1$, $v_3 = u_{\mathtt{str}}$, $u_{\mathtt{str}} \leq v_{end}$, $u_{\mathtt{str}} \neq v_{end}$, $v_4 \neq v_{end} \in \langle K \rangle$.

We then let KB^L consist of all formulas φ from $\langle\!\langle K \rangle\!\rangle$ that are also implied by $\langle \widetilde{K} \rangle$, again translating variable names using μ_K^{-1} and $\mu_{\widetilde{K}}^{-1}$. Thus, we have

$$\begin{aligned} \langle\!\langle K \rangle\!\rangle &= \{v_5 \neq 0, v_4 = v_3 + 1, v_3 = u_{\mathtt{str}}, v_3 \geq u_{\mathtt{str}}, v_4 < v_{end}, \ldots\} \\ \mu_K^{-1}(\langle\!\langle K \rangle\!\rangle) &= \{v_{\mathtt{c}} \neq 0, v_{\mathtt{s}} = v_{\mathtt{olds}} + 1, v_{\mathtt{olds}} = v_{\mathtt{str}}, v_{\mathtt{olds}} \geq v_{\mathtt{str}}, v_{\mathtt{s}} < v_{end}, \ldots\} \\ \mu_{\widetilde{K}}^{-1}(\langle \widetilde{K} \rangle) &= \{v_{\mathtt{c}} \neq 0, v_{\mathtt{s}} = v_{\mathtt{olds}} + 1, v_{\mathtt{olds}} = v_4, v_4 = v_3 + 1, v_3 = v_{\mathtt{str}}, v_{\mathtt{s}} < v_{end}, \ldots\} \\ KB^L &= \{v_{\mathtt{c}} \neq 0, v_{\mathtt{s}} = v_{\mathtt{olds}} + 1, v_{\mathtt{olds}} \geq v_{\mathtt{str}}, v_{\mathtt{s}} < v_{end}, \ldots\}. \end{aligned}$$

In Fig. 1, we do not show the second loop unfolding from K to $\widetilde{K}$, and directly draw a *generalization edge* with a dashed arrow from K to L. Such an edge expresses that all concrete states represented by K are also represented by the more general state L. Semantically, a state a' is a generalization of a state a iff $\models \langle a \rangle_{SL} \Rightarrow \mu(\langle a' \rangle_{SL})$ for some instantiation μ.

In the `strlen` example, we continue symbolic execution in state L. Similar to the execution from F to K, after 5 steps another state N at position $(\mathtt{loop}, 4)$ is reached. In Fig. 1, the dotted arrows from L to M and from M to N abbreviate several evaluation steps. As L is again a generalization of N using an instantiation μ with $\mu(v_{\mathtt{c}}) = w_{\mathtt{c}}$, $\mu(v_{\mathtt{s}}) = w_{\mathtt{s}}$, and $\mu(v_{\mathtt{olds}}) = w_{\mathtt{olds}}$, we draw a generalization edge from N to L. The construction of a symbolic execution graph is finished as soon as all leaves have only one stack frame, which is at a

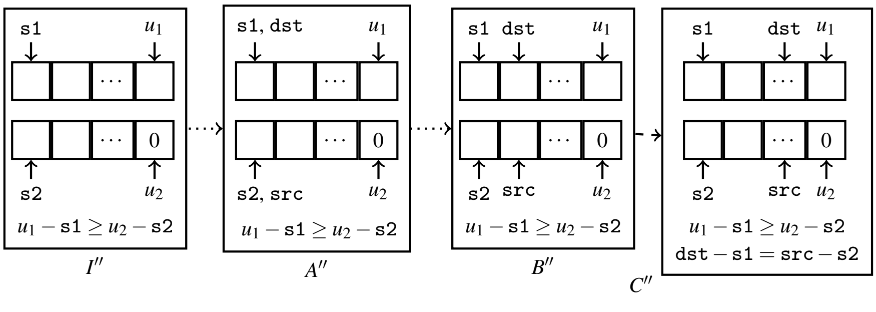
s1
u_1
0
s2
u_2
$u_1 - \mathtt{s1} \geq u_2 - \mathtt{s2}$
I''
s1, dst
u_1
0
s2, src
u_2
$u_1 - \mathtt{s1} \geq u_2 - \mathtt{s2}$
A''
s1 dst
u_1
0
s2 src
u_2
$u_1 - \mathtt{s1} \geq u_2 - \mathtt{s2}$
B''
s1 dst u_1
0
s2 src u_2
$u_1 - \mathtt{s1} \geq u_2 - \mathtt{s2}$
$\mathtt{dst} - \mathtt{s1} = \mathtt{src} - \mathtt{s2}$
C''

Fig. 3 The `strcpy` function and a graphical illustration of its symbolic execution

`ret` instruction. We call a non-empty symbolic execution graph with this property *complete*. In particular, a complete symbolic execution graph cannot contain an *ERR* state.

The approach presented so far is sufficient to prove memory safety (and together with the techniques in Sect. 5 also termination) of the `strlen` function, cf. Sect. 4 and 5. Up to now, when merging states we make relations between symbolic variables explicit (by adding inequations between symbolic variables). Then, these inequations are retained in the merged state if they are present in both states to be merged. In other words, these inequations restrict the state space of the represented concrete states and we want to keep as many restrictions as possible during merging in order to obtain a more precise abstraction. In some cases, however, it is also important to make relations between *differences of* symbolic variables explicit (e.g., about the distance between addresses). So in addition to inequations like $v_1 \geq v_2$ or $v_1 > v_2$ in $\langle\!\langle K \rangle\!\rangle$, we may also add equations like $v_1 - v_2 = w_1 - w_2$ for symbolic variables v_1, v_2, w_1, w_2. By making these equations explicit, they can also be retained when merging states.

So far, relations established and preserved by instructions within a "loop" (i.e., a path through the program leading from some program position back to the same position) are usually retained by our merging heuristic. For example, the instruction `s = getelementptr i8* olds, i32 1` within the block `loop` leads to the relation $v_4 = v_3 + 1$ in K and to the relation $\widetilde{v_4} = \widetilde{v_3} + 1$ in $\widetilde{K}$, where v_4 and $\widetilde{v_4}$ correspond to the program variable `s` and v_3 and $\widetilde{v_3}$ corresponds to the program variable `olds`. Thus, the relation $v_{\mathtt{s}} = v_{\mathtt{olds}} + 1$ is also contained in the merged state L for the corresponding "merged" symbolic variables $v_{\mathtt{s}}$ and $v_{\mathtt{olds}}$.

However, relations established before a loop may be generalized or removed during merging. As example, the instruction `olds = phi i8* [str,entry], [s,loop]` assigns the value of `str` to the variable `olds` when the block `loop` is entered for the first time. So in the state K, we had the relation $v_3 = u_{\mathtt{str}}$ where the symbolic variables v_3 and $u_{\mathtt{str}}$ correspond to the program variables `olds` and `str`. Since in $\widetilde{K}$, the value of `olds` has been increased by 1, this is generalized to the inequation $v_{\mathtt{olds}} \geq v_{\mathtt{str}}$ in the merged state L. So by merging states, we lose the information on the exact distance between `olds` and its initial value `str`.

```
char* strcpy(char* s1, char* s2) {
  char* dst = s1;
  char* src = s2;
  while ((*dst++ = *src++) != ’\0’);
  return s1;                           }
```

Of course, we need to abstract to obtain a *finite* representation of all evaluations. However, we might want to keep the knowledge that two distances between different symbolic variables are the same. This knowledge is necessary for a successful analysis of the `strcpy` function on the right (cf. [62,72]). This function copies the string at the source address `s2` to the destination address `s1`. The `while` loop of the function terminates as soon as the value `0` is reached in the source string.

To ease readability, we do not depict the full symbolic execution graph. Instead, Fig. 3 shows a graphical illustration of some key program states in the execution of `strcpy`. The initial state I'' describes states in which the destination `s1` begins an allocated memory block whose length is at least as long as the source string `s2`. Moreover, the symbolic variables u_1 and u_2 refer to the last address in each allocated memory block. State A'' corresponds to the first entry into the loop, in which the program variables `dst` and `src` point to the same addresses as `s1` and `s2`, respectively. After one loop iteration, both `src` and `dst` have been incremented by one, as shown in B''. For the states A'' and B'', the merging approach presented so far would generate a state requiring only $\texttt{s1} \leq \texttt{dst} \leq u_1$ and $\texttt{s2} \leq \texttt{src} \leq u_2$, but it would not keep any information about the exact distances of `dst` from `s1` and of `src` from `s2`. However, this is not sufficient to prove memory safety (and hence termination) of the `strcpy` function, as this generalized state would also represent cases in which the destination memory area starting at `dst` is shorter than the source area. To handle such examples successfully, our merging heuristic needs to relate the difference between `dst` and `s1` with the difference between `src` and `s2`, obtaining a state such as C''.

So when merging two states a and b, we also check whether there are symbolic variables $v_1^a, v_2^a, v_3^a, v_4^a$ with $(v_1^a, v_2^a) \neq (v_3^a, v_4^a)$ in state a such that $v_1^a - v_2^a = k_1 \cdot (v_3^a - v_4^a)$ for some constant k_1. To simplify the search, we only consider cases where $v_3^a - v_4^a = k_2$ for some constant k_2, and to avoid several equivalent equations due to symmetries, we require that $k_1 > 0$ and $k_2 \geq 0$. If the corresponding relation $v_1^b - v_2^b = k_1 \cdot (v_3^b - v_4^b)$ also holds for the symbolic variables $v_1^b, v_2^b, v_3^b, v_4^b$ in state b that are "merged with" $v_1^a, v_2^a, v_3^a, v_4^a$, then the relation $v_1 - v_2 = k_1 \cdot (v_3 - v_4)$ is added to the knowledge base of the merged state for the "merged" symbolic variables v_i. So for `strcpy`, since $\texttt{dst} - \texttt{s1} = \texttt{src} - \texttt{s2}$ holds in both states A'' and B'', this equation is contained in the knowledge base of the state C'' that results from merging A'' and B''. When merging states in this way, termination of `strcpy` can be proved automatically, similar to `strlen`. Def. 7 formalizes our technique for merging states.

Definition 7 (Merging States) Let $a = ([(p_1, LV_1^a, AL_1^a), ..., (p_n, LV_n^a, AL_n^a)], KB^a, AL^a, PT^a)$, $b = ([(p_1, LV_1^b, AL_1^b), \ldots, (p_n, LV_n^b, AL_n^b)], KB^b, AL^b, PT^b)$ be abstract states. Moreover, for all $i \in \{1, \ldots, n\}$, let the domains of LV_i^a and LV_i^b coincide. Then $c = (CS^c, KB^c, AL^c, PT^c)$ with $CS^c = [(p_1, LV_1^c, AL_1^c), \ldots, (p_n, LV_n^c, AL_n^c)]$ results from *merging* the states a and b if

- $LV_i^c = \{\texttt{x}_i = v_{\texttt{x}}^i \mid \texttt{x} \in \mathcal{V}_{\mathcal{P}}$ where $LV_i^a(\texttt{x})$ is defined $\}$ for all $1 \leq i \leq n$ and fresh pairwise different symbolic variables $v_{\texttt{x}}^i$. Moreover, we define $\mu_a(v_{\texttt{x}}^i) = LV_i^a(\texttt{x})$ and $\mu_b(v_{\texttt{x}}^i) = LV_i^b(\texttt{x})$ for all $\texttt{x} \in \mathcal{V}_{\mathcal{P}}$ where $LV_i^a(\texttt{x})$ is defined, and we let μ_a and μ_b be the identity on all remaining variables from $\mathcal{V}_{sym}$.
- $PT^c = \mu_a^{-1}(PT^a) \cap \mu_b^{-1}(PT^b)$, $AL^c = \mu_a^{-1}(AL^a) \cap \mu_b^{-1}(AL^b)$, and $AL_i^c = \mu_a^{-1}(AL_i^a) \cap \mu_b^{-1}(AL_i^b)$ for all $1 \leq i \leq n$. Here, the "inverse" of μ_a is defined as $\mu_a^{-1}(v) = v_{\texttt{x}}^i$ if $v = LV_i^a(\texttt{x})$ and $\mu_a^{-1}(v) = v$ for all other $v \in \mathcal{V}_{sym}$ (μ_b^{-1} is defined analogously).
- $KB^c = \{\ \varphi \in \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle) \mid\ \models \mu_b^{-1}(\langle b \rangle) \Rightarrow \varphi\ \}$, where $\langle\!\langle a \rangle\!\rangle$ is the smallest set such that
  - $\langle a \rangle \subseteq \langle\!\langle a \rangle\!\rangle$
  - $t_1 = t_2 \in \langle\!\langle a \rangle\!\rangle \implies t_1 \geq t_2, t_1 \leq t_2 \in \langle\!\langle a \rangle\!\rangle$
  - $(t_1 \neq t_2 \in \langle\!\langle a \rangle\!\rangle \wedge \models \langle a \rangle \Rightarrow t_1 > t_2) \implies t_1 > t_2 \in \langle\!\langle a \rangle\!\rangle$
  - $(t_1 \neq t_2 \in \langle\!\langle a \rangle\!\rangle \wedge \models \langle a \rangle \Rightarrow t_1 < t_2) \implies t_1 < t_2 \in \langle\!\langle a \rangle\!\rangle$
  - $\models \langle a \rangle \Rightarrow v_1 - v_2 = k_1 \cdot k_2 \wedge v_3 - v_4 = k_2 \quad \implies \quad v_1 - v_2 = k_1 \cdot (v_3 - v_4) \in \langle\!\langle a \rangle\!\rangle$
    for all $k_1, k_2 \in \mathbb{N}$ with $k_1 > 0$ and all $v_1, v_2, v_3, v_4 \in \mathcal{V}_{sym}(a)$ with $(v_1, v_2) \neq (v_3, v_4)$.

Note that $\langle\!\langle a \rangle\!\rangle$ may contain arbitrary formulas from $QF\_IA(\mathcal{V}_{sym})$ since $\langle a \rangle \subseteq \langle\!\langle a \rangle\!\rangle$. Moreover, the terms t_1, t_2 in $\langle\!\langle a \rangle\!\rangle \setminus \langle a \rangle$ are arbitrary (possibly non-linear) arithmetic terms. So our definition for $\langle\!\langle a \rangle\!\rangle$ is beyond those classes of formulas that are typically used in abstract interpretation (i.e., beyond octagons [57] and even polyhedra [29]). This is crucial for the success of our approach, since the conditions of programs often contain formulas that are not in these restricted classes. Indeed, our definition of $\langle\!\langle a \rangle\!\rangle$ was very successful in our implementation.

We now define a rule to compute *generalization edges* automatically. Recall that semantically, a state a' is a generalization of a state a iff $\models \langle a \rangle_{SL} \Rightarrow \mu(\langle a' \rangle_{SL})$ for some instantiation μ. To automate our procedure, we define a weaker relationship between a and a'. We say that $a' = (CS', KB', AL', PT')$ is a *generalization* of $a = (CS, KB, AL, PT)$ with the instantiation μ whenever the conditions (b) – (f) of the following rule are satisfied. Again, let a denote the state *before* the generalization step (i.e., above the horizontal line of the rule) and let a' be the state *resulting* from the generalization (i.e., below the line).

generalization with μ

$$\frac{([(p_1,\, LV_1,\, AL_1), \ldots, (p_n,\, LV_n,\, AL_n)],\, KB,\, AL,\, PT)}{([(p_1,\, LV'_1,\, AL'_1), \ldots, (p_n,\, LV'_n,\, AL'_n)],\, KB',\, AL',\, PT')} \quad \text{if}$$

(a) a has an incoming evaluation edge,[9]
(b) LV_i and LV'_i have the same domain and $LV_i(\mathtt{x}) = \mu(LV'_i(\mathtt{x}))$ for all $1 \leq i \leq n$ and all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ where LV_i and LV'_i are defined,
(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) if $[\![v_1, v_2]\!] \in AL'_i$, then $[\![\mu(v_1), \mu(v_2)]\!] \in AL_i$ (for all $1 \leq i \leq n$),
(f) if $(v_1 \hookrightarrow_{\mathtt{ty}} v_2) \in PT'$, then $(\mu(v_1) \hookrightarrow_{\mathtt{ty}} \mu(v_2)) \in PT$

The above rule does not refer to the merging of states in Def. 7, but it introduces a general form of "generalizations". The correctness of this rule is obvious, as it clearly implies $\models \langle a \rangle_{SL} \Rightarrow \mu(\langle a' \rangle_{SL})$. Condition (a) is needed to avoid cycles of refinement and generalization steps in the symbolic execution graph, which would not correspond to any computation.

Of course, many approaches are possible to compute such generalizations. Thm. 8 shows that the merging heuristic from Def. 7 satisfies the conditions of the generalization rule. So if a state c results from merging a and b, then c is indeed a *generalization* of both a and b. Thm. 8 also shows that if one uses the merging heuristic for generalizations, then the construction of symbolic execution graphs always terminates when applying the following strategy:

(1) If b is the next state to evaluate symbolically and there is a path from some state a to b, where a and b are at the same program position, the domains of all functions LV in a are equal to the domains of the corresponding functions LV in b, b has an incoming evaluation edge, and a has no incoming refinement edge, then:

(1a) If a is a generalization of b (i.e., the corresponding conditions of the generalization rule are satisfied), then we draw a generalization edge from b to a. Here, any SMT solver can be used to prove Condition (c) of the "**generalization**" rule.

(1b) Otherwise, remove a's children, and add a generalization edge from a to the merging c of a and b. If a already had an incoming generalization edge from some state q, then remove a and add a generalization edge from q to c instead.

(2) Otherwise, just evaluate b symbolically as usual, applying refinements when needed.

Theorem 8 (Soundness and Termination of Merging) *Let c result from merging the states a and b as in Def. 7. Then c is a generalization of a and b with the instantiations μ_a and μ_b, respectively. Moreover, if a is not already a generalization of b, and n is the height of the call stacks in a, b, and c, then $|\langle\!\langle c \rangle\!\rangle| + (\sum_{1 \leq i \leq n} |AL_i^c|) + |AL^c| + |PT^c| < |\langle\!\langle a \rangle\!\rangle| + (\sum_{1 \leq i \leq n} |AL_i^a|) + |AL^a| + |PT^a|$. Here, for any conjunction φ, let $|\varphi|$ denote the number of its conjuncts. Thus, the above strategy to construct symbolic execution graphs always terminates.*

Proof To show that c is a generalization of a and b with the instantiations μ_a and μ_b, we have

[9] *Evaluation* edges are edges that are not refinement or generalization edges.

to prove that the conditions (b) – (f) of the generalization rule are satisfied. By definition, $LV_i^a(\mathtt{x}) = \mu_a(v_{\mathtt{x}}^i) = \mu_a(LV_i^c(\mathtt{x}))$ and $LV_i^b(\mathtt{x}) = \mu_b(LV_i^c(\mathtt{x}))$ for all $1 \le i \le n$ and all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, which proves (b). Moreover, for $[\![v_1, v_2]\!] \in AL^c$, we have $[\![v_1, v_2]\!] \in \mu_a^{-1}(AL^a)$ and $[\![v_1, v_2]\!] \in \mu_b^{-1}(AL^b)$. This implies $[\![\mu_a(v_1), \mu_a(v_2)]\!] \in AL^a$ and $[\![\mu_b(v_1), \mu_b(v_2)]\!] \in AL^b$, which proves (d). Condition (e) on AL_i^c and condition (f) on PT^c can be proved in a similar way.

It remains to prove (c). As $KB^c \subseteq \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle)$, we have $\models \langle\!\langle a \rangle\!\rangle \Rightarrow \mu_a(KB^c)$ and therefore also $\models \langle a \rangle \Rightarrow \mu_a(KB^c)$. Moreover, as $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow \varphi$ holds for all $\varphi \in KB^c$, we also obtain $\models \langle b \rangle \Rightarrow \mu_b(KB^c)$. Note that we even have $\models \langle a \rangle \Rightarrow \mu_a(\langle c \rangle)$ and $\models \langle b \rangle \Rightarrow \mu_b(\langle c \rangle)$.

Finally, we show that $|\langle\!\langle c \rangle\!\rangle| + (\sum_{1 \le i \le n} |AL_i^c|) + |AL^c| + |PT^c| < |\langle\!\langle a \rangle\!\rangle| + (\sum_{1 \le i \le n} |AL_i^a|) + |AL^a| + |PT^a|$ if a is not a generalization of b.

We first show that $\langle\!\langle c \rangle\!\rangle = \langle c \rangle$. The reason is that whenever there is a $t_1 = t_2 \in \langle c \rangle$, then $t_1 = t_2 \in \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle)$ and thus also $t_1 \ge t_2, t_1 \le t_2 \in \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle)$. As $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow t_1 = t_2$ also implies $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow t_1 \ge t_2$ and $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow t_1 \le t_2$, we also have $t_1 \ge t_2, t_1 \le t_2 \in \langle c \rangle$. Moreover, suppose that $t_1 \ne t_2 \in \langle c \rangle$ and $\models \langle c \rangle \Rightarrow t_1 > t_2$. This implies $\models \mu_a^{-1}(\langle a \rangle) \Rightarrow t_1 > t_2$ (i.e., $t_1 > t_2 \in \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle)$) and $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow t_1 > t_2$. Hence, we also obtain $t_1 > t_2 \in \langle c \rangle$. The case where $t_1 \ne t_2 \in \langle c \rangle$ and $\models \langle c \rangle \Rightarrow t_1 < t_2$ is analogous. Finally, consider the case that $\models \langle c \rangle \Rightarrow v_1 - v_2 = k_1 \cdot k_2 \land v_3 - v_4 = k_2$ holds for some $k_1, k_2 \in \mathbb{N}$ and $v_1, v_2, v_3, v_4 \in \mathcal{V}_{sym}(c)$ with $(v_1, v_2) \ne (v_3, v_4)$. Since $\models \langle a \rangle \Rightarrow \mu_a(\langle c \rangle)$, we also have $\mu_a(v_1 - v_2 = k_1 \cdot (v_3 - v_4)) \in \langle\!\langle a \rangle\!\rangle$, i.e., $(v_1 - v_2 = k_1 \cdot (v_3 - v_4)) \in \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle)$. Moreover, due to $\models \langle b \rangle \Rightarrow \mu_b(\langle c \rangle)$ we have $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow v_1 - v_2 = k_1 \cdot (v_3 - v_4)$. This implies $v_1 - v_2 = k_1 \cdot (v_3 - v_4) \in KB^c \subseteq \langle c \rangle$.

Next note that $\langle c \rangle = KB^c$. Again the reason is that for any $\varphi \in \langle c \rangle$ we have $\varphi \in \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle)$ and $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow \varphi$. Thus, we only have to show that $|KB^c| + (\sum_{1 \le i \le n} |AL_i^c|) + |AL^c| + |PT^c| < |\langle\!\langle a \rangle\!\rangle| + (\sum_{1 \le i \le n} |AL_i^a|) + |AL^a| + |PT^a|$. From the definition, it is obvious that we always have $|KB^c| \le |\langle\!\langle a \rangle\!\rangle|$, $|AL^c| \le |AL^a|$, $|AL_i^c| \le |AL_i^a|$ for all $1 \le i \le n$, and $|PT^c| \le |PT^a|$.

Hence, it suffices to show that if $|KB^c| = |\langle\!\langle a \rangle\!\rangle|$, $|AL^c| = |AL^a|$, $|AL_i^c| = |AL_i^a|$ for all $1 \le i \le n$, and $|PT^c| = |PT^a|$, then a would be a generalization of b with the instantiation $\mu_b \circ \mu_a^{-1}$. To see this, note that we have $LV_i^b(\mathtt{x}) = \mu_b(v_{\mathtt{x}}^i) = \mu_b(\mu_a^{-1}(LV_i^a(\mathtt{x})))$, i.e., condition (b) of the generalization rule is satisfied. Clearly, $|AL^c| = |AL^a|$ means that $\mu_a^{-1}(AL^a) = \mu_b^{-1}(AL^b)$. Thus, if $[\![v_1, v_2]\!] \in AL^a$, then $[\![\mu_a^{-1}(v_1), \mu_a^{-1}(v_2)]\!] \in \mu_a^{-1}(AL^a) = \mu_b^{-1}(AL^b)$ and hence, $[\![\mu_b(\mu_a^{-1}(v_1)), \mu_b(\mu_a^{-1}(v_2))]\!] \in AL^b$, which shows condition (d). Conditions (e) and (f) follow from $|AL_i^c| = |AL_i^a|$ resp. $|PT^c| = |PT^a|$ for similar reasons. Finally, $|KB^c| = |\langle\!\langle a \rangle\!\rangle|$ means that for all $\varphi \in \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle)$, we have $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow \varphi$. Let $\psi \in \mu_b(\mu_a^{-1}(KB^a))$. Then we have $\mu_b^{-1}(\psi) \in \mu_a^{-1}(KB^a) \subseteq \mu_a^{-1}(\langle\!\langle a \rangle\!\rangle)$. Hence, we can infer $\models \mu_b^{-1}(\langle b \rangle) \Rightarrow \mu_b^{-1}(\psi)$ which implies $\models \langle b \rangle \Rightarrow \psi$, cf. condition (c). □

4 Correctness of Symbolic Execution

We now prove the correctness of our approach in Sect. 2 and 3, i.e., that our symbolic execution graphs represent an over-approximation of all concrete program runs. We proceed in two stages, as depicted graphically in Fig. 4. This proof structure is inspired by the correctness proof of our termination technique for Java w.r.t. a suitable formal semantics [11]. First, we relate the *formal* definition of the LLVM semantics from the Vellvm project [73] to our semantics $\to_{\mathsf{LLVM}}$ of LLVM from Sect. 2 and 3 that we use for program analysis. Here, $\to_{\mathsf{LLVM}}$ is defined by applying our symbolic execution rules of Sect. 3 to concrete states. Only for rules that deal with memory access (via `load`, `store`, `alloca`, or `malloc`), our symbolic execution rules have to be adapted slightly. This is necessary since the concrete rules essentially have to implement an LLVM interpreter. For example, in a concrete state we know the size of an allocated memory block in AL^* (say, n bytes). Thus, the concrete rules put n entries for this block into PT to track the contents of all currently allocated memory. In

$$\begin{tikzcd}
a_1 \arrow[r, "\textsc{Eval}"] & a_2 \arrow[r, "\textsc{Generalize}"] & a_2' \arrow[r, "\textsc{Refine}"] & a_2'' \arrow[r, "\textsc{Eval}"] & a_3 & \cdots & \\
c_1 \arrow[u, "\text{repr}"] \arrow[r, "\mathsf{LLVM}"] & c_2 \arrow[u, "\text{repr}"] \arrow[ur, "\text{repr}"] \arrow[urr, "\text{repr}"] \arrow[rrr, "\mathsf{LLVM}"] & & & c_3 \arrow[u, "\text{repr}"] & \cdots & \left.\right\} \text{Thm. 10} \\
v_1 \arrow[u, "\textsc{trans}"] \arrow[r, "\mathsf{Vellvm}"] & v_2 \arrow[u, "\textsc{trans}"] \arrow[rrr, "\mathsf{Vellvm}"] & & & v_3 \arrow[u, "\textsc{trans}"] & \cdots & \left.\right\} \text{Thm. 9}
\end{tikzcd}$$

Fig. 4 Relation between evaluation in LLVM and paths in the symbolic execution graph

our abstract rules, the size of an allocated memory block may be unknown, and thus, we do not know how many $\hookrightarrow_{\texttt{ty}}$-entries to add to PT. Hence, we can only represent a *part* of the memory contents in PT. Similarly, our symbolic execution can abstract information when a `store` operation partially overwrites a multi-byte value. However, for the concrete semantics $\rightarrow_{\mathsf{LLVM}}$, we need to keep track of each allocated byte of memory. See [3] for the four cases where our rules for the abstract semantics need to be adapted for the concrete semantics.

Vellvm is a formalization of LLVM in the Coq [7] theorem prover. In this subsection, we only regard programs over the fragment supported by our rules. While Vellvm's non-deterministic semantics LLVM_{ND} returns `undef` (which we currently do not support) for a `load` from uninitialized allocated memory, its deterministic semantics LLVM_D returns the value 0. Thus, we use the semantics LLVM_D and denote its transition relation by $\rightarrow_{\mathsf{Vellvm}}$.

For our proof, we define a relation $\textsc{trans}$ between Vellvm states and concrete states in our representation. Thm. 9 will state that for every evaluation step $v_1 \rightarrow_{\mathsf{Vellvm}} v_2$ with $\textsc{trans}(v_1,c_1)$, there is a c_2 with $\textsc{trans}(v_2,c_2)$ such that $c_1 \rightarrow_{\mathsf{LLVM}} c_2$ holds. Moreover, if Vellvm's execution gets stuck in a state v (i.e., if the next instruction to execute would violate memory safety, denoted $\mathit{Stuck}(v)$) and $\textsc{trans}(v,c)$, then we have $c \rightarrow_{\mathsf{LLVM}} \mathit{ERR}$. So the idea is that we can "replay" any Vellvm execution as an execution on our concrete states. In a second step, we relate symbolic execution on abstract states to evaluation on concrete states. Thm. 10 states that if some concrete state c_1 is represented by a state a_1 in a symbolic execution graph (denoted by "repr" in Fig. 4) and $c_1 \rightarrow_{\mathsf{LLVM}} c_2$, then the graph contains a path from a_1 to a state a_2 in the symbolic execution graph such that a_2 represents c_2.

Together, Thm. 9 and Thm. 10 show that symbolic execution graphs simulate Vellvm execution, and hence, they imply the soundness of our technique for analyzing memory safety w.r.t. the Vellvm semantics of LLVM: Suppose that there is an LLVM-computation $v_1 \rightarrow_{\mathsf{Vellvm}} v_2 \rightarrow_{\mathsf{Vellvm}} \cdots \rightarrow_{\mathsf{Vellvm}} v_n$ with $\mathit{Stuck}(v_n)$ and v_1 is represented in the symbolic execution graph (i.e., there is a state a_1 in the graph with $\textsc{trans}(v_1,c_1)$ and c_1 is represented by a_1). Then by Thm. 9 there is a symbolic evaluation $c_1 \rightarrow_{\mathsf{LLVM}} c_2 \rightarrow_{\mathsf{LLVM}} \cdots \rightarrow_{\mathsf{LLVM}} c_n \rightarrow_{\mathsf{LLVM}} \mathit{ERR}$, where $\textsc{trans}(v_i,c_i)$ holds for all i. Hence, Thm. 10 implies that the symbolic execution graph also contains a path from the state a_1 to an ERR node.

Thm. 9 and 10 can also be used as a basis for the *certification* of termination proofs for LLVM. Several certifiers were developed to check the soundness of automatically generated termination proofs for *term rewrite systems* [8,25,69]. The correctness of these certifiers has been formally proved using Coq [7] or Isabelle/HOL [60]. To certify termination proofs for LLVM, one could build upon Vellvm by formalizing and proving the soundness of our symbolic execution (Thm. 9 and 10) within Coq or Isabelle.[10] Building on that, one could then formalize our approach to generate integer transition systems from the symbolic execution graph (Thm. 13 in Sect. 5) and one would also have to formalize the techniques used to prove termination of these ITSs. Steps in this direction are currently investigated within the certifier CeTA [69], which can already be directly coupled with AProVE [38, Sect. 3].

[10] This step corresponds to other work for machine-checked abstract interpreters [9,17,46].

Vellvm's representation of (concrete) program states is similar to our Def. 3. The main difference is that Vellvm does not use symbolic variables since its program states are not designed for symbolic execution. This was also our main reason for developing a new representation for program states. We now express Vellvm's representation in our terminology.

Vellvm *States.* A Vellvm state has the form $(M, \overrightarrow{\Sigma})$ for a memory state M and a list of stack frames $\overrightarrow{\Sigma}$ which is analogous to our call stack CS. In a stack frame $\Sigma = (\mathit{fid}, \mathtt{b}, \overrightarrow{c}, \mathit{tmn}, \Delta, \alpha)$, fid is the id of the current function, $\mathtt{b}$ is the label of the current basic block, $\overrightarrow{c}$ are the remaining instructions to be executed in the current block, with tmn as the terminator of the block (its last command). Together, these components correspond to our position $p = (\mathtt{b}, j)$ in the program where the command sequence "$\overrightarrow{c}, \mathit{tmn}$" begins in block $\mathtt{b}$ at line j. Recall that we assume block labels to be different across different functions. Thus, we do not need to represent fid explicitly in our states. The component Δ keeps track of the values of the local variables of the block and corresponds to our functions LV_i. The final component α (roughly) corresponds to our lists AL_i and keeps track of the memory blocks allocated by the current stack frame that are released automatically when the current function returns.

Vellvm does not use absolute memory addresses, but pairs of a memory-block identifier (a number which is increased in each allocation) and an offset in that block. We say that a block identifier is *valid* if the corresponding memory block has been allocated and not yet released. In a Vellvm memory state $M = (N, B, C)$, N denotes the number of the next fresh memory block to allocate, B is a partial map from valid block identifiers to the size of the blocks (like our entries $[\![v_1, v_2]\!] \in AL^*$ with size $v_2 - v_1 + 1$), and C is a partial map from pairs of a valid block identifier and an offset in that block to values (similar to our PT).

Vellvm represents values in three ways. For integers, $\mathsf{mb}(\mathit{sz}, \mathit{byte})$ represents the memory content byte and the bit-width sz of the overall integer (but not the position in the integer that this byte corresponds to). We represent similar information in PT. For uninitialized memory cells, the pseudo-value $\mathsf{muninit}$ is used, which stands for the value $\mathsf{0}$ in the semantics LLVM_D. For pointers, Vellvm uses $\mathsf{mptr}(\mathit{blk}, \mathit{ofs}, \mathit{idx})$, where the block blk and offset ofs characterize the pointer's target, and the index idx indicates which of the bytes of the pointer is represented.

Translation TRANS. We now define a translation *relation* TRANS between Vellvm states and concrete states. The reason why TRANS is a relation instead of a function is that in contrast to us, Vellvm represents blocks of memory by their size and an identifier number but without absolute addresses. So for a Vellvm state v, we want to describe all concrete states (cf. Def. 3) $c = (CS, KB, AL, PT)$ where $\mathrm{TRANS}(v, c)$ holds.

Consider a Vellvm memory state $M = (N, B, C)$. To assign start and end addresses for its memory blocks, we relate M to *any* memory allocation AL^* of blocks of the same sizes. So we require $AL^* = \{[\![v_{blk}, w_{blk}]\!] \mid B(blk) \text{ is defined}\}$ with $\models KB \Rightarrow w_{blk} - v_{blk} = B(blk) - 1$ where v_{blk} and w_{blk} are pairwise different symbolic variables for all blk where $B(blk)$ is defined.

To handle actual memory contents, we consider the values of $C(blk, ofs)$ and introduce fresh symbolic variables such that $PT = \{x_{(blk,ofs)} \hookrightarrow_{\mathtt{i8}} y_{(blk,ofs)} \mid C(blk, ofs) \text{ is defined}\}$. The value for the address $x_{(blk,ofs)}$ is obtained by adding ofs to the corresponding symbolic variable v_{blk} for the start of the block blk. So we require $\models KB \Rightarrow x_{(blk,ofs)} = v_{blk} + ofs$ whenever $C(blk, ofs)$ is defined. Moreover, KB must contain knowledge about the values stored in memory. If $C(blk, ofs) = \mathsf{muninit}$, then $y_{(blk,ofs)} = 0$ according to the deterministic semantics LLVM_D. If $C(blk, ofs) = \mathsf{mb}(sz, byte)$, we require $\models KB \Rightarrow y_{(blk,ofs)} = byte$. Here, we assume that $byte$ is already represented as a signed integer from $[-2^7, 2^7 - 1]$. Similarly, if $C(blk, ofs) = \mathsf{mptr}(blk', ofs', idx)$, then KB must contain the knowledge that $y_{(blk,ofs)}$ is the idx's byte of the value forming the address $v_{blk'} + ofs'$ (this byte is obtained as in Def. 5).

Finally, we relate Vellvm's call stack $\overrightarrow{\Sigma} = [fr_1, \ldots, fr_n]$ with $fr_i = (fid_i, \mathtt{b}_i, \overrightarrow{c_i}, tmn_i, \Delta_i, \alpha_i)$ to a call stack $CS = [(p_1, LV_1, AL_1), \ldots, (p_n, LV_n, AL_n)]$ for our concrete state. For any $1 \leq i \leq n$, let $p_i = (\mathtt{b}_i, j_i)$, where j_i is the position in the block $\mathtt{b}_i$ where the command sequence "$\overrightarrow{c_i}, tmn_i$" begins. Moreover, $LV_i(\mathtt{x})$ is defined iff $\Delta_i(\mathtt{x})$ is defined. In this case, $LV_i(\mathtt{x})$ is a fresh symbolic variable with $\models KB \Rightarrow LV_i(\mathtt{x}) = \Delta_i(\mathtt{x})$. To determine $AL_1, \ldots, AL_n$, and AL, we define $AL_i = \{[\![v_{blk}, w_{blk}]\!] \mid blk \in \alpha_i\}$ for $1 \leq i \leq n$ and $AL = AL^* \setminus \bigcup_{1 \leq i \leq n} AL_i$.

Evaluation Rules. We now show that our evaluation $\rightarrow_{\mathsf{LLVM}}$ simulates $\rightarrow_{\mathsf{Vellvm}}$. For reasons of space, we only demonstrate this for one Vellvm evaluation rule from [73], adapted to our notation. In the following rule for `br`, $\mathsf{eval}(\Delta, t)$ evaluates t according to Δ. Vellvm uses an operation findblock to obtain the block $\mathtt{b}_1$ with the instructions $\overrightarrow{phi_1}\,\overrightarrow{cmd_1}\,tmn_1$. Here, $\overrightarrow{phi_1}$ are the `phi` instructions of the block $\mathtt{b}_1$. This operation is implicit in our rules. Similar to our `br` rules, $\mathsf{computephi}(\Delta, \mathtt{b}, \mathtt{b}_1, \overrightarrow{phi_1})$ yields a new mapping Δ' for the local variables according to the `phi` instructions $\overrightarrow{phi_1}$ in the target block $\mathtt{b}_1$.

`br_true` (tmn : **"**`br i1` t, `label` $\mathtt{b}_1$, `label` $\mathtt{b}_2$**" with** $t \in \mathcal{V}_{\mathcal{P}} \cup \{0,1\}$ **and** $\mathtt{b}_1, \mathtt{b}_2 \in Blks$)

$$\frac{M,\ (fid, \mathtt{b}, [\,], tmn, \Delta, \alpha) \cdot \overrightarrow{\Sigma}}{M,\ (fid, \mathtt{b}_1, \overrightarrow{cmd_1}, tmn_1, \mathsf{computephi}(\Delta, \mathtt{b}, \mathtt{b}_1, \overrightarrow{phi_1}), \alpha) \cdot \overrightarrow{\Sigma}} \quad \text{if}$$

- $\mathsf{eval}(\Delta, t) = 1$,
- findblock yields $\mathtt{b}_1$ with the instructions $\overrightarrow{phi_1}\,\overrightarrow{cmd_1}\,tmn_1$

Thm. 9 shows that our evaluation rules on concrete states correspond to evaluation w.r.t. Vellvm. As mentioned, here we only consider the LLVM fragment handled by our rules and in addition, we assume that a `load` operation for a type `i`n with $n \bmod 8 \neq 0$ is only performed for values that were written by a `store` of type `i`n. Similarly, we assume that values written by a `store` operation for a type `i`n with $n \bmod 8 \neq 0$ will only be read by a `load` of the same type. The reason is that for simplicity, our concrete states do not keep track of the type with which a `store` operation was performed. Therefore, we cannot distinguish whether a later `load` of, e.g., an `i20` value should yield the contents of the memory cell or an unknown value. Our abstract domain over-estimates such incompatible reads by an unknown value.

Theorem 9 (Simulating Vellvm by Evaluation of Concrete States) *Let $\mathcal{P}$ be an* LLVM *program. For all* Vellvm *states, $v \rightarrow_{\mathsf{Vellvm}} \overline{v}$ implies that for any concrete state c with* $\textsc{trans}(v,c)$ *there exists a concrete state $\overline{c}$ with* $\textsc{trans}(\overline{v},\overline{c})$ *such that $c \rightarrow_{\mathsf{LLVM}} \overline{c}$. Moreover, if $Stuck(v)$ holds, then* $\textsc{trans}(v,c)$ *implies $c \rightarrow_{\mathsf{LLVM}} ERR$.*

Proof We show the simulation of Vellvm's rule `br_true` by our corresponding rule. The other cases are analogous. Let $v = (M, (fid, \mathtt{b}, [\,], tmn, \Delta, \alpha) \cdot \overrightarrow{\Sigma})$ and $\overline{v} = (M, (fid, \mathtt{b}_1, \overrightarrow{cmd_1}, tmn_1, \Delta', \alpha) \cdot \overrightarrow{\Sigma})$ such that $v \rightarrow_{\mathsf{Vellvm}} \overline{v}$ holds by the rule `br_true`. Assume that we have $\textsc{trans}(v,c)$ for $c = (((\mathtt{b}, j), LV_1, AL_1) \cdot CS, KB, AL, PT)$. As `br_true` is applicable to v, we know $\mathsf{eval}(\Delta, t) = 1$ and hence $t = 1$ or $t \in \mathcal{V}_{\mathcal{P}}$ with $\Delta(t) = 1$, implying $\models \langle c \rangle \Rightarrow LV_1(t) = 1$. Thus, we can apply our rule for "`br i1` t, `label` $\mathtt{b}_1$, `label` $\mathtt{b}_2$" to c and obtain $c \rightarrow_{\mathsf{LLVM}} \overline{c}$ for a state $\overline{c} = (((\mathtt{b}_1, j_1), LV'_1, AL_1) \cdot CS,\ KB \cup KB_{\mathtt{phi}},\ AL,\ PT)$ with $(LV'_1, KB_{\mathtt{phi}}) = computePhi(LV_1, \mathtt{b}, \mathtt{b}_1)$ and $j_1 = firstNonPhi(\mathtt{b}_1)$.

It remains to prove that $\textsc{trans}(\overline{v},\overline{c})$ holds. Note that $(\mathtt{b}_1, j_1)$ with $j_1 = firstNonPhi(\mathtt{b}_1)$ corresponds exactly to the position where "$\overrightarrow{cmd_1}, tmn_1$" begins in $\mathtt{b}_1$. Moreover, the components of AL^*, PT, and M do not change in the steps $c \rightarrow_{\mathsf{LLVM}} \overline{c}$ and $v \rightarrow_{\mathsf{Vellvm}} \overline{v}$. The computations for the `phi` instructions are analogous in both settings, i.e., from $\models \langle c \rangle \Rightarrow LV_1(\mathtt{x}) = \Delta(\mathtt{x})$ we get $\models \langle \overline{c} \rangle \Rightarrow LV'_1(\mathtt{x}) = \Delta'(\mathtt{x})$ for all $\mathtt{x}$, where $\Delta' = \mathsf{computephi}(\Delta, \mathtt{b}, \mathtt{b}_1, \overrightarrow{phi_1})$. □

We now show that evaluation of concrete states with $\to_{\mathsf{LLVM}}$ can be simulated by symbolic execution of abstract states. Together with Thm. 9, this proves that our symbolic execution correctly simulates LLVM according to the semantics of Vellvm, cf. Fig. 4.

Theorem 10 (Simulating Evaluation of Concrete States by Abstract States) *Let $\mathcal{P}$ be an* LLVM *program with a complete symbolic execution graph $\mathcal{G}$. Let c be a concrete state that is represented by some abstract state a in $\mathcal{G}$. Then $c \to_{\mathsf{LLVM}} \overline{c}$ implies that there is a path from a to an abstract state $\overline{a}$ in $\mathcal{G}$ such that $\overline{c}$ is represented by $\overline{a}$.*

Proof Let $c \to_{\mathsf{LLVM}} \overline{c}$, where c is represented by a state a in the graph $\mathcal{G}$, i.e., $(as^c, mem^c) \models \sigma(\langle a \rangle_{SL})$ for some concrete instantiation σ. Then $\overline{c}$ is also represented by a state in $\mathcal{G}$:

(a) If a has an outgoing evaluation edge to $\overline{a}$, then $(as^{\overline{c}}, mem^{\overline{c}}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$ for a concrete instantiation $\overline{\sigma}$ with $\overline{\sigma}(v) = \sigma(v)$ for all $v \in \mathcal{V}_{sym}(a)$. This is trivial for all rules except those for `load`, `store`, `alloca`, and `malloc`, since the same rules are applied to the concrete and abstract states (note that the evaluation rules are non-overlapping). The proof for the slightly adapted concrete rules for the four instructions above can be found at [3].

(b) If a's outgoing edges are refinement edges, then one of its successors $\widetilde{a}$ has an evaluation edge to another abstract state $\overline{a}$, where $(as^{\overline{c}}, mem^{\overline{c}}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$ for a concrete instantiation $\overline{\sigma}$ with $\overline{\sigma}(v) = \sigma(v)$ for all $v \in \mathcal{V}_{sym}(a)$.

(c) If a's outgoing edge is a generalization edge to a state $\widetilde{a}$ with some instantiation μ, and $\widetilde{a}$ has an evaluation edge to another abstract state $\overline{a}$, then $(as^{\overline{c}}, mem^{\overline{c}}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$ for a concrete instantiation $\overline{\sigma}$ with $\overline{\sigma}(v) = \sigma(\mu(v))$ for all $v \in \mathcal{V}_{sym}(\widetilde{a})$.

(d) Otherwise, there is a generalization edge from a to a state $\widetilde{a}$ with some instantiation μ, a refinement edge from $\widetilde{a}$ to some $\widehat{a}$, and an evaluation edge from $\widehat{a}$ to some $\overline{a}$, where $(as^{\overline{c}}, mem^{\overline{c}}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$ for a concrete $\overline{\sigma}$ with $\overline{\sigma}(v) = \sigma(\mu(v))$ for all $v \in \mathcal{V}_{sym}(\widetilde{a})$. □

Recall that a complete symbolic execution graph must not contain the state *ERR*. Thus, all concrete states represented by the abstract states in the graph are memory safe.

Corollary 11 (Memory Safety of LLVM Programs) *Let $\mathcal{P}$ be a program with a complete symbolic execution graph $\mathcal{G}$. Then $\mathcal{P}$ is memory safe for all states represented by $\mathcal{G}$.*

Proof If a concrete state c is represented by an abstract state a in the graph $\mathcal{G}$ where $\mathrm{TRANS}(v,c)$ and $Stuck(v)$ for some Vellvm state v, then by Thm. 9 we have $c \to_{\mathsf{LLVM}} ERR$. By Thm. 10, $c \to_{\mathsf{LLVM}} ERR$ implies that there is an edge from a to *ERR* in $\mathcal{G}$. However, this contradicts the prerequisite that $\mathcal{G}$ is complete and therefore does not contain *ERR*. □

5 From Symbolic Execution Graphs to Integer Transition Systems

Finally, we extract an *integer transition system* from the symbolic execution graph and use existing tools to prove its termination. The extraction step essentially restricts the information in abstract states to the integer constraints on symbolic variables. This conversion of memory-based arguments into integer arguments often suffices for the termination proof. The reason for considering only $\mathcal{V}_{sym}$ instead of $\mathcal{V}_{\mathcal{P}}$ is that since the mappings LV_i are injective, the variables $\mathcal{V}_{\mathcal{P}}$ are completely represented by symbolic variables and the conditions in the abstract states (which are crucial for termination) only concern symbolic variables.

For example, termination of `strlen` is proved by showing that `s` is increased as long as it is smaller than v_{end}, the end of the input string. In Fig. 1, this is explicit as the invariant $v_s < v_{end}$ holds in all states represented by L. Each `loop` iteration increases the value of v_s.

Formally, *ITSs* are graphs whose nodes are abstract states and whose edges are *transitions*. Let $\mathcal{V} \subseteq \mathcal{V}_{sym}$ be the finite set of all symbolic variables in states of the symbolic execution graph. A *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 pure quantifier-free formulas over $\mathcal{V} \uplus \mathcal{V}'$. Here, $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ represents the variable values *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, 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.[11]

We convert symbolic execution graphs to ITSs by transforming every edge into a transition. If there is a generalization edge from a to $\overline{a}$ with an instantiation μ, then the new value of any $v \in \mathcal{V}_{sym}(\overline{a})$ in $\overline{a}$ is $\mu(v)$. Hence, we create the transition $(a,\ \langle a \rangle \cup \{v' = \mu(v) \mid v \in \mathcal{V}_{sym}(\overline{a})\},\ \overline{a})$.[12] So for the edge from N to L in Fig. 1, we obtain the condition $\{w_{\mathtt{s}} = w_{\mathtt{olds}} + 1, w_{\mathtt{olds}} = v_{\mathtt{s}}, v_{\mathtt{s}} < v_{end}, v'_{\mathtt{str}} = v_{\mathtt{str}}, v'_{end} = v_{end}, v'_{\mathtt{c}} = w_{\mathtt{c}}, v'_{\mathtt{s}} = w_{\mathtt{s}}, \ldots\}$. This can be simplified to $\{v_{\mathtt{s}} < v_{end}, v'_{end} = v_{end}, v'_{\mathtt{s}} = v_{\mathtt{s}} + 1, \ldots\}$.

An evaluation or refinement edge from a to $\overline{a}$ does not change the variables of $\mathcal{V}_{sym}(a)$. Thus, we construct the transition $(a,\ \langle a \rangle \cup \{v' = v \mid v \in \mathcal{V}_{sym}(a)\},\ \overline{a})$.

So in the ITS resulting from Fig. 1, the condition of the transition from A to B is $\{v'_{end} = v_{end}, u'_{\mathtt{str}} = u_{\mathtt{str}}\}$. The condition for the transition from B to D is the same, but extended by $v'_1 = v_1$. Hence, in the transition from A to B, the value of v_1 can change arbitrarily (since $v_1 \notin \mathcal{V}_{sym}(A)$), but in the transition from B to D, it must remain the same.

Definition 12 (ITS from Symbolic Execution Graph) For a symbolic execution graph $\mathcal{G}$, the *corresponding integer transition system* $\mathcal{I}_{\mathcal{G}}$ has one transition for each edge in $\mathcal{G}$:

- 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 $\langle a \rangle \cup \{v' = \mu(v) \mid v \in \mathcal{V}_{sym}(\overline{a})\}$.

From the non-generalization edges on the path from L to N in Fig. 1, we obtain transitions whose conditions contain $v'_{end} = v_{end}$ and $v'_{\mathtt{s}} = v_{\mathtt{s}}$. So $v_{\mathtt{s}}$ is increased by 1 in the transition from N to L and it remains the same in all other transitions of the graph's only cycle. Since the transition from N to L is only executed as long as $v_{\mathtt{s}} < v_{end}$ holds (where v_{end} is not changed by any transition), termination of the resulting ITS can easily be proved automatically.

Thm. 13 states the soundness of our approach for termination. If there is an infinite LLVM-computation $v_1 \to_{\mathsf{Vellvm}} v_2 \to_{\mathsf{Vellvm}} \ldots$ and v_1 is represented in the symbolic execution graph (i.e., there exists some c_1 with $\mathrm{TRANS}(v_1, c_1)$ that is represented by a_1), then Thm. 9 and 10 imply that there is a corresponding infinite path in the graph starting with the node a_1. We now show that then the ITS resulting from the graph is not terminating.

Theorem 13 (Termination of LLVM Programs) *Let $\mathcal{P}$ be an* LLVM *program with a complete symbolic execution graph $\mathcal{G}$. If $\mathcal{I}_{\mathcal{G}}$ is terminating, then $\mathcal{P}$ is also terminating for all* LLVM *states represented by the states in $\mathcal{G}$.*

Proof Let $c \to_{\mathsf{LLVM}} \overline{c}$, where $\mathcal{G}$ contains an abstract state a with $(as^c, mem^c) \models \sigma(\langle a \rangle_{SL})$ for some concrete instantiation σ. In the proof of Thm. 10, we showed that there is an abstract state $\overline{a}$ in $\mathcal{G}$ and a concrete instantiation $\overline{\sigma}$ with $(as^{\overline{c}}, mem^{\overline{c}}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$. To prove Thm. 13, it suffices to show $(a, \sigma) \to^+_{\mathcal{I}_{\mathcal{G}}} (\overline{a}, \overline{\sigma})$. By Thm. 9, then termination of $\mathcal{I}_{\mathcal{G}}$ also implies that there is no infinite LLVM evaluation according to the semantics of Vellvm.

(a) If a has an evaluation edge to $\overline{a}$, then $\overline{\sigma}(v) = \sigma(v)$ for all $v \in \mathcal{V}_{sym}(a)$. We show that

[11] For programs starting in states represented by an abstract state a_0, it would suffice to prove termination of all $\to_{\mathcal{I}}$-evaluations starting in ITS states of the form (a_0, σ).

[12] In the transition, we do not impose the additional constraints of $\langle \overline{a} \rangle$ on the post-variables $\mathcal{V}'$, since they are checked anyway in the next transition which starts in $\overline{a}$.

then $(a,\sigma) \to_{\mathcal{I}_\mathcal{G}} (\overline{a},\overline{\sigma})$. Note that $\mathcal{I}_\mathcal{G}$ has a transition $(a, \langle a \rangle \cup \{v' = v \mid v \in \mathcal{V}_{sym}(a)\}, \overline{a})$, so it suffices to show that $(\sigma \cup \overline{\sigma}')$ satisfies the condition of this transition. We have $(as^c, mem^c) \models \sigma(\langle a \rangle_{SL})$, and hence $(as^c, mem^c) \models \sigma(\langle a \rangle)$. Since σ is concrete (i.e., $\sigma(\langle a \rangle)$ does not contain variables), this implies $\models \sigma(\langle a \rangle)$ and thus, $(\sigma \cup \overline{\sigma}')\ (\langle a \rangle)$. Moreover, for all $v \in \mathcal{V}_{sym}(a)$, we have $(\sigma \cup \overline{\sigma}')(v') = \overline{\sigma}'(v') = \overline{\sigma}(v) = \sigma(v) = (\sigma \cup \overline{\sigma}')(v)$.

(b) If the path from a to $\overline{a}$ consists of a refinement and a subsequent evaluation edge, then $\overline{\sigma}(v) = \sigma(v)$ for all $v \in \mathcal{V}_{sym}(a)$. We show that then we have $(a,\sigma) \to^+_{\mathcal{I}_\mathcal{G}} (\overline{a},\overline{\sigma})$. To see this, note that in a's two successors, the knowledge base is extended by φ and $\neg\varphi$ for some formula φ, respectively. If $\models \sigma(\varphi)$, then let $\widetilde{a}$ be the successor with the knowledge base $\widetilde{KB} = KB \cup \{\varphi\}$. Otherwise, let $\widetilde{a}$ be the successor with the knowledge base $\widetilde{KB} = KB \cup \{\neg\varphi\}$. So in both cases, we have $\models \sigma(\widetilde{KB})$ and thus, $(as^c, mem^c) \models \sigma(\langle \widetilde{a} \rangle_{SL})$. Hence, $(\widetilde{a},\sigma) \to_{\mathcal{I}_\mathcal{G}} (\overline{a},\overline{\sigma})$ can be shown as in (a). As $\mathcal{I}_\mathcal{G}$ has a transition $(a, \langle a \rangle \cup \{v' = v \mid v \in \mathcal{V}_{sym}(a)\}, \widetilde{a})$, we can show $(a,\sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a},\sigma)$ as in (a).

(c) Let a have a generalization edge to $\widetilde{a}$ with instantiation μ and an evaluation edge from $\widetilde{a}$ to $\overline{a}$ with $\overline{\sigma}(v) = \sigma(\mu(v))$ for all $v \in \mathcal{V}_{sym}(\widetilde{a})$. We show $(a,\sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a}, \sigma \circ \mu) \to_{\mathcal{I}_\mathcal{G}} (\overline{a},\overline{\sigma})$. We first prove $(a,\sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a}, \sigma \circ \mu)$. Due to the edge from a to $\widetilde{a}$, $\mathcal{I}_\mathcal{G}$ has the transition $(a, \langle a \rangle \cup \{v' = \mu(v) \mid v \in \mathcal{V}_{sym}(\widetilde{a})\}, \widetilde{a})$, and we show that $(\sigma \cup (\sigma \circ \mu)')$ satisfies the condition of this transition. We have $(as^c, mem^c) \models \sigma(\langle a \rangle_{SL})$, and hence $(as^c, mem^c) \models \sigma(\langle a \rangle)$, from which $\models \sigma(\langle a \rangle)$ follows and finally $\models (\sigma \cup (\sigma \circ \mu)')\ (\langle a \rangle)$. Moreover, for all $v \in \mathcal{V}_{sym}(\widetilde{a})$, we have $(\sigma \cup (\sigma \circ \mu)')(v') = (\sigma \circ \mu)'(v') = \sigma(\mu(v)) = (\sigma \cup (\sigma \circ \mu)')(\mu(v))$. Now we show $(\widetilde{a}, \sigma \circ \mu) \to_{\mathcal{I}_\mathcal{G}} (\overline{a},\overline{\sigma})$. As there is a generalization edge from a to $\widetilde{a}$ with the instantiation μ, we know that $\models \langle a \rangle_{SL} \Rightarrow \mu(\langle \widetilde{a} \rangle_{SL})$. Thus, $(as^c, mem^c) \models \sigma(\langle a \rangle_{SL})$ implies $(as^c, mem^c) \models (\sigma \circ \mu)(\langle \widetilde{a} \rangle_{SL})$. Hence, $(\widetilde{a}, \sigma \circ \mu) \to_{\mathcal{I}_\mathcal{G}} (\overline{a},\overline{\sigma})$ follows as in (a).

(d) Finally, let a have a generalization edge to $\widetilde{a}$ with the instantiation μ, and there is a path consisting of a refinement and an evaluation edge from $\widetilde{a}$ to $\overline{a}$, where $\overline{\sigma}(v) = \sigma(\mu(v))$ for all $v \in \mathcal{V}_{sym}(\widetilde{a})$. We show that then $(a,\sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a}, \sigma \circ \mu) \to^+_{\mathcal{I}_\mathcal{G}} (\overline{a},\overline{\sigma})$. Here, $(a,\sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a}, \sigma \circ \mu)$ follows as in (c), and $(\widetilde{a}, \sigma \circ \mu) \to^+_{\mathcal{I}_\mathcal{G}} (\overline{a},\overline{\sigma})$ can be proved as in (b). □

6 Limitations, Related Work, Experiments, and Conclusion

We have developed a new approach to prove memory safety and termination of C (resp. LLVM) programs with explicit pointer arithmetic and memory access. It relies on a representation of abstract program states which allows an easy automation of the rules for symbolic execution (by using standard SMT solving to check the first-order conditions of these rules). Moreover, this representation is suitable for generalizing abstract states and for generating integer transition systems. In this way, LLVM programs are translated fully automatically into ITSs amenable to automated termination analysis.

Limitations and Future Work. To simplify the formalization of our approach, we have not discussed global variables, which our implementation supports. In line with most other techniques, we currently do not handle the case that calls to `malloc` may fail, and we also assume that reading from uninitialized (but allocated) heap locations is safe and yields an arbitrary value. Our method could easily be adapted to lift these limitations. Furthermore, in this paper we disregard integer overflows and treat all integer types except `i1` as the infinite set $\mathbb{Z}$. An extension of our approach to bounded integers can be found in [44].

In the paper, we only gave rules for a subset of all LLVM instructions. Our implementation handles several more instructions, but there exist instructions (or cases of instructions) where our implementation does not yet contain suitable rules for symbolic execution. In particular, our abstract domain currently does not handle `undef` values, floating point values, or vectors, and consequently, all corresponding instructions are unsupported.[13]

When encountering an instruction that cannot be handled, the symbolic execution can still continue by removing all potentially affected knowledge. The same holds if one cannot prove all conditions of a symbolic execution rule. It often suffices to remove all information about the value that is computed by the instruction, e.g., for floating point operations.

In this paper, we did not treat recursive programs and we also did not present any method to prove that an LLVM program is *not* memory safe or does *not* terminate. However, we are working on extending our approach accordingly and our implementation already contains some support for recursion and non-termination by adapting our approaches for recursion and non-termination of Java programs [12,13]. Another direction for further work could be to embed our analysis into a *Counter-Example-Guided Abstraction Refinement (CEGAR)* loop [24] to disprove termination or memory safety, and to automatically refine the abstraction.

Finally, we cannot yet analyze C programs using inductive data structures defined via "`struct`". However, in the future, we want to adapt our corresponding technique for termination analysis of Java programs [11,12,14,63]. Instead of ITSs, here one generates integer term rewrite systems [35,37] from the symbolic execution graph, where data objects are transformed into *terms* to represent them in a precise way.

Moreover, we would like to improve the scalability of our approach by a compositional treatment of LLVM functions. There are several tools based on separation logic which support local modular reasoning for shape analysis and the verification of memory safety, e.g., [19–21]. Moreover, compositional approaches for termination analysis were developed in [22,28, 70], based on [65]. Combining such approaches with the byte-precise handling of explicit low-level pointer arithmetic for termination analysis will be the subject of further work.

Related Work and Experimental Evaluation. There exist numerous other methods and tools for termination analysis of imperative programs (e.g., 2LS [22], ARMC [64], ARTMC [41, 45], COSTA [2], CppInv [50], Ctrl [47,48], Cyclist [16], FuncTion [32], HipTNT+ [52], Juggernaut [30], Julia [66], KITTeL [35], L2CA [10], LoopFrog [70], SeaHorn [71], Sonar/ Mutant [5,4],TAN [49],Terminator [26–28],Termite [39],TRex [42],T2 [15], Ultimate [43], ...). Until very recently, most other approaches did not handle the heap at all, or supported dynamic data structures by an abstraction to integers (e.g., to represent sizes or lengths) or to terms (representing finite unravelings). In particular, most tools failed when the control flow depends on explicit pointer arithmetic and on detailed information about the contents of addresses. While our approach was inspired by our previous work on termination of Java, in the current paper we extend these techniques to prove termination and memory safety of programs with explicit pointer arithmetic. This requires a fundamentally new approach, as pointer arithmetic cannot be expressed in the Java-based techniques of [11,12,14,63].

We implemented our technique in the termination prover AProVE [38,68], which uses the SMT solvers Yices [34] and Z3 [31] in the back-end. AProVE participated very successfully in the *International Competition on Software Verification (SV-COMP)*[14] at *TACAS* and in the *International Termination Competition (TermComp)*,[15] both of which feature categories for termination of C programs since 2014.

To evaluate AProVE empirically, we compare its performance with other tools on all 631 programs from the *Termination* category of *SV-COMP 2016*. For termination of low-level C programs, one also has to ensure their memory safety. Approaches for proving memory

[13] The instructions supported by our implementation are `icmp` (`eq`,`ne`,`sgt`,`sge`,`slt`,`sle`, `ugt`,`uge`,`ult`,`ule`), `add`, `sub`, `mul`, `sdiv`, `srem`, `urem`, `and`, `or`, `xor`, `shl`, `ashr`, `lshr`, `call`, `br`, `bitcast`, `ptrtoint`, `trunc`, `sext`, `zext`, `getelementptr` (with at most 2 parameters), `select`, `phi`, `ret`, `alloca`, `load`, and `store`.

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

[15] `http://termination-portal.org/wiki/Termination_Competition`

safety of programs with pointer arithmetic were proposed in [19,40], for example. However, while there exist several tools to prove memory safety of C programs, many of them do not handle explicit byte-accurate pointer arithmetic (e.g., Thor [55,56] or SLAyer [6]) or require the user to provide the needed loop invariants (as in the Jessie plug-in of Frama-C [58]). In contrast, our approach can prove memory safety of such algorithms automatically. More precisely, for the 631 programs in our collection, AProVE shows memory safety for 547 examples. In contrast, the most powerful tool for *verifying* memory safety at *SV-COMP 2016* (Predator [33]) proves memory safety for 431 examples (see [3] for details). However, this comparison is not very meaningful, since Predator considers bounded integers, whereas AProVE assumes integers to be unbounded. Thus, the resulting notions of memory safety are incomparable. Moreover, there exist several tools to *disprove* memory safety (e.g., Predator, CPAchecker [54], and LLBMC [36]). In contrast, AProVE can only prove, but not *disprove* memory safety, since our symbolic execution graph *over-approximates* all possible program runs. So the occurrence of *ERR* in our graph does not imply that the program is really unsafe.

To evaluate the power of our approach for proving termination, we compared AProVE to the most powerful other tools (Ultimate and SeaHorn) from the *Termination* category of *SV-COMP 2016*, and to HipTNT+. (AProVE, Ultimate, and HipTNT+ were the most powerful tools in the C category of *TermComp 2015* and in the *Termination* category of *SV-COMP 2015*.) In addition, we included the tool KITTeL in our evaluation, which operates on LLVM as well. Recall that in the present paper, we only introduced techniques to prove (but not to disprove) termination of programs. Therefore, to evaluate the contributions of the present paper, we excluded those C programs from our evaluation that are known to be non-terminating. This resulted in 498 programs.[16]

On the side, we show the performance of the tools for a time limit of 900 seconds per example. For AProVE, Ultimate, and SeaHorn, we used the results of *SV-COMP 2016*. The other tools were run on an Intel Xeon with 4 cores clocked at 2.33 GHz each and 16 GB of RAM. "**YES**" gives the number of examples where termination could be proved, "**MAYBE**" states how often the tool could not find a proof within 900 seconds, and "**Runtime**" is the average time in seconds for those examples where the tool proved termination. The table shows that in these experiments, AProVE is the most powerful tool for proving termination of C programs. On the other hand, since AProVE constructs symbolic execution graphs to prove memory safety and to infer suitable invariants needed for termination proofs, its runtime is often higher. For details on the experiments and to access our implementation in AProVE via a web interface, we refer to [3].

Tool	**YES**	**MAYBE**	**Runtime**
AProVE	409	89	39.9
Ultimate	392	106	24.6
HipTNT+	310	188	1.3
SeaHorn	245	253	10.9
KITTeL	220	278	0.2

Acknowledgements We are grateful to the developers of the other tools for termination or memory safety [33,35,43,52,71] for their help with the experiments.

References

1. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *Proc. CAV '12*.
2. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java Bytecode. In *Proc. FMOODS '08*.
3. AProVE: `http://aprove.informatik.rwth-aachen.de/eval/PointerJournal/`.

[16] As mentioned above, we also started implementing support for non-termination in AProVE. When running the tools on all 631 C examples, AProVE proves termination for 409 and non-termination for 91 examples. Ultimate shows termination for 392 and non-termination for 111 programs. Finally, HipTNT+ proves termination in 312 and non-termination in 107 cases. Again, the detailed results can be found at [3].

4. J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV '06*.
5. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance analyses from invariance analyses. In *Proc. POPL '07*.
6. J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *Proc. CAV '11*.
7. Y. Bertot and P. Castéran. *CoqArt*. Springer, 2004.
8. F. Blanqui and A. Koprowski. CoLoR: A Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Structures in Comp. Sc.*, 4:827–859, 2011.
9. M. Bodin, T. Jensen, and A. Schmitt. Certified abstract interpretation with pretty-big-step semantics. In *Proc. CPP '15*.
10. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. *Formal Methods in System Design*, 38(2):158–192, 2011.
11. M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, LNAI 6463, 2010.
12. M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proc. RTA '11*.
13. M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In *Proc. FoVeOOS '11*.
14. M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. CAV '12*.
15. M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *Proc. CAV '13*.
16. J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In *Proc. SAS '14*.
17. D. Cachera and D. Pichardie. A certified denotational abstract interpreter. In *Proc. ITP '10*.
18. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI '08*.
19. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Proc. SAS '06*.
20. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Space invading systems code. In *Proc. LOPSTR '08*.
21. C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *Proc. NFM '11*.
22. H. Y. Chen, C. David, D. Kroening, P. Schrammel, and N. Wächter. Synthesising interprocedural bit-precise termination proofs. In *Proc. ASE '15*.
23. Clang compiler: http://clang.llvm.org.
24. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.
25. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Automated certified proofs with CiME3. In *Proc. RTA '11*.
26. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Proc. SAS '05*.
27. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*.
28. B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009.
29. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL '78*.
30. C. David, D. Kroening, and M. Lewis. Unrestricted termination and non-termination arguments for bit-vector programs. In *Proc. ESOP '15*.
31. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*.
32. V. D'Silva and C. Urban. Conflict-driven conditional termination. In *Proc. CAV '15*.
33. K. Dudka, P. Peringer, and T. Vojnar. Predator: A shape analyzer based on symbolic memory graphs (competition contribution). In *Proc. TACAS '14*.
34. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf.
35. S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In *Proc. RTA '11*.
36. S. Falke, F. Merz, and C. Sinz. LLBMC: Improved bounded model checking of C using LLVM (competition contribution). In *Proc. TACAS '13*.
37. C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. RTA '09*.
38. J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE.

In *Proc. IJCAR '14*.
39. L. Gonnord, D. Monniaux, and G. Radanne. Synthesis of ranking functions using extremal counterexamples. In *Proc. PLDI '15*.
40. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *Proc. CAV '07*.
41. P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving termination of tree manipulating programs. In *Proc. ATVA '07*.
42. W. R. Harris, A. Lal, A. Nori, and S. K. Rajamani. Alternation for termination. In *Proc. SAS '10*.
43. M. Heizmann, J. Hoenicke, J. Leike, and A. Podelski. Linear ranking for linear lasso programs. In *Proc. ATVA '13*.
44. J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Proving termination of programs with bitvector arithmetic by symbolic execution. In *Proc. SEFM '16*.
45. R. Iosif and A. Rogalewicz. Automata-based termination proofs. *Comp. and Inf.*, 32(4):739–775, 2013.
46. J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *Proc. POPL '15*.
47. C. Kop and N. Nishida. Automatic constrained rewriting induction towards verifying procedural programs. In *Proc. APLAS '14*.
48. C. Kop and N. Nishida. Constrained Term Rewriting tooL. In *Proc. LPAR '15*.
49. D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination analysis with compositional transition invariants. In *Proc. CAV '10*.
50. D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using Max-SMT. In *Proc. FMCAD '13*.
51. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO '04*.
52. T. C. Le, S. Qin, and W. Chin. Termination and non-termination specification inference. In *Proc. PLDI '15*.
53. LLVM reference manual. `http://llvm.org/docs/LangRef.html`.
54. S. Löwe, M. Mandrykin, and P. Wendler. CPAchecker with sequential combination of explicit-value analyses and predicate analyses (competition contribution). In *Proc. TACAS '14*.
55. S. Magill. *Instrumentation Analysis: An Automated Method for Producing Numeric Abstractions of Heap-Manipulating Programs*. PhD thesis, CMU, Pittsburgh, PA, USA, 2010. Available at `http://www.cs.cmu.edu/~smagill/papers/thesis.pdf`.
56. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. POPL '10*.
57. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
58. Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *J. Symb. Comput.*, 45(11), 2010.
59. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6), 2006.
60. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer, 2002.
61. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. CSL '01*.
62. `http://fxr.watson.org/fxr/source/lib/libsa/strlen.c?v=OPENBSD`.
63. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*.
64. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *Proc. PADL '07*.
65. T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. POPL '95*.
66. F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java Bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.
67. T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, and P. Schneider-Kamp. Proving termination and memory safety for programs with pointer arithmetic. In *Proc. IJCAR '14*.
68. T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. AProVE: Termination and memory safety of C programs (competition contribution). In *Proc. TACAS '15*.
69. R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. TPHOLs '09*.
70. A. Tsitovich, N. Sharygina, C. M. Wintersteiger, and D. Kroening. Loop summarization and termination analysis. In *Proc. TACAS '11*.
71. C. Urban, A. Gurfinkel, and T. Kahsai. Synthesizing ranking functions from bits and pieces. In *Proc. TACAS '16*.
72. Wikibooks C Programming: `http://en.wikibooks.org/wiki/C_Programming/`.
73. J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM IR for verified program transformations. In *Proc. POPL '12*.