RWTH Aachen

Department of Computer Science
Technical Report

Automated Termination Analysis for Programs with Pointer Arithmetic

Thomas Ströder, Jürgen Giesl, Marc Brockschmidt,
Florian Frohn, Carsten Fuhs, Jera Hensel,
Peter Schneider-Kamp

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2014-05

RWTH Aachen · Department of Computer Science · September 2015 (revised)

Automated Termination Analysis for Programs with Pointer Arithmetic*

Thomas Ströder[1], Jürgen Giesl[1], Marc Brockschmidt[2], Florian Frohn[1], Carsten Fuhs[3], Jera Hensel[1], and Peter Schneider-Kamp[4]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
[2] Microsoft Research Cambridge, UK
[3] Dept. of Computer Science, University College London, UK
[4] IMADA, University of Southern Denmark, Denmark

Abstract. Proving termination automatically for programs with explicit pointer arithmetic is 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 represents all possible runs of the 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.

1 Introduction

Consider the following standard C implementation of `strlen` [23, 29], computing the length of the string at 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 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. To avoid handling the intricacies of C, we analyze programs in the platform-independent intermediate representation (IR) of the LLVM compilation framework [17]. Our approach works in three steps: First, a *symbolic execution graph* is created

* Supported by DFG grant GI 274/6-1 and Research Training Group 1298 (*AlgoSyn*).

that represents an over-approximation of all possible program runs. We present our abstract domain based on *separation logic* [22] and the automated construction of such graphs in Sect. 2. In this step, we handle all issues related to memory, and in particular prove memory safety of our input program. In Sect. 3, 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. 4, 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.

2 From LLVM to Symbolic Execution Graphs

In Sect. 2.1, we introduce concrete LLVM states and *abstract* states that represent *sets* of concrete states, cf. [9]. Based on this, Sect. 2.2 shows how to construct symbolic execution graphs automatically. Sect. 2.3 presents our algorithm to *generalize* states, needed to always obtain *finite* symbolic execution graphs.

To simplify the presentation, we restrict ourselves to a single LLVM function without function calls and to types of the form in (for n-bit integers), in* (for pointers to values of type in), in**, in***, 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. Moreover, we assume a 1 byte data alignment (i.e., values may be stored at any address).

2.1 Abstract Domain

Consider the `strlen` function from Sect. 1. In the corresponding LLVM code,[5] `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. 2.2.

```
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 }
```

Concrete LLVM states consist of the program counter, the values of local variables, and the state of the memory. The program counter is a 3-tuple $(\mathtt{b}_{prev}, \mathtt{b}, i)$, where $\mathtt{b}$ is the name of the current basic block, $\mathtt{b}_{prev}$ is the previously executed

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

block,[6] and i is the index of the next instruction. So if $Blks$ is the set of all basic blocks, then the set of code positions is $Pos = (Blks \cup \{\varepsilon\}) \times Blks \times \mathbb{N}$. We represent assignments to the local program variables $\mathcal{V}_\mathcal{P}$ (e.g., $\mathcal{V}_\mathcal{P} = \{\mathtt{str}, \mathtt{c0}, \ldots\}$) as functions $s : \mathcal{V}_\mathcal{P} \to \mathbb{Z}$. The state of the memory is represented by a partial function $m : \mathbb{N}_{>0} \to \mathbb{Z}$ with finite domain that maps addresses to integer values. So a concrete LLVM state is a 3-tuple $(p, s, m) \in Pos \times (\mathcal{V}_\mathcal{P} \to \mathbb{Z}) \times (\mathbb{N}_{>0} \to \mathbb{Z})$.

To model violations of memory safety, we introduce a special state ERR to be reached when accessing non-allocated memory. So (p, s, m) denotes only memory safe states where all addresses in m's domain are allocated. Let $\to_{\mathsf{LLVM}}$ be LLVM's evaluation relation on concrete states, i.e., $(p, s, m) \to_{\mathsf{LLVM}} (\overline{p}, \overline{s}, \overline{m})$ holds iff (p, s, m) evaluates to $(\overline{p}, \overline{s}, \overline{m})$ by executing one LLVM instruction. Similarly, $(p, s, m) \to_{\mathsf{LLVM}} ERR$ means that the instruction at position p accesses an address where m is undefined. An LLVM program is *memory safe* for (p, s, m) iff there is no evaluation $(p, s, m) \to^+_{\mathsf{LLVM}} ERR$, where $\to^+_{\mathsf{LLVM}}$ is the transitive closure of $\to_{\mathsf{LLVM}}$.

To formalize *abstract* states that stand for sets of concrete states, we use a fragment of *separation logic* [22]. Here, an infinite set of symbolic variables $\mathcal{V}_{sym}$ with $\mathcal{V}_{sym} \cap \mathcal{V}_\mathcal{P} = \varnothing$ can be used in place of concrete integers. We represent abstract states as tuples (p, LV, KB, AL, PT). Again, $p \in Pos$ is the program counter. The function $LV : \mathcal{V}_\mathcal{P} \to \mathcal{V}_{sym}$ maps every local variable to a symbolic variable. To ease the generalization of states in Sect. 2.3, we require injectivity of LV. The *knowledge base* $KB \subseteq QF\_IA(\mathcal{V}_{sym})$ is a set of pure quantifier-free first-order formulas that express integer arithmetic properties of $\mathcal{V}_{sym}$.

The *allocation list* AL contains expressions of the form $alloc(v_1,v_2)$ 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 are allocated. Finally, PT is a set of "points-to" atoms $v_1 \hookrightarrow_{\mathtt{ty}} v_2$ where $v_1, v_2 \in \mathcal{V}_{sym}$ and $\mathtt{ty}$ is an LLVM type. This means that the value v_2 of type $\mathtt{ty}$ is stored at the address v_1. Let $size(\mathtt{ty})$ be the number of bytes required for values of type $\mathtt{ty}$ (e.g., $size(\mathtt{i8}) = 1$ and $size(\mathtt{i32}) = 4$). As each memory cell stores one byte, $v_1 \hookrightarrow_{\mathtt{i32}} v_2$ means that v_2 is stored in the four cells at the addresses $v_1, \ldots, v_1{+}3$.

Definition 1 (Abstract States). Abstract states *have the form* (p, LV, KB, AL, PT) *where* $p \in Pos$, $LV : \mathcal{V}_\mathcal{P} \to \mathcal{V}_{sym}$ *is injective,* $KB \subseteq QF\_IA(\mathcal{V}_{sym})$, $AL \subseteq \{alloc(v_1, v_2) \mid v_1, v_2 \in \mathcal{V}_{sym}\}$, *and* $PT \subseteq \{(v_1 \hookrightarrow_{\mathtt{ty}} v_2) \mid v_1, v_2 \in \mathcal{V}_{sym}, \mathtt{ty}$ *is an* LLVM *type*$\}$. *Additionally, there is a state ERR for violations of memory safety.*

We often identify LV with the set of equations $\{\mathtt{x} = LV(\mathtt{x}) \mid \mathtt{x} \in \mathcal{V}_\mathcal{P}\}$ and extend LV to a function from $\mathcal{V}_\mathcal{P} \uplus \mathbb{Z}$ to $\mathcal{V}_{sym} \uplus \mathbb{Z}$ by defining $LV(z) = z$ for all $z \in \mathbb{Z}$. As an example, consider the following abstract state for our `strlen` program:

$$\begin{array}{llll} (\ (\varepsilon, \mathtt{entry}, 0), & \{\mathtt{str} = u_{\mathtt{str}}, \ldots, \mathtt{size} = u_{\mathtt{size}}\}, & \{z = 0\}, & \\ & \{alloc(u_{\mathtt{str}}, v_{end})\}, & \{v_{end} \hookrightarrow_{\mathtt{i8}} z\}\ ). & \end{array} \tag{$\dagger$}$$

It represents states at the beginning of the `entry` block, where $LV(\mathtt{x}) = u_{\mathtt{x}}$ for all $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, the memory cells between $LV(\mathtt{str}) = u_{\mathtt{str}}$ and v_{end} are allocated, and the value at the address v_{end} is z (where the knowledge base implies $z = 0$).

To define the semantics of abstract states a, we introduce the formulas $\langle a \rangle_{SL}$ and $\langle a \rangle_{FO}$. The separation logic formula $\langle a \rangle_{SL}$ defines which concrete states are

[6] $\mathtt{b}_{prev}$ is needed for `phi` instructions (cf. Sect. 2.2). In the beginning, we set $\mathtt{b}_{prev} = \varepsilon$.

represented by a. The first-order formula $\langle a\rangle_{FO}$ is used to construct symbolic execution graphs, allowing us to use standard SMT solving for all reasoning in our approach. Moreover, we also use $\langle a\rangle_{FO}$ for the subsequent generation of integer transition systems from the symbolic execution graphs. In addition to KB, $\langle a\rangle_{FO}$ states that the expressions $alloc(v_1, v_2) \in AL$ represent disjoint intervals and that two addresses must be different if they point to different values in PT.

In $\langle a\rangle_{SL}$, we combine the elements of AL with the separating conjunction "$*$" to ensure that different allocated memory blocks are disjoint. Here, as usual $\varphi_1 * \varphi_2$ means that φ_1 and φ_2 hold for disjoint parts of the memory. In contrast, the elements of PT are combined by the ordinary conjunction "$\wedge$". So $v_1 \hookrightarrow_{\mathtt{ty}} v_2 \in PT$ 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 memory addresses.

Definition 2 (Representing States by Formulas). *For $v_1, v_2 \in \mathcal{V}_{sym}$, let $\langle alloc(v_1, v_2)\rangle_{SL} = v_1 \leq v_2 \ \wedge\ (\forall x.\exists y.\ (v_1 \leq x \leq v_2) \Rightarrow (x \hookrightarrow y))$. Due to the two's complement representation, for any* LLVM *type* `ty`*, we define $\langle v_1 \hookrightarrow_{\mathtt{ty}} v_2\rangle_{SL} =$*

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

where $v_3 \in \mathcal{V}_{sym}$ is fresh. Here,[7] $\langle v_1 \hookrightarrow_0 v_3\rangle_{SL} = true$ and $\langle v_1 \hookrightarrow_{n+1} v_3\rangle_{SL} = v_1 \hookrightarrow (v_3\ mod\ 256) \ \wedge\ \langle\, (v_1 + 1) \hookrightarrow_n (v_3\ div\ 256)\,\rangle_{SL}$. Then $a = (p, LV, KB, AL, PT)$ is represented by[8] $\langle a\rangle_{SL} = LV \wedge KB \wedge (\ast_{\varphi \in AL}\ \langle\varphi\rangle_{SL}) \wedge (\bigwedge_{\varphi \in PT}\ \langle\varphi\rangle_{SL})$.

Moreover, the following first-order information on $\mathcal{V}_{sym}$ is deduced from an abstract state $a = (p, LV, KB, AL, PT)$. Let $\langle a\rangle_{FO}$ be the smallest set with

$$\begin{aligned}\langle a\rangle_{FO} = KB \ \cup\ & \{v_1 \leq v_2 \mid alloc(v_1, v_2) \in AL\} \ \cup \\ & \{v_2 < w_1 \vee w_2 < v_1 \mid alloc(v_1, v_2), alloc(w_1, w_2) \in AL,\ (v_1, v_2) \neq (w_1, w_2)\} \cup \\ & \{v_1 \neq w_1 \mid (v_1 \hookrightarrow_{\mathtt{ty}} v_2), (w_1 \hookrightarrow_{\mathtt{ty}} w_2) \in PT \text{ and } \models \langle a\rangle_{FO} \Rightarrow v_2 \neq w_2\}.\end{aligned}$$

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 $\mathcal{V}_{\mathcal{P}}$. Instantiations are extended to formulas in the usual way, i.e., $\sigma(\varphi)$ instantiates every $v \in \mathcal{V}_{sym}$ that occurs free in φ by $\sigma(v)$. An instantiation is called *concrete* iff $\sigma(v) \in \mathbb{Z}$ for all $v \in \mathcal{V}_{sym}$. Then an abstract state a at position p represents those concrete states (p, s, m) where (s, m) is a *model* of $\sigma(\langle a\rangle_{SL})$ for a concrete instantiation σ of the symbolic variables. So for example, the abstract state (†) on the previous page represents all concrete states $((\varepsilon, \mathtt{entry}, 0), s, m)$ where m is a memory that stores a string at the address $s(\mathtt{str})$.[9]

[7] We assume a little-endian data layout (where least significant bytes are stored in the lowest address). 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).

[8] We identify *sets* of first-order formulas $\{\varphi_1, ..., \varphi_n\}$ with their conjunction $\varphi_1 \wedge ... \wedge \varphi_n$.

[9] The reason is that then there is an address $end \geq s(\mathtt{str})$ such that $m(end) = 0$ and m is defined for all numbers between $s(\mathtt{str})$ and end. Hence, $(s, m) \models \sigma(\langle a\rangle_{SL})$ holds for an instantiation with $\sigma(u_{\mathtt{x}}) = s(\mathtt{x})$ for all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $\sigma(v_{end}) = end$, and $\sigma(z) = 0$.

It remains to define when (s, m) is a *model* of a formula from our fragment of separation logic. For $s : \mathcal{V}_\mathcal{P} \to \mathbb{Z}$ and any formula φ, let $s(\varphi)$ result from replacing all $\mathtt{x} \in \mathcal{V}_\mathcal{P}$ in φ by $s(\mathtt{x})$. Note that by construction, local variables `x` are never quantified in our formulas. Then we define $(s, m) \models \varphi$ iff $m \models s(\varphi)$.

We now define $m \models \psi$ for formulas ψ that may still contain symbolic variables from $\mathcal{V}_{sym}$ (this is needed for Sect. 2.2). As usual, all free variables $v_1, \dots, v_n$ in ψ are implicitly universally quantified, i.e., $m \models \psi$ iff $m \models \forall v_1, \dots v_n.\, \psi$. The semantics of arithmetic operations and relations and of first-order connectives and quantifiers is as usual. In particular, we define $m \models \forall v.\, \psi$ iff $m \models \sigma(\psi)$ 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 $z_1, z_2 \in \mathbb{Z}$, let $m \models z_1 \hookrightarrow z_2$ hold iff $m(z_1) = z_2$.[10] The semantics of $*$ is defined as usual in separation logic: For two partial functions $m_1, m_2 : \mathbb{N}_{>0} \to \mathbb{Z}$, we write $m_1 \bot m_2$ to indicate that the domains of m_1 and m_2 are disjoint and $m_1 \cdot m_2$ denotes the union of m_1 and m_2. Then $m \models \varphi_1 * \varphi_2$ iff there exist $m_1 \bot m_2$ such that $m = m_1 \cdot m_2$ where $m_1 \models \varphi_1$ and $m_2 \models \varphi_2$.

As usual, "$\models \varphi$" means that φ is a tautology, i.e., that $(s, m) \models \varphi$ holds for any $s : \mathcal{V}_\mathcal{P} \to \mathbb{Z}$ and $m : \mathbb{N}_{>0} \to \mathbb{Z}$. Clearly, $\models \langle a \rangle_{SL} \Rightarrow \langle a \rangle_{FO}$, i.e., $\langle a \rangle_{FO}$ contains first-order information that holds in every concrete state represented by a.

2.2 Constructing 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. Other instructions can be handled in a similar way, cf. App. A. In contrast to other formalizations of LLVM's operational semantics [30], our rules operate on *abstract* instead of concrete states to allow a *symbolic* execution of LLVM. In particular, we also have rules for refining and generalizing abstract states.

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 $AL = \{alloc(u_{\mathtt{str}}, v_{end})\}$, which stays the same in all states in this example. We also abbreviated parts of LV, KB, PT by "...". Instead of $v_{end} \hookrightarrow_{\mathtt{i8}} z$ 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 $p\colon ins$ 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 abstract 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 $\mathtt{ad}, \dots, \mathtt{ad} + size(\mathtt{ty}) - 1$ are allocated, i.e., if there is an $alloc(v_1, v_2) \in AL$ such that $\langle a \rangle \Rightarrow (v_1 \leq LV(\mathtt{ad}) \wedge$

[10] We use "$\hookrightarrow$" instead of "$\mapsto$" in separation logic, since $m \models z_1 \mapsto z_2$ would imply that $m(z)$ is undefined for all $z \neq z_1$. This would be inconvenient in our formalization, since PT usually only contains information about a *part* of the allocated memory.

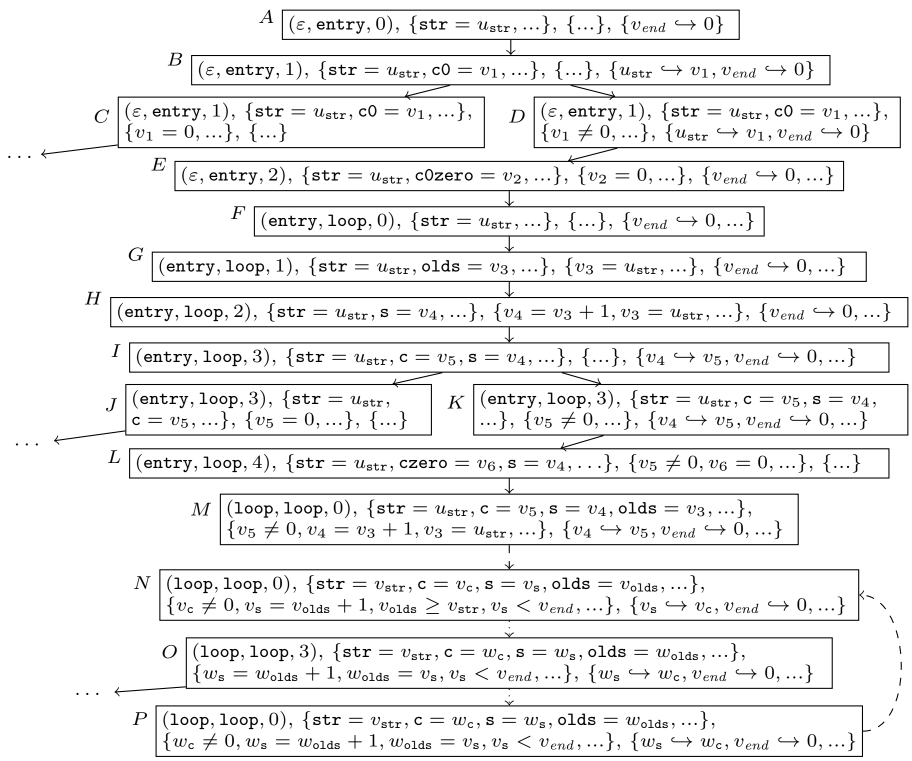

Fig. 1. Symbolic execution graph for `strlen`

$LV(\texttt{ad}) + size(\texttt{ty}) - 1 \leq v_2)$ is valid. Then, we reach a new abstract state where the previous position $p = (\texttt{b}_{prev}, \texttt{b}, i)$ is updated to the position $p^+ = (\texttt{b}_{prev}, \texttt{b}, i+1)$ of the next instruction in the same basic block, and we set $LV(\texttt{x}) = w$ for a fresh $w \in \mathcal{V}_{sym}$. If we already know the value at the address `ad` (i.e., if there are $w_1, w_2 \in \mathcal{V}_{sym}$ with $\models \langle a \rangle \Rightarrow (LV(\texttt{ad}) = w_1)$ and $w_1 \hookrightarrow_{\texttt{ty}} w_2 \in PT$) then we add $w = w_2$ to KB. Otherwise, we add $LV(\texttt{ad}) \hookrightarrow_{\texttt{ty}} w$ to PT. We used this rule to obtain B from A in Fig. 1. In a similar way, one can also formulate a rule for instructions that `store` a value at some address in the memory (cf. App. A).

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

$$\frac{(p,\ LV \uplus \{\texttt{x} = v\},\ KB,\ AL,\ PT)}{(p^+,\ LV \uplus \{\texttt{x} = w\},\ KB \cup \{w = w_2\},\ AL,\ PT)} \quad \text{if}$$

- there is $alloc(v_1, v_2) \in AL$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV(\texttt{ad}) \wedge LV(\texttt{ad}) + size(\texttt{ty}) - 1 \leq v_2)$,
- there are $w_1, w_2 \in \mathcal{V}_{sym}$ with $\models \langle a \rangle \Rightarrow (LV(\texttt{ad}) = w_1)$ and $w_1 \hookrightarrow_{\texttt{ty}} w_2 \in PT$,
- $w \in \mathcal{V}_{sym}$ is fresh

$$\frac{(p,\ LV \uplus \{\texttt{x} = v\},\ KB,\ AL,\ PT)}{(p^+,\ LV \uplus \{\texttt{x} = w\},\ KB,\ AL,\ PT \cup \{LV(\texttt{ad}) \hookrightarrow_{\texttt{ty}} w\})} \quad \text{if}$$

- there is $alloc(v_1, v_2) \in AL$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV(\texttt{ad}) \wedge LV(\texttt{ad}) + size(\texttt{ty}) - 1 \leq v_2)$,
- there are no $w_1, w_2 \in \mathcal{V}_{sym}$ with $\models \langle a \rangle \Rightarrow (LV(\texttt{ad}) = w_1)$ and $w_1 \hookrightarrow_{\texttt{ty}} w_2 \in PT$,
- $w \in \mathcal{V}_{sym}$ is fresh

If `load` accesses an address that was not allocated, then memory safety is violated and we reach the *ERR* state.

`load` **from unallocated memory** (p : **"**`x = load ty* ad`**" with** $\mathtt{x}, \mathtt{ad} \in \mathcal{V}_{\mathcal{P}}$**)**
$$\frac{(p, LV, KB, AL, PT)}{ERR} \quad \text{if}$$
there is no $alloc(v_1,v_2) \in AL$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV(\mathtt{ad}) \wedge LV(\mathtt{ad}) + size(\mathtt{ty}) - 1 \leq v_2)$

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`. So 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`.[11]

`icmp` **(**p : **"**`x = icmp eq ty` t_1`,` t_2**" with** $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ **and** $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$**)**
$$\frac{(p,\ LV \uplus \{\mathtt{x} = v\},\ KB,\ AL,\ PT)}{(p^+,\ LV \uplus \{\mathtt{x} = w\},\ KB \cup \{w = 1\},\ AL,\ PT)} \quad \begin{array}{l}\text{if} \models \langle a \rangle \Rightarrow (LV(t_1) = LV(t_2))\\ \text{and } w \in \mathcal{V}_{sym} \text{ is fresh}\end{array}$$
$$\frac{(p,\ LV \uplus \{\mathtt{x} = v\},\ KB,\ AL,\ PT)}{(p^+,\ LV \uplus \{\mathtt{x} = w\},\ KB \cup \{w = 0\},\ AL,\ PT)} \quad \begin{array}{l}\text{if} \models \langle a \rangle \Rightarrow (LV(t_1) \neq LV(t_2))\\ \text{and } w \in \mathcal{V}_{sym} \text{ is fresh}\end{array}$$

The previous rule is only applicable if *KB* contains enough information to evaluate the 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.[12]

refining abstract states (p : "`x = icmp eq ty` t_1`,`t_2", $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$)
$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p,\ LV,\ KB \cup \{\varphi\},\ AL,\ PT)\ \mid\ (p,\ LV,\ KB \cup \{\neg\varphi\},\ AL,\ PT)}$$
if φ is $LV(t_1) = LV(t_2)$ and both $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$

For example, in state B of Fig. 1, we evaluate "`c0zero = icmp eq i8 c0, 0`", i.e., we check whether 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 conditional branching instruction `br` is very similar to `icmp`. To evaluate "`br i1` t, `label` $\mathtt{b}_1$, `label` $\mathtt{b}_2$", 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 with block $\mathtt{b}_1$ resp. $\mathtt{b}_2$. This rule allows us to create the successor F of E, where we jump to the block `loop`.

[11] Other integer comparisons (for $<$, $\leq$, ...) are handled analogously.
[12] Analogous refinement rules can also be used for other conditional LLVM instructions.

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

$$\frac{(p, LV, KB, AL, PT)}{((\mathtt{b}, \mathtt{b}_1, 0), LV, KB, AL, PT)} \quad \text{if } p = (\mathtt{b}_{prev}, \mathtt{b}, i) \text{ and } \models \langle a \rangle \Rightarrow (LV(t) = 1)$$

$$\frac{(p, LV, KB, AL, PT)}{((\mathtt{b}, \mathtt{b}_2, 0), LV, KB, AL, PT)} \quad \text{if } p = (\mathtt{b}_{prev}, \mathtt{b}, i) \text{ and } \models \langle a \rangle \Rightarrow (LV(t) = 0)$$

Next, we have to evaluate a `phi` instruction. These instructions are needed due to the static single assignment form of LLVM. Here, "`x = phi ty [`t_1`,`$\mathtt{b}_1$`]`, …, `[`t_n`,`$\mathtt{b}_n$`]`" means that if the previous block was $\mathtt{b}_j$, then the value t_j is assigned to `x`. All $t_1, \ldots, t_n$ must have type `ty`. Since we reached state F in Fig. 1 after evaluating the `entry` block, we obtain the state G with $\mathtt{olds} = v_3$ and $v_3 = u_{\mathtt{str}}$.

phi (p : "`x = phi ty [`t_1`,`$\mathtt{b}_1$`]`, …, `[`t_n`,`$\mathtt{b}_n$`]`" **with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_i \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$, $\mathtt{b}_i \in \mathit{Blks}$)

$$\frac{(p,\ LV \uplus \{\mathtt{x} = v\},\ KB,\ AL,\ PT)}{(p^+,\ LV \uplus \{\mathtt{x} = w\},\ KB \cup \{w = LV(t_j)\},\ AL,\ PT)} \quad \begin{array}{l}\text{if } p = (\mathtt{b}_j, \mathtt{b}, k) \text{ and}\\ w \in \mathcal{V}_{sym} \text{ is fresh}\end{array}$$

The `strlen` function traverses the string using a pointer `s` that is increased in each iteration. The `loop` terminates, since eventually `s` 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, "$\mathtt{ad}_2$ `= getelementptr ty*` $\mathtt{ad}_1$`, i`n t" increases $\mathtt{ad}_1$ by the size of t elements of type `ty` (i.e., by $\mathit{size}(\mathtt{ty}) \cdot t$) and assigns this address to $\mathtt{ad}_2$.[13]

getelementptr (p : "$\mathtt{ad}_2$ `= getelementptr ty*` $\mathtt{ad}_1$`, i`n t", $\mathtt{ad}_1, \mathtt{ad}_2 \in \mathcal{V}_\mathcal{P}$, $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV \uplus \{\mathtt{ad}_2 = v\},\ KB,\ AL,\ PT)}{(p^+, LV \uplus \{\mathtt{ad}_2 = w\},\ KB \cup \{w = LV(\mathtt{ad}_1) + \mathit{size}(\mathtt{ty}) \cdot LV(t)\},\ AL,\ PT)} \quad w \in \mathcal{V}_{sym} \text{ fresh}$$

In Fig. 1, this rule is used for the step from G to H, where LV and KB now imply $\mathtt{s} = \mathtt{str} + 1$. In the step to I, 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 $u_{\mathtt{str}} \leq u_{\mathtt{str}} + 1 \leq v_{end}$). This holds because $\langle H \rangle$ implies $u_{\mathtt{str}} \leq v_{end}$ and $u_{\mathtt{str}} \neq v_{end}$ (as $u_{\mathtt{str}} \hookrightarrow v_1, v_{end} \hookrightarrow 0 \in PT$ and $v_1 \neq 0 \in KB$). Finally, we check whether `c` is `0`. We again perform a refinement which yields the states J and K. State K corresponds to the case $\mathtt{c} \neq \mathtt{0}$ and thus, we obtain $\mathtt{czero} = \mathtt{0}$ in L and branch back to instruction 0 of the `loop` block in state M.

2.3 Generalizing Abstract States

After reaching M, one unfolds the loop once more until one reaches a state $\widetilde{M}$ at position $(\mathtt{loop}, \mathtt{loop}, 0)$ again, analogous to the first iteration. To obtain *finite* symbolic execution graphs, we *generalize* our states whenever an evaluation visits a program position twice. Thus, we have to find a state that is more general than $M = (p, LV_M, KB_M, AL, PT_M)$ and $\widetilde{M} = (p, LV_{\widetilde{M}}, KB_{\widetilde{M}}, AL, PT_{\widetilde{M}})$. For readability, we again write "$\hookrightarrow$" instead of "$\hookrightarrow_{\mathtt{i8}}$". Then $p = (\mathtt{loop}, \mathtt{loop}, 0)$ and

[13] Since we do not consider the handling of data structures in this paper, we do not regard `getelementptr` instructions with more than two parameters.

$$
\begin{aligned}
AL &= \{alloc(u_{\mathtt{str}}, v_{end})\}\\
LV_M &= \{\mathtt{str} = u_{\mathtt{str}}, \mathtt{c} = v_5, \mathtt{s} = v_4, \mathtt{olds} = v_3, \ldots\}\\
LV_{\widetilde{M}} &= \{\mathtt{str} = u_{\mathtt{str}}, \mathtt{c} = \widetilde{v_5}, \mathtt{s} = \widetilde{v_4}, \mathtt{olds} = \widetilde{v_3}, \ldots\}\\
PT_M &= \{u_{\mathtt{str}} \hookrightarrow v_1, v_4 \hookrightarrow v_5, v_{end} \hookrightarrow z\}\\
PT_{\widetilde{M}} &= \{u_{\mathtt{str}} \hookrightarrow v_1, v_4 \hookrightarrow v_5, \widetilde{v_4} \hookrightarrow \widetilde{v_5}, v_{end} \hookrightarrow z\}\\
KB_M &= \{v_5 \neq 0, v_4 = v_3 + 1, v_3 = u_{\mathtt{str}}, v_1 \neq 0, z = 0, \ldots\}\\
KB_{\widetilde{M}} &= \{\widetilde{v_5} \neq 0, \widetilde{v_4} = \widetilde{v_3} + 1, \widetilde{v_3} = v_4, v_4 = v_3 + 1, v_3 = u_{\mathtt{str}}, v_1 \neq 0, z = 0, \ldots\}.
\end{aligned}
$$

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

To merge M and $\widetilde{M}$, we keep those constraints of M that also hold in $\widetilde{M}$. To this end, we proceed in two steps. First, we create a new state $N = (p, LV_N, KB_N, AL_N, PT_N)$ using fresh symbolic variables $v_{\mathtt{x}}$ for all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ and define

$$LV_N = \{\mathtt{str} = v_{\mathtt{str}}, \mathtt{c} = v_{\mathtt{c}}, \mathtt{s} = v_{\mathtt{s}}, \mathtt{olds} = v_{\mathtt{olds}}, \ldots\}.$$

Matching N's fresh variables to the variables in M and $\widetilde{M}$ yields mappings with $\mu_M(v_{\mathtt{str}}) = u_{\mathtt{str}}$, $\mu_M(v_{\mathtt{c}}) = v_5$, $\mu_M(v_{\mathtt{s}}) = v_4$, $\mu_M(v_{\mathtt{olds}}) = v_3$, and $\mu_{\widetilde{M}}(v_{\mathtt{str}}) = u_{\mathtt{str}}$, $\mu_{\widetilde{M}}(v_{\mathtt{c}}) = \widetilde{v_5}$, $\mu_{\widetilde{M}}(v_{\mathtt{s}}) = \widetilde{v_4}$, $\mu_{\widetilde{M}}(v_{\mathtt{olds}}) = \widetilde{v_3}$. By injectivity of LV_M, we can also define a pseudo-inverse of μ_M that maps M's variables to N by setting $\mu_M^{-1}(LV_M(\mathtt{x})) = v_{\mathtt{x}}$ for $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ and $\mu_M^{-1}(v) = v$ for all other $v \in \mathcal{V}_{sym}$ ($\mu_{\widetilde{M}}^{-1}$ works analogously).

In a second step, we use these mappings to check which constraints of M also hold in $\widetilde{M}$. So we set $AL_N = \mu_M^{-1}(AL) \cap \mu_{\widetilde{M}}^{-1}(AL) = \{alloc(v_{\mathtt{str}}, v_{end})\}$ and

$$
\begin{aligned}
PT_N &= \mu_M^{-1}(PT_M) \cap \mu_{\widetilde{M}}^{-1}(PT_{\widetilde{M}})\\
&= \{v_{\mathtt{str}} \hookrightarrow v_1, v_{\mathtt{s}} \hookrightarrow v_{\mathtt{c}}, v_{end} \hookrightarrow z\} \cap \{v_{\mathtt{str}} \hookrightarrow v_1, v_4 \hookrightarrow v_5, v_{\mathtt{s}} \hookrightarrow v_{\mathtt{c}}, v_{end} \hookrightarrow z\}\\
&= \{v_{\mathtt{str}} \hookrightarrow v_1, v_{\mathtt{s}} \hookrightarrow v_{\mathtt{c}}, v_{end} \hookrightarrow z\}.
\end{aligned}
$$

It remains to construct KB_N. We have $v_3 = u_{\mathtt{str}}$ ("$\mathtt{olds} = \mathtt{str}$") in $\langle M\rangle$, but $\widetilde{v_3} = v_4$, $v_4 = v_3 + 1$, $v_3 = u_{\mathtt{str}}$ ("$\mathtt{olds} = \mathtt{str} + 1$") in $\langle\widetilde{M}\rangle$. To keep as much information as possible in such cases, we rewrite equations to inequations before performing the generalization. For this, let $\langle\!\langle M\rangle\!\rangle$ result from extending $\langle M\rangle$ by $t_1 \geq t_2$ and $t_1 \leq t_2$ for any equation $t_1 = t_2 \in \langle M\rangle$. So in our example, we obtain $v_3 \geq u_{\mathtt{str}} \in \langle\!\langle M\rangle\!\rangle$ ("$\mathtt{olds} \geq \mathtt{str}$"). Moreover, for any $t_1 \neq t_2 \in \langle M\rangle$, we check whether $\langle M\rangle$ implies $t_1 > t_2$ or $t_1 < t_2$, and add the respective inequation to $\langle\!\langle M\rangle\!\rangle$. 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 $v_4 < v_{end} \in \langle\!\langle M\rangle\!\rangle$ ("$\mathtt{s} < v_{end}$") from $v_4 = v_3 + 1$, $v_3 = u_{\mathtt{str}}$, $u_{\mathtt{str}} \leq v_{end}$, $u_{\mathtt{str}} \neq v_{end}$, $v_4 \neq v_{end} \in \langle M\rangle$.

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

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

Definition 3 (Merging States). *Let $a = (p, LV_a, KB_a, AL_a, PT_a)$ and $b = (p, LV_b, KB_b, AL_b, PT_b)$ be abstract states. Then $c = (p, LV_c, KB_c, AL_c, PT_c)$ results from* merging *the states a and b if*

- $LV_c = \{\mathtt{x} = v_{\mathtt{x}} \mid \mathtt{x} \in \mathcal{V}_{\mathcal{P}}\}$ *for fresh pairwise different symbolic variables $v_{\mathtt{x}}$. Moreover, we define $\mu_a(v_{\mathtt{x}}) = LV_a(\mathtt{x})$ and $\mu_b(v_{\mathtt{x}}) = LV_b(\mathtt{x})$ for all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ and let μ_a and μ_b be the identity on all remaining variables from $\mathcal{V}_{sym}$.*
- $AL_c = \mu_a^{-1}(AL_a) \cap \mu_b^{-1}(AL_b)$ *and* $PT_c = \mu_a^{-1}(PT_a) \cap \mu_b^{-1}(PT_b)$. *Here, the "inverse" of the instantiation μ_a is defined as $\mu_a^{-1}(v) = v_{\mathtt{x}}$ if $v = LV_a(\mathtt{x})$ 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*

$$\begin{aligned}\langle\!\langle a \rangle\!\rangle \ = \ \langle a \rangle & \cup \{\ t_1 \geq t_2,\ t_1 \leq t_2 \ \mid\ t_1 = t_2 \in \langle a \rangle\ \} \\ & \cup \{\ t_1 > t_2 \ \mid\ t_1 \neq t_2 \in \langle a \rangle,\ \models \langle a \rangle \Rightarrow t_1 > t_2\ \} \\ & \cup \{\ t_1 < t_2 \ \mid\ t_1 \neq t_2 \in \langle a \rangle,\ \models \langle a \rangle \Rightarrow t_1 < t_2\ \}.\end{aligned}$$

In Fig. 1, we do not show the second loop unfolding from M to $\widetilde{M}$, and directly draw a *generalization edge* from M to N, depicted by a dashed arrow. Such an edge expresses that all concrete states represented by M are also represented by the more general state N. Semantically, a state $\overline{a}$ is a generalization of a state a iff $\models \langle a \rangle_{SL} \Rightarrow \mu(\langle \overline{a} \rangle_{SL})$ for some instantiation μ. To automate our procedure, we define a weaker relationship between a and $\overline{a}$. We say that $\overline{a} = (p, \overline{LV}, \overline{KB}, \overline{AL}, \overline{PT})$ is a *generalization* of $a = (p, LV, KB, AL, PT)$ with the instantiation μ whenever the conditions (b)-(e) of the following rule are satisfied.

generalization with μ $\qquad \dfrac{(p, LV, KB, AL, PT)}{(p, \overline{LV}, \overline{KB}, \overline{AL}, \overline{PT})}$ if

(a) a has an incoming evaluation edge,[14]
(b) $LV(\mathtt{x}) = \mu(\overline{LV}(\mathtt{x}))$ for all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$,
(c) $\models \langle a \rangle \Rightarrow \mu(\overline{KB})$,
(d) if $alloc(v_1, v_2) \in \overline{AL}$, then $alloc(\mu(v_1), \mu(v_2)) \in AL$,
(e) if $(v_1 \hookrightarrow_{\mathtt{ty}} v_2) \in \overline{PT}$, then $(\mu(v_1) \hookrightarrow_{\mathtt{ty}} \mu(v_2)) \in PT$

Clearly, then we indeed have $\models \langle a \rangle_{SL} \Rightarrow \mu(\langle \overline{a} \rangle_{SL})$. 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 (or "widenings"). Thm. 4 shows that the merging heuristic from Def. 3 satisfies the conditions of the generalization rule. Thus, since N results from merging M and $\widetilde{M}$, it is indeed a *generalization* of M. Thm. 4 also shows that if one uses the merging heuristic to compute generalizations, then the construction of symbolic execution graphs always terminates when applying the following strategy:

- If there is a path from a state a to a state b, where a and b are at the same program position, where b has an incoming evaluation edge, and where a has no incoming refinement edge, then we check whether a is a generalization of

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

b (i.e., whether the corresponding conditions of the generalization rule are satisfied). In that case, we draw a generalization edge from b to a.

- 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.

Theorem 4 (Soundness and Termination of Merging). *Let c result from merging the states a and b as in Def. 3. 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, then $|\langle\!\langle c \rangle\!\rangle| + |AL_c| + |PT_c| < |\langle\!\langle a \rangle\!\rangle| + |AL_a| + |PT_a|$. Here, for any conjunction φ, let $|\varphi|$ denote the number of its conjuncts. Thus, the above strategy to construct symbolic execution graphs always terminates.*[15]

In our example, we continue symbolic execution in state N. Similar to the execution from F to M, after 6 steps another state P at position $(\mathtt{loop}, \mathtt{loop}, 0)$ is reached. In Fig. 1, dotted arrows abbreviate several evaluation steps. As N is again a generalization of P 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 P to N. The construction of the symbolic execution graph is finished as soon as all its leaves correspond to `ret` instructions (for "return").

Based on this construction, we now connect the symbolic execution graph to memory safety of the input program. We say that a concrete LLVM state (p, s, m) is *represented* by the symbolic execution graph iff the graph contains an abstract state a at position p where $(s, m) \models \sigma(\langle a \rangle_{SL})$ for some concrete instantiation σ.

Theorem 5 (Memory Safety of LLVM Programs). *Let $\mathcal{P}$ be an LLVM program with a symbolic execution graph $\mathcal{G}$. If $\mathcal{G}$ does not contain the abstract state ERR, then $\mathcal{P}$ is memory safe for all LLVM states represented by $\mathcal{G}$.*

3 From Symbolic Execution Graphs to Integer Systems

To prove termination of the input program, we extract an *integer transition system* (ITS) from the symbolic execution graph and then 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 the conditions in the abstract states only concern the symbolic variables and therefore, these are usually the essential variables for proving termination.

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

Formally, *ITSs* are graphs whose nodes are abstract states and whose edges are *transitions*. For any abstract state a, let $\mathcal{V}(a)$ denote the symbolic variables occurring in a. Let $\mathcal{V} \subseteq \mathcal{V}_{sym}$ be the finite set of all symbolic variables occurring

[15] The proofs for all theorems can be found in App. B.

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 the variables $\mathcal{V} \uplus \mathcal{V}'$. Here, $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ represents the values of the variables *after* the transition. An *ITS state* (a, σ) consists of an abstract state a and a concrete instantiation $\sigma : \mathcal{V} \to \mathbb{Z}$. For any such σ, let $\sigma' : \mathcal{V}' \to \mathbb{Z}$ with $\sigma'(v') = \sigma(v)$. Given an ITS $\mathcal{I}$, (a, σ) *evaluates* to $(\overline{a}, \overline{\sigma})$ (denoted "$(a, \sigma) \to_{\mathcal{I}} (\overline{a}, \overline{\sigma})$") iff $\mathcal{I}$ has a transition $(a, CON, \overline{a})$ with $\models (\sigma \cup \overline{\sigma}')\,(CON)$. Here, 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.[16]

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 $v \in \mathcal{V}(\overline{a})$ in $\overline{a}$ is $\mu(v)$. Hence, we create the transition $(a,\ \langle a \rangle \cup \{v' = \mu(v) \mid v \in \mathcal{V}(\overline{a})\},\ \overline{a})$.[17] So for the edge from P to N 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 $\mathcal{V}(a)$. Thus, we construct the transition $(a,\ \langle a \rangle \cup \{v' = v \mid v \in \mathcal{V}(a)\},\ \overline{a})$.

So in the ITS resulting from Fig. 1, the condition of the transition from A to B contains $\{v'_{end} = v_{end}\} \cup \{u'_{\mathtt{x}} = u_{\mathtt{x}} \mid \mathtt{x} \in \mathcal{V}_{\mathcal{P}}\}$. 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 $v_1 \notin \mathcal{V}(A)$), but in the transition from B to D, the value of v_1 must remain the same.

Definition 6 (ITS from Symbolic Execution Graph). *Let $\mathcal{G}$ be a symbolic execution graph. Then the* corresponding integer transition system $\mathcal{I}_{\mathcal{G}}$ *has one transition for each edge in $\mathcal{G}$:*

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

From the non-generalization edges on the path from N to P 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 P to N and it remains the same in all other transitions of the graph's only cycle. Since the transition from P to N 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.

The following theorem shows the soundness of our approach.

Theorem 7 (Termination of LLVM Programs). *Let $\mathcal{P}$ be an* LLVM *program with a symbolic execution graph $\mathcal{G}$ that does not contain the state ERR. If $\mathcal{I}_{\mathcal{G}}$ is terminating, then $\mathcal{P}$ is also terminating for all* LLVM *states represented by $\mathcal{G}$.*

[16] 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).

[17] 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 $\overline{a}$.

4 Related Work, Experiments, and Conclusion

We 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 standard SMT solving). 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.

Previous methods and tools for termination analysis of imperative programs (e.g., AProVE [4, 5], ARMC [24], COSTA [1], Cyclist [7], FuncTion [28], Julia [25], KITTeL [12], LoopFrog [27], TAN [16], TRex [14], T2 [6], Ultimate [15], ...) either do not handle the heap at all, or support dynamic data structures by an abstraction to integers (e.g., to represent sizes or lengths) or to terms (representing finite unravelings). However, most tools fail when the control flow depends on explicit pointer arithmetic and on detailed information about the contents of addresses. While the general methodology of our approach was inspired by our previous work on termination of Java [4, 5], in the current paper we lift such techniques to prove termination and memory safety of programs with explicit pointer arithmetic. This requires a fundamentally new approach, since pointer arithmetic and memory allocation cannot be expressed in the Java-based techniques of [4, 5].

We implemented our technique in the termination prover AProVE using the SMT solvers Yices [11] and Z3 [20] in the back-end. A preliminary version of our implementation participated very successfully in the *International Competition on Software Verification (SV-COMP)* [26] at *TACAS*, which featured a category for termination of C programs for the first time in 2014. To evaluate AProVE's power, we performed experiments on a collection of 208 C programs from several sources, including the *SV-COMP 2014* termination category and standard string algorithms from [29] and the OpenBSD C library [23]. Of these 208 programs, 129 use pointers and 79 only operate on integers.

To prove termination of low-level C programs, one also has to ensure their memory safety. 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 [19] or SLAyer [3]) or require the user to provide the needed loop invariants (as in the Jessie plug-in of Frama-C [21]). In contrast, our approach can prove memory safety of such algorithms fully automatically. Although our approach is targeted toward termination and only analyzes memory safety as a prerequisite for termination, it turned out that on our collection, AProVE is more powerful than the leading publicly available tools for proving memory safety. To this end, we compared AProVE with the tools CPAchecker [18] and Predator [10] which reached the first and the third place in the category for *memory safety* at *SV-COMP 2014*.[18] For the 129 pointer programs in our collection, AProVE can show memory safety for 102 examples, whereas CPAchecker resp. Predator prove

[18] The second place in this category was reached by the bounded model checker LLBMC [13]. However, in general such tools only disprove, but cannot verify memory safety.

memory safety for 77 resp. 79 examples (see [2] for details).

To evaluate the power of our approach for proving termination, we compared AProVE to the other tools from the termination category of *SV-COMP 2014*. In addition, we included the termination analyzer KITTeL [12] in our evaluation, which operates on LLVM as well. On the side, we show the performance of the tools on integer and pointer programs when using a time limit of 300 seconds for each example. Here, we used an Intel Core i7-950 processor and 6 GB of memory. "**T**" gives the number of examples where termination could be proved, "**N**" is the number of examples where non-termination could be shown, "**F**" states how often the tool failed in less than 300 seconds, "**TO**" gives the number of time-outs (i.e., examples for which the tool took longer than 300 seconds), and "**RT**" is the average run time in seconds for those examples where the tool proved termination or non-termination. For pointer programs, we omitted the results for those tools that were not able to prove termination of any examples.

	79 integer programs					129 pointer programs				
	T	**N**	**F**	**TO**	**RT**	**T**	**N**	**F**	**TO**	**RT**
AProVE	67	0	11	1	19.6	91	0	19	19	58.6
FuncTion	11	0	66	2	23.1	-	-	-	-	-
KITTeL	58	0	12	9	0.2	9	0	1	119	0.2
T2	55	0	23	1	1.8	6	0	123	0	3.6
TAN	31	0	37	11	2.4	3	0	124	2	10.6
Ultimate	57	4	12	6	3.2	-	-	-	-	-

Most other termination provers ignore the problem of memory safety and just prove termination under the *assumption* that the program is memory safe. So they may also return "Yes" for memory unsafe programs and may treat read accesses to the heap as non-deterministic input. Since AProVE constructs symbolic execution graphs to prove memory safety and to infer suitable invariants needed for termination proofs, its runtime is often higher than that of other tools. On the other hand, the table shows that our approach is slightly more powerful than the other tools for integer programs (i.e., our graph-based technique is also suitable for programs on integers) and it is clearly the most powerful one for pointer programs. The reason is due to our novel representation of the memory which handles pointer arithmetic and keeps information about the contents of addresses. For details on our experiments and to access our implementation in AProVE via a web interface, we refer to [2]. In future work, we plan to extend our approach to recursive programs and to inductive data structures defined via `struct` (e.g., by integrating existing shape analyses based on separation logic).

Acknowledgments. We are grateful to the developers of the other tools for termination or memory safety [6, 10, 12, 15, 16, 18, 28] for their help with the experiments.

References

1. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination analysis of Java Bytecode. In: Proc. FMOODS '08
2. AProVE: `http://aprove.informatik.rwth-aachen.de/eval/Pointer/`
3. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: Proc. CAV '11

4. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for JBC. In: Proc. FoVeOOS '11
5. Brockschmidt, M., Musiol, R., Otto, C., Giesl, J.: Automated termination proofs for Java programs with cyclic data. In: Proc. CAV '12
6. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: Proc. CAV '13
7. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Proc. APLAS '12
8. Clang compiler: http://clang.llvm.org
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77
10. Dudka, K., Müller, P., Peringer, P., Vojnar, T.: Predator: A shape analyzer based on symbolic memory graphs (competition contribution). In: Proc. TACAS '14
11. Dutertre, B., de Moura, L.M.: The Yices SMT solver (2006), tool paper at http://yices.csl.sri.com/tool-paper.pdf
12. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Proc. RTA '11
13. Falke, S., Merz, F., Sinz, C.: LLBMC: Improved bounded model checking of C using LLVM (competition contribution). In: Proc. TACAS '13
14. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for termination. In: Proc. SAS '10
15. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Proc. ATVA '13
16. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.: Termination analysis with compositional transition invariants. In: Proc. CAV '10
17. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO '04
18. Löwe, S., Mandrykin, M., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses (comp. contr.). In: Proc. TACAS '14
19. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: Proc. POPL '10
20. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS '08
21. Moy, Y., Marché, C.: Modular inference of subprogram contracts for safety checking. J. Symb. Comput. 45(11), 1184–1211 (2010)
22. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Proc. CSL '01
23. http://fxr.watson.org/fxr/source/lib/libsa/strlen.c?v=OPENBSD
24. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Proc. PADL '07
25. Spoto, F., Mesnard, F., Payet, É.: A termination analyser for Java Bytecode based on path-length. ACM TOPLAS 32(3) (2010)
26. *SV-COMP* at *TACAS 2014*: http://sv-comp.sosy-lab.org/2014/
27. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop summarization and termination analysis. In: Proc. TACAS '11
28. Urban, C.: The abstract domain of segmented ranking functions. In: Proc. SAS '13
29. Wikibooks C Programming: http://en.wikibooks.org/wiki/C_Programming/
30. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formalizing the LLVM IR for verified program transformations. In: Proc. POPL '12

A Further Symbolic Execution Rules

In the following, we present symbolic execution rules for additional LLVM instructions (in particular, for all instructions occurring in our `strlen` example). Our implementation contains rules for several further LLVM instructions that can be formulated in a similar way.

A.1 `store` instruction

The rule for `store` is analogous to the rule for `load`. The instruction "`store ty` t`, ty* ad`" stores the value t of type `ty` at the address `ad`. Again, we check whether $LV(\texttt{ad}), \ldots, LV(\texttt{ad}) + \mathit{size}(\texttt{ty}) - 1$ are addresses in an allocated part of the memory. Of course, the information that `ad` now points to t should be added to the set PT. All other information in PT that is not influenced by this change can be kept.[19]

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

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV,\ KB \cup \{v = LV(t)\},\ AL,\ \overline{PT} \cup \{LV(\texttt{ad}) \hookrightarrow_{\texttt{ty}} v\})} \quad \text{if}$$

- there is $alloc(v_1, v_2) \in AL$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV(\texttt{ad}) \wedge LV(\texttt{ad}) + \mathit{size}(\texttt{ty}) - 1 \leq v_2)$,
- $\overline{PT} = \{(w_1 \hookrightarrow_{\texttt{sy}} w_2) \in PT \mid \models \langle a \rangle \Rightarrow (\,[\,LV(\texttt{ad}),\ LV(\texttt{ad}) + \mathit{size}(\texttt{ty}) - 1\,] \perp [\,w_1,\ w_1 + \mathit{size}(\texttt{sy}) - 1]\,)\}$,
- $v \in \mathcal{V}_{sym}$ is fresh

In Sect. 2.2, we also presented a `load`-rule for the case where the addresses $LV(\texttt{ad}), \ldots, LV(\texttt{ad}) + \mathit{size}(\texttt{ty}) - 1$ are not allocated. An analogous rule is used for violations of memory safety caused by the instruction "`store ty` t`, ty* ad`".

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

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

there is no $alloc(v_1, v_2) \in AL$ with $\models \langle a \rangle \Rightarrow (v_1 \leq LV(\texttt{ad}) \wedge LV(\texttt{ad}) + \mathit{size}(\texttt{ty}) - 1 \leq v_2)$

A.2 `alloca` instruction

Now we present 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., AL is extended by $alloc(v_1, v_2)$ for fresh symbolic variables v_1, v_2) and KB is extended by $v_2 = v_1 + \mathit{size}(\texttt{ty}) \cdot LV(t)$. Moreover, the address of the first memory cell in the newly allocated block is assigned to `x`. Thus, we update LV by $\texttt{x} = v_1$.

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

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

$$\frac{(p,\ LV \uplus \{\mathtt{x} = v\},\ KB,\ AL,\ PT)}{(p^+,\ LV \uplus \{\mathtt{x} = v_1\},\ KB \cup \{v_2 = v_1 + size(\mathtt{ty}) \cdot LV(t) - 1\},\ AL \cup \{alloc(v_1, v_2)\},\ PT)}$$

if $\models \langle a \rangle \Rightarrow (LV(t) > 0)$ and $v_1, v_2 \in \mathcal{V}_{sym}$ are fresh

To ease the presentation, in this paper we only consider programs which allocate memory on the stack (using `alloca`), but we do not consider allocation or deallocation on the heap (using `malloc` and `free`). The reason is that the latest releases of LLVM do not have built-in `malloc` or `free` instructions anymore, but one would have to call them as external functions (provided by the standard C library). However, we restricted ourselves to the analysis of a single LLVM function.

In fact, LLVM does not explicitly distinguish between the heap and stack, but applies the same memory model for both (using `load` and `store`). The only difference is that memory acquired by `alloca` is automatically deallocated at the end of the function. So we need not consider the additional book-keeping required to keep track of `malloc` and `free`, which would be orthogonal to our contribution.

A.3 `br` instruction (unconditional)

The instruction "`br label` $\mathtt{b}_{next}$" means that the execution has to continue with the first instruction in the block $\mathtt{b}_{next}$. It is similar to the conditional branch instruction in Sect. 2.2.

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

$$\frac{(p, LV, KB, AL, PT)}{((\mathtt{b}, \mathtt{b}_{next}, 0), LV, KB, AL, PT)} \quad \text{if } p = (\mathtt{b}_{prev}, \mathtt{b}, i)$$

A.4 Refining abstract states for the conditional `br` instruction

Similar to the refinement rule in Sect. 2.2, an analogous refinement rule is also used for conditional branching instructions, if the current knowledge base does not contain enough information to decide whether the condition of the `br` instruction is *true* or *false*.

refining abstract states (p : "`br i1` t, `label` $\mathtt{b}_1$, `label` $\mathtt{b}_2$")

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p,\ LV,\ KB \cup \{LV(t) = 1\},\ AL,\ PT)\ \mid\ (p,\ LV,\ KB \cup \{LV(t) \neq 1\},\ AL,\ PT)}$$

if $\not\models \langle a \rangle \Rightarrow (LV(t) = 1)$ and $\not\models \langle a \rangle \Rightarrow (LV(t) \neq 1)$

A.5 `sub` instruction

The next rule handles statements of the form "`x = sub ty` t_1`,` t_2". Here, both t_1 and t_2 must have the type `ty` and the variable `x` also gets this type. Of course, in a similar way one can also handle other instructions for arithmetic operations in LLVM, e.g., `add`.

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

$$\frac{(p,\ LV \uplus \{\mathtt{x} = v\},\ KB,\ AL,\ PT)}{(p^+, LV \uplus \{\mathtt{x} = w\},\ KB \cup \{w = LV(t_1) - LV(t_2)\},\ AL,\ PT)} \quad \text{with } w \in \mathcal{V}_{sym} \text{ fresh}$$

A.6 `ptrtoint` instruction

Finally, we present a rule for the `ptrtoint` instruction. It 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).

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

$$\frac{(p,\ LV \uplus \{\mathtt{x} = v\},\ KB,\ AL,\ PT)}{(p^+, LV \uplus \{\mathtt{x} = w\},\ KB \cup \{w = LV(\mathtt{ad})\},\ AL,\ PT)} \quad \text{with } w \in \mathcal{V}_{sym} \text{ fresh}$$

B Proofs

Theorem 4 (Soundness and Termination of Merging). *Let c result from merging the states a and b as in Def. 3. 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, then $|\langle\!\langle c\rangle\!\rangle|+|AL_c|+|PT_c| < |\langle\!\langle a\rangle\!\rangle|+|AL_a|+|PT_a|$. Here, for any conjunction φ, let $|\varphi|$ denote the number of its conjuncts. Thus, the strategy in Sect. 2.3 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, respectively, we have to prove that the conditions (b)-(e) of the generalization rule in Sect. 2.3 are satisfied. By definition, we have $LV_a(\mathtt{x}) = \mu_a(v_{\mathtt{x}}) = \mu_a(LV_c(\mathtt{x}))$ and $LV_b(\mathtt{x}) = \mu_b(LV_c(\mathtt{x}))$ for all $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$, which proves (b). Moreover, for $\mathit{alloc}(v_1, v_2) \in AL_c$, we have $\mathit{alloc}(v_1, v_2) \in \mu_a^{-1}(AL_a)$ and $\mathit{alloc}(v_1, v_2) \in \mu_b^{-1}(AL_b)$. This implies $\mathit{alloc}(\mu_a(v_1), \mu_a(v_2)) \in AL_a$ and $\mathit{alloc}(\mu_b(v_1), \mu_b(v_2)) \in AL_b$, which proves (d). Condition (e) 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 have $\models \langle b\rangle \Rightarrow \mu_b(KB_c)$. 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 have to show that $|\langle\!\langle c\rangle\!\rangle| + |AL_c| + |PT_c| < |\langle\!\langle a\rangle\!\rangle| + |AL_a| + |PT_a|$ 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 we have $t_1 = t_2 \in \mu_a^{-1}(\langle\!\langle a\rangle\!\rangle)$ and thus also $t_1 \geq t_2, t_1 \leq t_2 \in \mu_a^{-1}(\langle\!\langle a\rangle\!\rangle)$. As $\models \mu_b^{-1}(\langle b\rangle) \Rightarrow t_1 = t_2$ also implies $\models \mu_b^{-1}(\langle b\rangle) \Rightarrow t_1 \geq t_2$ and $\models \mu_b^{-1}(\langle b\rangle) \Rightarrow t_1 \leq t_2$, we also have $t_1 \geq t_2, t_1 \leq t_2 \in \langle c\rangle$. Moreover, suppose that $t_1 \neq t_2 \in \langle c\rangle$ 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 have $t_1 > t_2 \in \langle c\rangle$. The case where $t_1 \neq t_2 \in \langle c\rangle$ and $\models \langle c\rangle \Rightarrow t_1 < t_2$ is analogous.

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 $|KB_c|+|AL_c|+|PT_c| < |\langle\!\langle a\rangle\!\rangle|+|AL_a|+|PT_a|$. From the definition, it is obvious that we always have $|KB_c| \leq |\langle\!\langle a\rangle\!\rangle|$, $|AL_c| \leq |AL_a|$, and $|PT_c| \leq |PT_a|$.

Hence, it suffices to show that if $|KB_c| = |\langle\!\langle a\rangle\!\rangle|$, $|AL_c| = |AL_a|$, 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 $LV_b(\mathtt{x}) = \mu_b(v_{\mathtt{x}}) = \mu_b(\mu_a^{-1}(LV_a(\mathtt{x})))$, 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 $\mathit{alloc}(v_1, v_2) \in AL_a$, then $\mathit{alloc}(\mu_a^{-1}(v_1), \mu_a^{-1}(v_2)) \in \mu_a^{-1}(AL_a) = \mu_b^{-1}(AL_b)$ and hence, $\mathit{alloc}(\mu_b(\mu_a^{-1}(v_1)), \mu_b(\mu_a^{-1}(v_2))) \in AL_b$, which shows condition (d). Condition (e) follows from $|PT_c| = |PT_a|$ for a similar reason. 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). $\square$

Theorem 5 (Memory Safety of LLVM Programs). *Let $\mathcal{P}$ be an* LLVM *program with a symbolic execution graph $\mathcal{G}$. If $\mathcal{G}$ does not contain the abstract state ERR, then $\mathcal{P}$ is memory safe for all* LLVM *states represented by $\mathcal{G}$.*

Proof. Let $(p, s, m) \to_{\mathsf{LLVM}} (\overline{p}, \overline{s}, \overline{m})$, where (p, s, m) is represented by the symbolic execution graph $\mathcal{G}$. So $\mathcal{G}$ contains an abstract state a with $(s, m) \models \sigma(\langle a \rangle_{SL})$ for some concrete instantiation σ. As the definition of the symbolic execution graph directly follows the operational semantics of LLVM [17, 30], we immediately obtain the following:

(a) If a's outgoing edge is an evaluation edge, then for a's successor $\overline{a}$, we have $(\overline{s}, \overline{m}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$ for a concrete instantiation $\overline{\sigma}$ with $\overline{\sigma}(v) = \sigma(v)$ for all $v \in \mathcal{V}(a)$. Here, let $\mathcal{V}(a)$ denote all symbolic variables occurring in a.
(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 $(\overline{s}, \overline{m}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$ for a concrete instantiation $\overline{\sigma}$ with $\overline{\sigma}(v) = \sigma(v)$ for all $v \in \mathcal{V}(a)$.
(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 $(\overline{s}, \overline{m}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$ for a concrete instantiation $\overline{\sigma}$ with $\overline{\sigma}(v) = \sigma(\mu(v))$ for all $v \in \mathcal{V}(\widetilde{a})$.
(d) Otherwise, a's outgoing edge is a generalization edge to a state $\widetilde{a}$ with some instantiation μ, $\widetilde{a}$ has a refinement edge to a successor $\widehat{a}$, and there is an evaluation edge from $\widehat{a}$ to another abstract state $\overline{a}$, where $(\overline{s}, \overline{m}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$ for a concrete instantiation $\overline{\sigma}$ with $\overline{\sigma}(v) = \sigma(\mu(v))$ for all $v \in \mathcal{V}(\widetilde{a})$.

Thus, in all cases, $(\overline{p}, \overline{s}, \overline{m})$ is also represented by $\mathcal{G}$.

Similarly, if (p, s, m) is represented by the abstract state a in the graph $\mathcal{G}$, then $(p, s, m) \to_{\mathsf{LLVM}} \mathit{ERR}$ implies that there is an edge from a to ERR in $\mathcal{G}$. Thus, if the graph $\mathcal{G}$ does not contain ERR, then $\mathcal{P}$ is memory safe for all states represented by $\mathcal{G}$. $\square$

Theorem 7 (Termination of LLVM Programs). *Let $\mathcal{P}$ be an* LLVM *program with a symbolic execution graph $\mathcal{G}$ that does not contain the state ERR. If $\mathcal{I}_{\mathcal{G}}$ is terminating, then $\mathcal{P}$ is also terminating for all* LLVM *states represented by $\mathcal{G}$.*

Proof. Let $(p, s, m) \to_{\mathsf{LLVM}} (\overline{p}, \overline{s}, \overline{m})$, where (p, s, m) is represented by the symbolic execution graph $\mathcal{G}$. We show that if $\mathcal{G}$ does not contain ERR, then termination of the ITS $\mathcal{I}_{\mathcal{G}}$ implies termination of $\mathcal{P}$ for all states represented by $\mathcal{G}$. To this end, let $(p, s, m) \to_{\mathsf{LLVM}} (\overline{p}, \overline{s}, \overline{m})$, where $\mathcal{G}$ contains an abstract state a with $(s, m) \models \sigma(\langle a \rangle_{SL})$ for some concrete instantiation σ. In the proof of Thm. 5, we showed that there is an abstract state $\overline{a}$ in $\mathcal{G}$ and a concrete instantiation $\overline{\sigma}$ with $(\overline{s}, \overline{m}) \models \overline{\sigma}(\langle \overline{a} \rangle_{SL})$. In the following, we show that we also have $(a, \sigma) \to^{+}_{\mathcal{I}_{\mathcal{G}}} (\overline{a}, \overline{\sigma})$. This suffices to prove Thm. 7. The reason is that if there is an infinite $\to_{\mathsf{LLVM}}$-evaluation starting with an LLVM state represented by $\mathcal{G}$, then there is also a corresponding infinite evaluation with the ITS $\mathcal{I}_{\mathcal{G}}$. Hence, termination of $\mathcal{I}_{\mathcal{G}}$ implies termination of the LLVM program $\mathcal{P}$ for all states represented by $\mathcal{G}$.

(a) In case (a), the abstract state a has an evaluation edge to $\overline{a}$, where $\overline{\sigma}(v) = \sigma(v)$ for all $v \in \mathcal{V}(a)$. We show that then we have $(a, \sigma) \to_{\mathcal{I}_\mathcal{G}} (\overline{a}, \overline{\sigma})$.
Note that $\mathcal{I}_\mathcal{G}$ has a transition $(a, \langle a \rangle \cup \{v' = v \mid v \in \mathcal{V}(a)\}, \overline{a})$. Hence, we have to show that $(\sigma \cup \overline{\sigma}')$ satisfies the condition of this transition. We have[20]

$$\begin{array}{rll} & \models (\sigma \cup \overline{\sigma}')\,(\langle a \rangle), & \text{since} \\ & \models \sigma(\langle a \rangle), & \text{since} \\ (s, m) & \models \sigma(\langle a \rangle), & \text{since} \\ (s, m) & \models \sigma(\langle a \rangle_{SL}). & \end{array}$$

Moreover, for all $v \in \mathcal{V}(a)$, we have

$$(\sigma \cup \overline{\sigma}')(v') \;=\; \overline{\sigma}'(v') \;=\; \overline{\sigma}(v) \;=\; \sigma(v) \;=\; (\sigma \cup \overline{\sigma}')(v).$$

(b) In case (b), there is a path consisting of a refinement and a subsequent evaluation edge from a to $\overline{a}$ and $\overline{\sigma}(v) = \sigma(v)$ for all $v \in \mathcal{V}(a)$. 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, $(s, m) \models \sigma(\langle \widetilde{a} \rangle_{SL})$. Hence, $(\widetilde{a}, \sigma) \to_{\mathcal{I}_\mathcal{G}} (\overline{a}, \overline{\sigma})$ can be shown as in (a).
It remains to show $(a, \sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a}, \sigma)$. Clearly, $\mathcal{I}_\mathcal{G}$ has a transition $(a, \langle a \rangle \cup \{v' = v \mid v \in \mathcal{V}(a)\}, \widetilde{a})$. As in (a), one can show that $(\sigma \cup \sigma')$ satisfies the condition of this transition.

(c) In case (c), a has an generalization edge to $\widetilde{a}$ with the instantiation μ and an evaluation edge from $\widetilde{a}$ to $\overline{a}$, where $\overline{\sigma}(v) = \sigma(\mu(v))$ for all $v \in \mathcal{V}(\widetilde{a})$. We show that then we have $(a, \sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a}, \sigma \circ \mu) \to_{\mathcal{I}_\mathcal{G}} (\overline{a}, \overline{\sigma})$.
To see this, we start with proving $(a, \sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a}, \sigma \circ \mu)$. Due to the generalization edge from a to $\widetilde{a}$, $\mathcal{I}_\mathcal{G}$ has the transition $(a, \langle a \rangle \cup \{v' = \mu(v) \mid v \in \mathcal{V}(\widetilde{a})\}, \widetilde{a})$. So we have to show that $(\sigma \cup (\sigma \circ \mu)')$ satisfies the condition of this transition. We have

$$\begin{array}{rll} & \models (\sigma \cup (\sigma \circ \mu)')\,(\langle a \rangle), & \text{since} \\ & \models \sigma(\langle a \rangle), & \text{since} \\ (s, m) & \models \sigma(\langle a \rangle), & \text{since} \\ (s, m) & \models \sigma(\langle a \rangle_{SL}). & \end{array}$$

Moreover, for all $v \in \mathcal{V}(\widetilde{a})$, we have

$$(\sigma \cup (\sigma \circ \mu)')(v') \;=\; (\sigma \circ \mu)'(v') \;=\; \sigma(\mu(v)) \;=\; (\sigma \cup (\sigma \circ \mu)')(\mu(v)).$$

Now we have to 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, $(s, m) \models \sigma(\langle a \rangle_{SL})$ implies $(s, m) \models (\sigma \circ \mu)(\langle \widetilde{a} \rangle_{SL})$. Hence, $(\widetilde{a}, \sigma \circ \mu) \to_{\mathcal{I}_\mathcal{G}} (\overline{a}, \overline{\sigma})$ follows as in (a).

[20] Note that since σ is a concrete instantiation (i.e., $\sigma(\langle a \rangle)$ does not contain any variables), $(s, m) \models \sigma(\langle a \rangle)$ implies $\models \sigma(\langle a \rangle)$.

(d) Finally, we consider the case where a has 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 $v \in \mathcal{V}(\widetilde{a})$. We show that then we have $(a, \sigma) \to_{\mathcal{I}_\mathcal{G}} (\widetilde{a}, \sigma \circ \mu) \to^+_{\mathcal{I}_\mathcal{G}} (\overline{a}, \overline{\sigma})$.

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). $\square$

Aachener Informatik-Berichte

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

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

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

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

2012-01	Fachgruppe Informatik: Annual Report 2012
2012-02	Thomas Heer: Controlling Development Processes
2012-03	Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04	Marcus Gelderie: Strategy Machines and their Complexity
2012-05	Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06	Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data
2012-07	André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms
2012-08	Hongfei Fu: Computing Game Metrics on Markov Decision Processes
2012-09	Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains
2012-10	Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations
2012-12	Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs
2012-15	Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations
2012-16	Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets
2012-17	Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods
2013-01 *	Fachgruppe Informatik: Annual Report 2013
2013-02	Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen
2013-03	Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM
2013-04	Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries
2013-05	Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung

2013-08 Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers

2013-10 Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata

2013-12 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs

2013-13 Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

2013-14 Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures

2013-16 Carsten Otto: Java Program Analysis by Symbolic Execution

2013-19 Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol

2013-20 Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models

2014-01 * Fachgruppe Informatik: Annual Report 2014

2014-02 Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software

2014-03 Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide

2014-04 Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata

2014-05 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic

2014-06 Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video

2014-07 Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations

2014-08 Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs

2014-09 Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014

2014-14 Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks

2014-15 Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code

2014-16 Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results

2015-01 * Fachgruppe Informatik: Annual Report 2015

2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications

2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity

2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon

2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization

2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus

* These reports are only available as a printed version.

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