Proving Termination of C Programs with Lists[*]

Jera Hensel and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract. There are many techniques and tools to prove termination of C programs, but up to now these tools were not very powerful for fully automated termination proofs of programs whose termination depends on recursive data structures like lists. We present the first approach that extends powerful techniques for termination analysis of C programs (with memory allocation and explicit pointer arithmetic) to lists.

1 Introduction

In [11,16,17,25], we introduced an approach for automatic termination analysis of C that also handles programs whose termination relies on the relation between allocated memory addresses and the data stored at such addresses. This approach is implemented in our tool AProVE [14]. Instead of analyzing C directly, AProVE compiles the program to LLVM code using Clang [9]. Then it constructs a (finite) symbolic execution graph (SEG) such that every program run corresponds to a path through the SEG. AProVE proves memory safety during the construction of the SEG to ensure absence of undefined behavior (which would also allow non-termination). Afterwards, the SEG is transformed into an integer transition system (ITS) such that all paths through the SEG (and hence, the C program) are terminating if the ITS is terminating. To analyze termination of the ITS, AProVE applies standard techniques and calls the tools T2 [7] and LoAT [12,13] to detect non-termination of ITSs. However, like other termination tools for C, up to now AProVE supported dynamic data structures only in a very restricted way.

```
struct list {
  unsigned int value;
  struct list* next;   };

int main() {
  // initialize length
  unsigned int n = nondet_uint();
  // initialize list of length n
  struct list* tail = NULL;
  struct list* curr;
  for (unsigned int k = 0; k < n; k++) {
    curr = malloc(sizeof(struct list));
    curr->value = nondet_uint();
    curr->next = tail;
    tail = curr;                          }
  // traverse list
  struct list* ptr = tail;
  while(ptr != NULL) {
    ptr = *((struct list**)((void*)ptr +
          offsetof(struct list, next)));}}
```

In this paper, we introduce a novel technique to analyze C programs on lists. In the program on the right, nondet_uint returns a random unsigned integer. The for loop creates a list of n random numbers if $n > 0$. The while loop traverses this list via pointer arithmetic: Starting with tail, it computes the address of the next field of the current element by adding the offset of the next field within a list to the address of the current list and dereferencing the computed address (i.e., the content of the

[*] funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2)

next field). This is done by offsetof, defined in the C library stddef.h.[1] Since the list is acyclic and the next pointer of its last element is the null pointer, list traversal always terminates. Of course, the while loop could also traverse the list via ptr = ptr->next, but in C, memory accesses can be combined with pointer arithmetic. This example contains both the access via curr->next (when initializing the list) and pointer arithmetic (when traversing the list).

We present a new general technique to infer *list invariants* via symbolic execution, which express all properties that are crucial for memory safety and termination. In our example, the list invariant contains the information that dereferencing the next pointer in the while loop is safe and that one finally reaches the null pointer. In general, our novel list invariants allow us to abstract from detailed information about lists (e.g., about their intermediate elements) such that abstract states with "similar" lists can be merged and generalized during the symbolic execution in order to obtain finite SEGs. At the same time, list invariants express enough information about the lists (e.g., their length, their start address, etc.) such that memory safety and termination can still be proved.

We define the abstract states used for symbolic execution in Sect. 2. In Sect. 3, after recapitulating the construction of SEGs, we adapt our techniques for merging and generalizing states from [25] to infer list invariants. Moreover, we adapt those rules for symbolic execution that are affected by introducing list invariants. Sect. 4 discusses the generation of ITSs and the soundness of our approach. Sect. 5 gives an overview on related work. Moreover, we evaluate the implementation of our approach in the tool AProVE using benchmark sets from *SV-COMP* [3] and the *Termination Competition* [15]. All proofs can be found in [18].

Limitations To ease the presentation, in this paper we treat integer types as unbounded. Moreover, we assume that a program consists of a single non-recursive function and that values may be stored at any address. Our approach can also deal with bitvectors, data alignments, and programs with arbitrary many (possibly recursive) functions, see [11,16,25] for details. However, so far only lists without sharing can be handled by our new technique. Extending it to more general recursive data structures is one of the main challenges for future work.

2 Abstract States for Symbolic Execution

The LLVM code for the for loop is given on the next page. It is equivalent to the code produced by Clang without optimizations on a 64-bit computer. We explain it in detail in Sect. 3. To ease readability, we omitted instructions and keywords that are irrelevant for our presentation, renamed variables, and wrote list instead of struct.list. Moreover, we gave the C instructions (in gray) before the corresponding LLVM code. The code consists of several *basic blocks* including cmpF and bodyF (corresponding to the loop comparison and body).

[1] Note that ptr + n increases ptr by n times the size of the type *ptr. As we want to increase ptr by a number of bytes and ptr is not an i8 pointer, we first cast ptr to void*. Then ((void*)ptr + offsetof(struct list, next)) contains the next pointer, so we cast our computed address to struct list** before dereferencing it.

```
list = type { i32, list* }

define i32 @main() { ...
cmpF:
  k < n
  0: k = load i32, i32* k_ad
  1: kltn = icmp ult i32 k, n
  2: br i1 kltn, label bodyF, label initPtr
bodyF:
  curr = malloc(sizeof(struct list));
  0: mem = call i8* @malloc(i64 16)
  1: curr = bitcast i8* mem to list*
  curr->value = nondet_uint();
  2: nondet = call i32 @nondet_uint()
  3: curr_val = getelementptr list,
              list* curr, i32 0, i32 0
  4: store i32 nondet, i32* curr_val
  curr->next = tail;
  5: tail = load list*, list** tail_ptr
  6: curr_next = getelementptr list,
               list* curr, i32 0, i32 1
  7: store list* tail, list** curr_next
  tail = curr;
  8: store list* curr, list** tail_ptr
  k++
  9: kinc = add i32 k, 1
  10:store i32 kinc, i32* k_ad
  11:br label cmpF
  ...                                      }
```

We now recapitulate the *abstract states* of [25] used for symbolic execution and extend them by a component LI for list invariants, i.e., they have the form $((\mathtt{b}, i), LV, AL, PT, LI, KB)$. The first component is a *program position* $(\mathtt{b}, i)$, indicating that instruction i of block $\mathtt{b}$ is executed next. $\mathit{Pos} \subseteq (\mathit{Blks} \times \mathbb{N})$ is the set of all program positions, and Blks are all basic blocks.

The second component is a partial injective function $LV : \mathcal{V}_{\mathcal{P}} \rightharpoonup \mathcal{V}_{sym}$, which maps *local program variables* $\mathcal{V}_{\mathcal{P}}$ of the program $\mathcal{P}$ to an infinite set $\mathcal{V}_{sym}$ of symbolic variables with $\mathcal{V}_{sym} \cap \mathcal{V}_{\mathcal{P}} = \varnothing$. We identify LV with the set of equations $\{\mathtt{x} = LV(\mathtt{x}) \mid \mathtt{x} \in \mathit{domain}(LV)\}$ and we often extend LV to a function from $\mathcal{V}_{\mathcal{P}} \uplus \mathbb{N}$ to $\mathcal{V}_{sym} \uplus \mathbb{N}$ by defining $LV(n) = n$ for all $n \in \mathbb{N}$.

The third component of each state is a set AL of (bytewise) allocations $[\![v_1, v_2]\!]$ with $v_1, v_2 \in \mathcal{V}_{sym}$, which indicate that $v_1 \leq v_2$ and that all addresses between v_1 and v_2 have been allocated. We require any two entries $[\![v_1, v_2]\!]$ and $[\![w_1, w_2]\!]$ from AL with $v_1 \neq w_1$ or $v_2 \neq w_2$ to be disjoint.

The fourth and fifth components PT and LI model the memory contents. PT contains "points-to" entries of the form $v_1 \hookrightarrow_{\mathtt{ty}} v_2$ where $v_1, v_2 \in \mathcal{V}_{sym}$ and $\mathtt{ty}$ is an LLVM type, meaning that the address v_1 of type $\mathtt{ty}$ points to v_2. In contrast, the set LI of *list invariants* (which is new compared to [25]) does not describe pointwise memory contents but contains invariants $v_{ad} \overset{v_\ell}{\hookrightarrow}_{\mathtt{ty}} [(\mathit{off}_i : \mathtt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n$ where $n \in \mathbb{N}_{>0}$, $v_{ad}, v_\ell, v_i, \hat{v}_i \in \mathcal{V}_{sym}$, $\mathit{off}_i \in \mathbb{N}$ for all $1 \leq i \leq n$, $\mathtt{ty}$ and $\mathtt{ty}_i$ are LLVM types for all $1 \leq i \leq n$, and there is exactly one "recursive field" $1 \leq j \leq n$ such that $\mathtt{ty}_j = \mathtt{ty*}$.[2] Such an invariant represents a `struct ty` with n fields that corresponds to a recursively defined list of length v_ℓ. Here, v_{ad} points to the first list element, the i-th field starts at address $v_{ad} + \mathit{off}_i$ (i.e., with offset off_i)[3] and has type $\mathtt{ty}_i$, and the values of the i-th fields of the first and last list element are v_i and $\hat{v}_i$, respectively. For example, the following list invariant (1) represents all lists of length x_ℓ and type `list` whose elements store a 32-bit integer in their first field and the pointer to the next element in their second field with offset 8. The first list element starts at address $x_{\mathtt{mem}}$, the second starts at address $x_{\mathtt{next}}$, and the last element contains the null pointer. Moreover, the first element stores the integer value $x_{\mathtt{nd}}$ and the last list element stores the integer $\hat{x}_{\mathtt{nd}}$.

[2] Soundness of our approach is not affected if there are other recursive fields, but our symbolic execution technique for list traversal on list invariants in Sect. 3.2.2 can only be applied if the traversal is done along field j.

[3] The field offsets can be computed using the data layout string in the LLVM program.

$$x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)] \tag{1}$$

For example, this invariant represents the list with the allocation $[\![x_{\texttt{mem}}, x_{\texttt{mem}}{+}15]\!]$, where the first four bytes store the integer 5 and the last eight bytes store the pointer $x_{\texttt{next}}$, and the allocation $[\![x_{\texttt{next}}, x_{\texttt{next}}{+}15]\!]$, where the first four bytes store the integer 2 and the last eight bytes store the null pointer (i.e., the address 0). Here, we have $x_\ell = 2$. Sect. 3.2.2 will show that the expressiveness of our list invariants is indeed needed to prove termination of programs that traverse a list.

The last component of a state is a <u>k</u>nowledge <u>b</u>ase KB of quantifier-free first-order formulas that express integer arithmetic properties of $\mathcal{V}_{sym}$. We identify *sets* of first-order formulas $\{\varphi_1, \ldots, \varphi_m\}$ with their conjunction $\varphi_1 \wedge \ldots \wedge \varphi_m$.

A special state ERR is reached if we cannot prove absence of undefined behavior (e.g., if memory safety might be violated by dereferencing the null pointer).

As an example, the following abstract state (2) represents concrete states at the beginning of the block `cmpF`, where the program variable `curr` is assigned the symbolic variable $x_{\texttt{mem}}$, the allocation $[\![x_{\texttt{k\_ad}}, x^{end}_{\texttt{k\_ad}}]\!]$ consisting of 4 bytes stores the value $x_{\texttt{kinc}}$, and $x_{\texttt{mem}}$ points to the first element of a list of length x_ℓ (equal to $x_{\texttt{kinc}}$) that satisfies the list invariant (1). (This state will later be obtained during the symbolic execution, see State O in Fig. 3 in Sect. 3.1.)

$$\boxed{\begin{array}{l}(\texttt{cmpF}, 0),\ \{\texttt{curr} = x_{\texttt{mem}},\ \texttt{kinc} = x_{\texttt{kinc}},\ \ldots\},\ \{[\![x_{\texttt{k\_ad}},\ x^{end}_{\texttt{k\_ad}}]\!],\ \ldots\},\ \{x_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} x_{\texttt{kinc}},\ \ldots\},\\ \{x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\},\ \{x^{end}_{\texttt{k\_ad}} = x_{\texttt{k\_ad}} + 3,\ x_\ell = x_{\texttt{kinc}},\ \ldots\}\end{array}} \tag{2}$$

A state $s = (p, LV, AL, PT, LI, KB)$ is *represented by a formula* $\langle s \rangle$ which contains KB and encodes AL, PT, and LI in first-order logic. This allows us to use standard SMT solving for all reasoning during the construction of the SEG. Moreover, $\langle s \rangle$ is also used for the generation of the ITS afterwards. The encoding of AL and PT is as in [25], see [18]: $\langle s \rangle$ contains formulas which express that allocated addresses are positive, that allocations represent disjoint memory areas, that equal addresses point to equal values, and that addresses are different if they point to different values. For each element of LI, we add the following new formulas to $\langle s \rangle$ which express that the list length v_ℓ is ≥ 1 and the address v_{ad} of the first element is not null. If $v_\ell = 1$, then the values v_i and $\hat{v}_i$ of the fields in the first and the last element are equal. If $v_\ell \geq 2$, then the `next` pointer v_j in the first element must not be null. Finally, if there is a field whose values v_k and $\hat{v}_k$ differ in the first and the last element, then the length v_ℓ must be ≥ 2.

$$\begin{array}{l}\{v_\ell \geq 1 \wedge v_{ad} \geq 1 \mid (v_{ad} \overset{v_\ell}{\hookrightarrow}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI\} \cup\\ \{\bigwedge_{i=1}^n v_i = \hat{v}_i \mid (v_{ad} \overset{v_\ell}{\hookrightarrow}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI \text{ and } \models \langle s \rangle \Rightarrow v_\ell = 1\} \cup\\ \{v_j \geq 1 \mid (v_{ad} \overset{v_\ell}{\hookrightarrow}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI \text{ with } \texttt{ty}_j = \texttt{ty}{*} \text{ and } \models \langle s \rangle \Rightarrow v_\ell \geq 2\} \cup\\ \{v_\ell \geq 2 \mid (v_{ad} \overset{v_\ell}{\hookrightarrow}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI \text{ and } \exists k \in \mathbb{N}_{>0}, k \leq n, \text{ s.t. } \models \langle s \rangle \Rightarrow v_k \neq \hat{v}_k\}\end{array}$$

In *concrete* states c, all values of variables and memory contents are determined uniquely. To ease the formalization, we assume that all integers are unsigned and refer to [16] for the general case. So for all $v \in \mathcal{V}_{sym}(c)$ (i.e., all $v \in \mathcal{V}_{sym}$ occurring in c) we have $\models \langle c \rangle \Rightarrow v = n$ for some $n \in \mathbb{N}$. Moreover, here PT only contains information about allocated addresses and $LI = \varnothing$ since the abstract knowledge in list invariants is unnecessary if all memory contents are known.

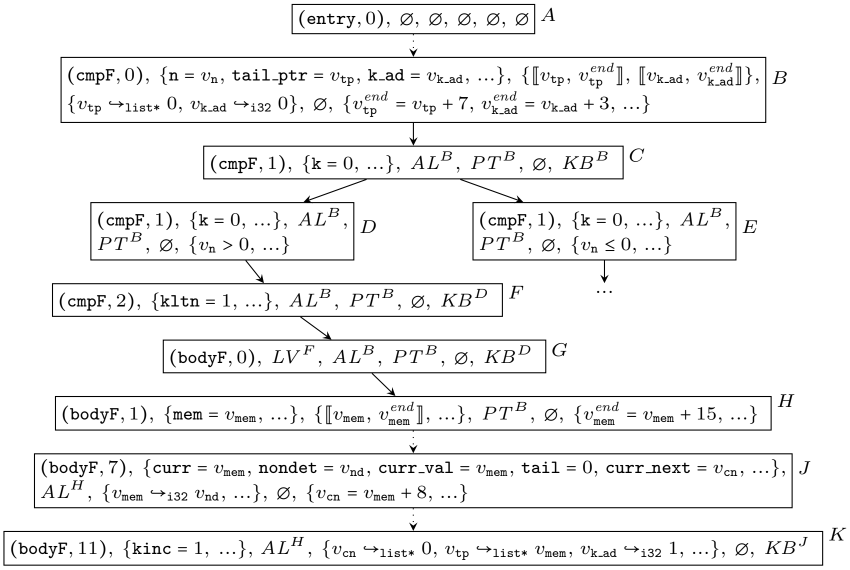

$(\texttt{entry}, 0),\ \varnothing,\ \varnothing,\ \varnothing,\ \varnothing,\ \varnothing$
A
$(\texttt{cmpF}, 0),\ \{\texttt{n} = v_{\texttt{n}},\ \texttt{tail\_ptr} = v_{\texttt{tp}},\ \texttt{k\_ad} = v_{\texttt{k\_ad}},\ \ldots\},\ \{[\![v_{\texttt{tp}},\ v_{\texttt{tp}}^{end}]\!],\ [\![v_{\texttt{k\_ad}},\ v_{\texttt{k\_ad}}^{end}]\!]\},$
$\{v_{\texttt{tp}} \hookrightarrow_{\texttt{list*}} 0,\ v_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} 0\},\ \varnothing,\ \{v_{\texttt{tp}}^{end} = v_{\texttt{tp}} + 7,\ v_{\texttt{k\_ad}}^{end} = v_{\texttt{k\_ad}} + 3,\ \ldots\}$
B
$(\texttt{cmpF}, 1),\ \{\texttt{k} = 0,\ \ldots\},\ AL^B,\ PT^B,\ \varnothing,\ KB^B$
C
$(\texttt{cmpF}, 1),\ \{\texttt{k} = 0,\ \ldots\},\ AL^B,$
$PT^B,\ \varnothing,\ \{v_{\texttt{n}} > 0,\ \ldots\}$
D
$(\texttt{cmpF}, 1),\ \{\texttt{k} = 0,\ \ldots\},\ AL^B,$
$PT^B,\ \varnothing,\ \{v_{\texttt{n}} \leq 0,\ \ldots\}$
E
$\ldots$
$(\texttt{cmpF}, 2),\ \{\texttt{kltn} = 1,\ \ldots\},\ AL^B,\ PT^B,\ \varnothing,\ KB^D$
F
$(\texttt{bodyF}, 0),\ LV^F,\ AL^B,\ PT^B,\ \varnothing,\ KB^D$
G
$(\texttt{bodyF}, 1),\ \{\texttt{mem} = v_{\texttt{mem}},\ \ldots\},\ \{[\![v_{\texttt{mem}},\ v_{\texttt{mem}}^{end}]\!],\ \ldots\},\ PT^B,\ \varnothing,\ \{v_{\texttt{mem}}^{end} = v_{\texttt{mem}} + 15,\ \ldots\}$
H
$(\texttt{bodyF}, 7),\ \{\texttt{curr} = v_{\texttt{mem}},\ \texttt{nondet} = v_{\texttt{nd}},\ \texttt{curr\_val} = v_{\texttt{mem}},\ \texttt{tail} = 0,\ \texttt{curr\_next} = v_{\texttt{cn}},\ \ldots\},$
$AL^H,\ \{v_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} v_{\texttt{nd}},\ \ldots\},\ \varnothing,\ \{v_{\texttt{cn}} = v_{\texttt{mem}} + 8,\ \ldots\}$
J
$(\texttt{bodyF}, 11),\ \{\texttt{kinc} = 1,\ \ldots\},\ AL^H,\ \{v_{\texttt{cn}} \hookrightarrow_{\texttt{list*}} 0,\ v_{\texttt{tp}} \hookrightarrow_{\texttt{list*}} v_{\texttt{mem}},\ v_{\texttt{k\_ad}} \hookrightarrow_{\texttt{i32}} 1,\ \ldots\},\ \varnothing,\ KB^J$
K

Fig. 1: SEG for the First Iteration of the `for` Loop

For instance, all concrete states $((\texttt{cmpF}, 0), LV, AL, PT, \varnothing, KB)$ represented by the state (2) contain ℓ allocations of 16 bytes for some $\ell \geq 1$, where in the first four bytes a 32-bit integer is stored and in the last eight bytes the address of the next allocation (or 0, in case of the last allocation) is stored.

See [18] for a formal definition to determine which concrete states are represented by a state s. To this end, as in [25] we define a *separation logic* formula $\langle s \rangle_{SL}$ which also encodes the knowledge contained in the memory components of states. To extend this formula to list invariants, we use a fragment similar to *quantitative* separation logic [4], extending conventional separation logic by list predicates. For any state s, we have $\models \langle s \rangle_{SL} \Rightarrow \langle s \rangle$, i.e., $\langle s \rangle$ is a weakened version of $\langle s \rangle_{SL}$ that we use for symbolic execution and the termination proof.

3 Symbolic Execution with List Invariants

We first recapitulate the construction of SEGs. Then, Sect. 3.1 extends the technique for *merging* and generalization of states from [25] to infer list invariants. Finally, we adapt the rules for symbolic execution to list invariants in Sect. 3.2.

Our symbolic execution starts with a state A at the first instruction of the first block (called `entry` in our example). Fig. 1 shows the first iteration of the `for` loop. Dotted arrows indicate that we omitted some symbolic execution steps. For every state, we perform symbolic execution by applying the corresponding inference rule as in [25] to compute its successor state(s) and repeat this until all paths end in return states. We call an SEG with this property *complete*.

As an example, we recapitulate the inference rule for the `load` instruction in the case where a value is loaded from allocated and initialized memory. It loads the value of type `ty` that is stored at the address `ad` to the program variable `x`. Let *size*(`ty`) denote the size of `ty` in bytes for any LLVM type `ty`. If we can prove that there is an allocation $[\![v_1,\ v_2]\!]$ containing all addresses $LV(\mathtt{ad}), \ldots, LV(\mathtt{ad}) + size(\mathtt{ty}) - 1$ and there exists an entry $(w_1 \hookrightarrow_{\mathtt{ty}} w_2) \in PT$ such that w_1 is equal to the address $LV(\mathtt{ad})$ loaded from, then we transform the state s at position $p = (\mathtt{b}, i)$ to a state s' at position $p^+ = (\mathtt{b}, i+1)$. In s', a fresh symbolic variable w is assigned to $\mathtt{x}$ and $w = w_2$ is added to KB. We write $LV[\mathtt{x} := w]$ for the function where $LV[\mathtt{x} := w](\mathtt{x}) = w$ and $LV[\mathtt{x} := w](\mathtt{y}) = LV(\mathtt{y})$ for all $\mathtt{y} \neq \mathtt{x}$.

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

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{s' = (p^+,\ LV[\mathtt{x} := w],\ AL,\ PT,\ LI,\ KB \cup \{w = w_2\})} \quad \text{if } w \in \mathcal{V}_{sym} \text{ is fresh and}$$

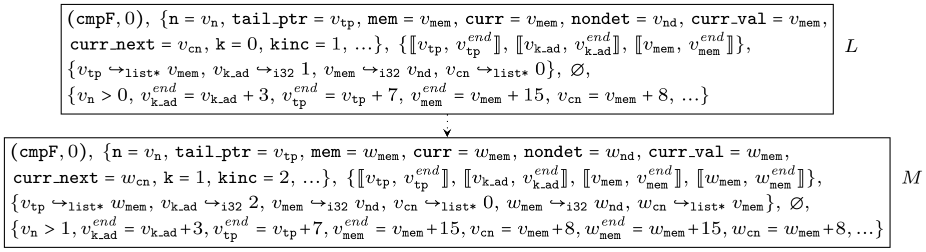
- there is $[\![v_1,\ v_2]\!] \in AL$ with $\models \langle s \rangle \Rightarrow (v_1 \leq LV(\mathtt{ad}) \wedge LV(\mathtt{ad}) + size(\mathtt{ty}) - 1 \leq v_2)$
- there are $w_1, w_2 \in \mathcal{V}_{sym}$ with $\models \langle s \rangle \Rightarrow (LV(\mathtt{ad}) = w_1)$ and $(w_1 \hookrightarrow_{\mathtt{ty}} w_2) \in PT$

In our example, the `entry` block comprises the first three lines of the C program and the initialization of the pointer to the loop variable `k`: First, a non-deterministic unsigned integer is assigned to `n`, i.e., $(\mathtt{n} = v_{\mathtt{n}}) \in LV^B$, where $v_{\mathtt{n}}$ is not restricted. Moreover, memory for the pointers `tail_ptr` and `k_ad` is allocated and they point to `tail = NULL` and `k = 0`, respectively ($\mathtt{tail\_ptr} = v_{\mathtt{tp}}$ and $\mathtt{k\_ad} = v_{\mathtt{k\_ad}}$ with $(v_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} 0), (v_{\mathtt{k\_ad}} \hookrightarrow_{\mathtt{i32}} 0) \in PT^B$). For simplicity, in Fig. 1 we use concrete values directly instead of introducing fresh variables for them. Since we assume a 64-bit architecture, `tail_ptr`'s allocation contains 8 bytes. For the integer value of `k`, only 4 bytes are allocated. Alignments and pointer sizes depend on the memory layout and are given in the LLVM program.

State C results from B by evaluating the `load` instruction at $(\mathtt{cmpF}, 0)$, see the above `load` rule. Thus, there is an *evaluation edge* from B to C.

The next instruction is an integer comparison whose Boolean return value depends on whether the unsigned value of `k` is less than the one of `n`. If we cannot decide the validity of a comparison, we refine the state into two successor states. Thus, the states D and E (with $(v_{\mathtt{n}} > 0) \in KB^D$ and $(v_{\mathtt{n}} \leq 0) \in KB^E$) are reached by *refinement edges* from State C. Evaluating D yields $\mathtt{kltn} = 1$ in F. Therefore, the branch instruction leads to the block `bodyF` in State G. State E is evaluated to a state with $\mathtt{kltn} = 0$. This path branches to the block `initPtr` and terminates quickly as `tail_ptr` points to an empty list.

The instruction at $(\mathtt{bodyF}, 0)$ allocates 16 bytes of memory starting at $v_{\mathtt{mem}}$ in State H. The next instruction casts the pointer to the allocation from `i8*` to `list*` and assigns it to `curr`. Now the allocated area can be treated as a list element. Then `nondet_uint()` is invoked to assign a 32-bit integer to `nondet`. The `getelementptr` instruction computes the address of the integer field of the list element by indexing this field (the second `i32 0`) based on the start address (`curr`). The first index (`i32 0`) specifies that a field of `*curr` itself is computed and not of another list stored after `*curr`. Since the address of the integer value of the list element coincides with the start address of the list element, this instruction assigns $v_{\mathtt{mem}}$ to `curr_val`. Afterwards, the value of `nondet` is stored at `curr_val` ($v_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{nd}}$), the value `0` stored at $v_{\mathtt{tp}}$ is loaded to `tail`, and a second `getelementptr` instruction computes the address of the recursive field of the current list element ($v_{\mathtt{cn}} = v_{\mathtt{mem}} + 8$) and assigns it to `curr_next`,



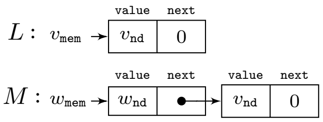
$$(\mathtt{cmpF}, 0),\ \{\mathtt{n} = v_{\mathtt{n}},\ \mathtt{tail\_ptr} = v_{\mathtt{tp}},\ \mathtt{mem} = v_{\mathtt{mem}},\ \mathtt{curr} = v_{\mathtt{mem}},\ \mathtt{nondet} = v_{\mathtt{nd}},\ \mathtt{curr\_val} = v_{\mathtt{mem}},\ \mathtt{curr\_next} = v_{\mathtt{cn}},\ \mathtt{k} = 0,\ \mathtt{kinc} = 1,\ \ldots\},\ \{\llbracket v_{\mathtt{tp}}, v_{\mathtt{tp}}^{end}\rrbracket, \llbracket v_{\mathtt{k\_ad}}, v_{\mathtt{k\_ad}}^{end}\rrbracket, \llbracket v_{\mathtt{mem}}, v_{\mathtt{mem}}^{end}\rrbracket\},\ \{v_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} v_{\mathtt{mem}},\ v_{\mathtt{k\_ad}} \hookrightarrow_{\mathtt{i32}} 1,\ v_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{nd}},\ v_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} 0\},\ \varnothing,\ \{v_{\mathtt{n}} > 0,\ v_{\mathtt{k\_ad}}^{end} = v_{\mathtt{k\_ad}} + 3,\ v_{\mathtt{tp}}^{end} = v_{\mathtt{tp}} + 7,\ v_{\mathtt{mem}}^{end} = v_{\mathtt{mem}} + 15,\ v_{\mathtt{cn}} = v_{\mathtt{mem}} + 8,\ \ldots\} \quad L$$

$$(\mathtt{cmpF}, 0),\ \{\mathtt{n} = v_{\mathtt{n}},\ \mathtt{tail\_ptr} = v_{\mathtt{tp}},\ \mathtt{mem} = w_{\mathtt{mem}},\ \mathtt{curr} = w_{\mathtt{mem}},\ \mathtt{nondet} = w_{\mathtt{nd}},\ \mathtt{curr\_val} = w_{\mathtt{mem}},\ \mathtt{curr\_next} = w_{\mathtt{cn}},\ \mathtt{k} = 1,\ \mathtt{kinc} = 2,\ \ldots\},\ \{\llbracket v_{\mathtt{tp}}, v_{\mathtt{tp}}^{end}\rrbracket, \llbracket v_{\mathtt{k\_ad}}, v_{\mathtt{k\_ad}}^{end}\rrbracket, \llbracket v_{\mathtt{mem}}, v_{\mathtt{mem}}^{end}\rrbracket, \llbracket w_{\mathtt{mem}}, w_{\mathtt{mem}}^{end}\rrbracket\},\ \{v_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} w_{\mathtt{mem}},\ v_{\mathtt{k\_ad}} \hookrightarrow_{\mathtt{i32}} 2,\ v_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{nd}},\ v_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} 0,\ w_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} w_{\mathtt{nd}},\ w_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} v_{\mathtt{mem}}\},\ \varnothing,\ \{v_{\mathtt{n}} > 1, v_{\mathtt{k\_ad}}^{end} = v_{\mathtt{k\_ad}}+3, v_{\mathtt{tp}}^{end} = v_{\mathtt{tp}}+7, v_{\mathtt{mem}}^{end} = v_{\mathtt{mem}}+15, v_{\mathtt{cn}} = v_{\mathtt{mem}}+8, w_{\mathtt{mem}}^{end} = w_{\mathtt{mem}}+15, w_{\mathtt{cn}} = w_{\mathtt{mem}}+8, \ldots\} \quad M$$

Fig. 2: Second Iteration of the `for` Loop

leading to state J. In the path to K, the values of `tail` and `curr` are stored at `curr_next` and `tail_ptr`, respectively ($v_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} 0$, $v_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} v_{\mathtt{mem}}$). Finally, the incremented value of `k` is assigned to `kinc` and stored at `k_ad` ($v_{\mathtt{k\_ad}} \hookrightarrow_{\mathtt{i32}} 1$).

To ensure a finite graph construction, when a program position is reached for the second time, we try to merge the states at this position to a *generalized* state. However, this is only meaningful if the domains of the LV functions of the two states coincide (i.e., the states consider the same program variables). Therefore, after the branch from the loop body back to `cmpF` (see State L in Fig. 2), we evaluate the loop a second time and reach M. Here, a second list element with value $w_{\mathtt{nd}}$ and a `next` pointer $w_{\mathtt{cn}}$ pointing to $v_{\mathtt{mem}}$ has been stored at a new allocation $\llbracket w_{\mathtt{mem}}, w_{\mathtt{mem}}^{end}\rrbracket$. Now, `curr` points to the new element and `k` has been incremented again, so `k_ad` points to 2.



3.1 Inferring List Invariants and Generalization of States

As mentioned, our goal is to merge L and M to a more general state O that represents all states which are represented by L or M. The challenging part during generalization is to find loop invariants automatically that always hold at this position and provide sufficient information to prove termination of the loop. For O, we can neither use the information that `curr` points to a struct whose `next` field contains the null pointer (as in L), nor that its `next` field points to another struct whose `next` field contains the null pointer (as in M).

With the approach of [25], when merging states like L and M where a list has different lengths, the merged state would only contain those list elements that are allocated in both states (often this is only the first element). Then elements which are the null pointer in one but not in the other state are lost. Hence, proving memory safety (and thus, also termination) fails when the list is traversed afterwards, since now there might be `next` pointers to non-allocated memory.

We solve this problem by introducing *list invariants*. In our example, we will infer an invariant stating that `curr` points to a list of length $x_\ell \geq 1$. This invariant also implies that all struct fields are allocated and that there is no sharing.

To this end, we adapt the merging heuristic from [25]. To merge two states s and s' at the same program position with $domain(LV^s) = domain(LV^{s'})$, we introduce a fresh symbolic variable $x_{\mathtt{var}}$ for each program variable `var` and use instantiations μ_s and $\mu_{s'}$ which map $x_{\mathtt{var}}$ to the corresponding symbolic variables of s and s'. For the merged state $\overline{s}$, we set $LV^{\overline{s}}(\texttt{var}) = x_{\texttt{var}}$. Moreover, we identify corresponding variables that only occur in the memory components and extend μ_s and $\mu_{s'}$ accordingly. In a second step, we check which constraints from the memory components and the knowledge base hold in both states in order to find invariants that we can add to the memory components and the knowledge base of $\overline{s}$. For example, if $[\![\mu_s(x),\, \mu_s(x^{end})]\!] \in AL^s$ and $[\![\mu_{s'}(x),\, \mu_{s'}(x^{end})]\!] \in AL^{s'}$ for $x, x^{end} \in \mathcal{V}_{sym}$, then $[\![x,\, x^{end}]\!]$ is added to $AL^{\overline{s}}$. To extend this heuristic to lists, we have to regard several memory entries together. If there is an $\texttt{ad} \in \mathcal{V}_{\mathcal{P}}$ such that $\mu_s(x_{\texttt{ad}}) = v_1^{start}$ and $\mu_{s'}(x_{\texttt{ad}}) = w_1^{start}$ both point to lists of type $\texttt{ty}$ but of different lengths $\ell_s \neq \ell_{s'}$ with $\ell_s, \ell_{s'} \geq 1$, then we create a list invariant.

For a state s we say that v_1^{start} *points to a list of type* $\texttt{ty}$ *with* n *fields and length* ℓ_s *with allocations* $[\![v_k^{start},\, v_k^{end}]\!]$ *and values* $v_{k,i}$ (for $1 \leq k \leq \ell_s$ and $1 \leq i \leq n$) if the following conditions (a) – (d) hold:

(a) $\texttt{ty}$ is an LLVM struct type with subtypes $\texttt{ty}_i$ and field offsets $\mathit{off}_i \in \mathbb{N}$ for all $1 \leq i \leq n$ such that there exists exactly one $1 \leq j \leq n$ with $\texttt{ty}_j = \texttt{ty*}$.

(b) There exist pairwise different $[\![v_k^{start},\, v_k^{end}]\!] \in AL^s$ for all $1 \leq k \leq \ell_s$ and $\models \langle s \rangle \Rightarrow v_k^{end} = v_k^{start} + \mathit{size}(\texttt{ty}) - 1$.

(c) For all $1 \leq k \leq \ell_s$ and $1 \leq i \leq n$ there exist $v_{k,i}^{start}, v_{k,i} \in \mathcal{V}_{sym}$ with $\models \langle s \rangle \Rightarrow v_{k,i}^{start} = v_k^{start} + \mathit{off}_i$ and $(v_{k,i}^{start} \hookrightarrow_{\texttt{ty}_i} v_{k,i}) \in PT^s$.

(d) For all $1 \leq k < \ell_s$ we have $\models \langle s \rangle \Rightarrow v_{k,j} = v_{k+1}^{start}$.

Condition (a) states that $\texttt{ty}$ is a list type with n fields, where the pointer to the next element is in the j-th field. In (b) we ensure that each list element has a unique allocation of the correct size where v_1^{start} is the start address of the first allocation. Condition (c) requires that for the k-th element, the initial address plus the i-th offset points to a value $v_{k,i}$ of type $\texttt{ty}_i$. Finally, (d) states that the recursive field of each element indeed points to the initial address of the next element.

Then, for fresh $x_\ell, x_i, \hat{x}_i \in \mathcal{V}_{sym}$, we add the following list invariant to $LI^{\overline{s}}$.

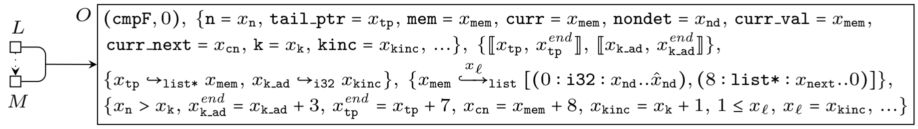
$$x_{\texttt{ad}} \overset{x_\ell}{\hookrightarrow}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : x_i..\hat{x}_i)]_{i=1}^n \tag{3}$$

To ensure that the allocations expressed by the list invariant are disjoint from all allocations in $AL^{\overline{s}}$, we do not use the list allocations $[\![v_k^{start},\, v_k^{end}]\!]$ to infer generalized allocations in $AL^{\overline{s}}$. Similarly, to create $PT^{\overline{s}}$, we only use entries $v \hookrightarrow_{\texttt{ty}} w$ from PT^s and $PT^{s'}$ where v is disjoint from the list addresses, i.e., where $\models \langle s \rangle \Rightarrow v < v_k^{start} \vee v > v_k^{end}$ holds for all $1 \leq k \leq \ell_s$ and analogously for s'. Moreover, we add formulas to $KB^{\overline{s}}$ stating that (A) the length x_ℓ of the list is at least the smaller length of the merged lists, (B) x_ℓ is equal to all variables x which result from merging variables v and w that are equal to the lengths ℓ_s and $\ell_{s'}$ in s and s', and (C) the symbolic variable x_i for the value of the i-th field of the first list element is equal to all variables x with $\mu_s(x) = v_{1,i}$ and $\mu_{s'}(x) = w_{1,i}$ where $v_{1,i}$ and $w_{1,i}$ are the values of the i-th field of the first list element in s and s' (and analogously for the values $\hat{x}_i$ of the last list element):

(A) $\min(\ell_s, \ell_{s'}) \leq x_\ell$

(B) $\bigwedge_{x \in \mu_s^{-1}(v) \cap \mu_{s'}^{-1}(w)} x_\ell = x$ for all $v, w \in \mathcal{V}_{sym}$ with $\models \langle s \rangle \Rightarrow v = \ell_s$ and $\models \langle s' \rangle \Rightarrow w = \ell_{s'}$

(C) $\bigwedge_{x \in \mu_s^{-1}(v_{1,i}) \cap \mu_{s'}^{-1}(w_{1,i})} x_i = x$ and $\bigwedge_{x \in \mu_s^{-1}(v_{\ell_s,i}) \cap \mu_{s'}^{-1}(w_{\ell_{s'},i})} \hat{x}_i = x$ for all $1 \leq i \leq n$



$(\mathtt{cmpF}, 0),\ \{\mathtt{n} = x_{\mathtt{n}},\ \mathtt{tail\_ptr} = x_{\mathtt{tp}},\ \mathtt{mem} = x_{\mathtt{mem}},\ \mathtt{curr} = x_{\mathtt{mem}},\ \mathtt{nondet} = x_{\mathtt{nd}},\ \mathtt{curr\_val} = x_{\mathtt{mem}},$
$\mathtt{curr\_next} = x_{\mathtt{cn}},\ \mathtt{k} = x_{\mathtt{k}},\ \mathtt{kinc} = x_{\mathtt{kinc}},\ \ldots\},\ \{[\![x_{\mathtt{tp}},\ x_{\mathtt{tp}}^{end}]\!],\ [\![x_{\mathtt{k\_ad}},\ x_{\mathtt{k\_ad}}^{end}]\!]\},$
$\{x_{\mathtt{tp}} \hookrightarrow_{\mathtt{list*}} x_{\mathtt{mem}},\ x_{\mathtt{k\_ad}} \hookrightarrow_{\mathtt{i32}} x_{\mathtt{kinc}}\},\ \{x_{\mathtt{mem}} \overset{x_\ell}{\hookrightarrow}_{\mathtt{list}} [(0 : \mathtt{i32} : x_{\mathtt{nd}}..\hat{x}_{\mathtt{nd}}), (8 : \mathtt{list*} : x_{\mathtt{next}}..0)]\},$
$\{x_{\mathtt{n}} > x_{\mathtt{k}},\ x_{\mathtt{k\_ad}}^{end} = x_{\mathtt{k\_ad}} + 3,\ x_{\mathtt{tp}}^{end} = x_{\mathtt{tp}} + 7,\ x_{\mathtt{cn}} = x_{\mathtt{mem}} + 8,\ x_{\mathtt{kinc}} = x_{\mathtt{k}} + 1,\ 1 \le x_\ell,\ x_\ell = x_{\mathtt{kinc}},\ \ldots\}$

Fig. 3: Merging of States

To identify the variables in the list invariant (3) of $\overline{s}$ with the corresponding values in s and s', the instantiations μ_s and $\mu_{s'}$ are extended such that $\mu_s(x_\ell) = \ell_s$, $\mu_{s'}(x_\ell) = \ell_{s'}$, $\mu_s(x_i) = v_{1,i}$, $\mu_{s'}(x_i) = w_{1,i}$, $\mu_s(\hat{x}_i) = v_{\ell_s,i}$, and $\mu_{s'}(\hat{x}_i) = w_{\ell_{s'},i}$ for all $1 \le i \le n$. Similarly, if there already exist list invariants in s and s', for each pair of corresponding variables a new variable is introduced and mapped to its origin by μ_s and $\mu_{s'}$. This adaption of the merging heuristic only concerns the result of merging but not the rules *when* to merge two states. Thus, the same reasoning as in [25] can be used to prove soundness and termination of merging.

In our example, L and M contain lists of length $\ell_L = 1$ and $\ell_M = 2$. To ease the presentation, we re-use variables that are known to be equal instead of introducing fresh variables. If $x_{\mathtt{mem}}$ is the variable for the program variable `curr`, we have $\mu_L(x_{\mathtt{mem}}) = v_{\mathtt{mem}}$ and $\mu_M(x_{\mathtt{mem}}) = w_{\mathtt{mem}}$. Indeed, $v_{\mathtt{mem}}$ resp. $w_{\mathtt{mem}}$ points to a list with values $v_{k,i}$ resp. $w_{k,i}$ as defined in (a)–(d): For the type `list` with $n = 2$, $\mathtt{ty}_1 = \mathtt{i32}$, $\mathtt{ty}_2 = \mathtt{list*}$, $\mathit{off}_1 = 0$, $\mathit{off}_2 = 8$, and $j = 2$ (see (a)), we have $[\![v_{\mathtt{mem}}, v_{\mathtt{mem}}^{end}]\!] \in AL^L$ and $[\![v_{\mathtt{mem}}, v_{\mathtt{mem}}^{end}]\!]$, $[\![w_{\mathtt{mem}}, w_{\mathtt{mem}}^{end}]\!] \in AL^M$, all consisting of $\mathit{size}(\mathtt{list}) = 16$ bytes, see (b). We have $(v_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{nd}}), (v_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} 0) \in PT^L$ with $(v_{\mathtt{cn}} = v_{\mathtt{mem}} + 8) \in KB^L$ and $(v_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{nd}}), (v_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} 0), (w_{\mathtt{mem}} \hookrightarrow_{\mathtt{i32}} w_{\mathtt{nd}}), (w_{\mathtt{cn}} \hookrightarrow_{\mathtt{list*}} v_{\mathtt{mem}}) \in PT^M$ with $(v_{\mathtt{cn}} = v_{\mathtt{mem}} + 8), (w_{\mathtt{cn}} = w_{\mathtt{mem}} + 8) \in KB^M$ (see (c)), so the first list element in M points to the second one (see (d)). Therefore, when merging L and M to a new state O (see Fig. 3), the lists are merged to a list invariant of variable length x_ℓ and we add the formulas (A) $1 \le x_\ell$ and (B) $x_\ell = x_{\mathtt{kinc}}$ to KB^O. By (C), the `i32` value of the first element is identified with $x_{\mathtt{nd}}$, since $\mu_L(x_{\mathtt{nd}})$ is equal to the first value of the first list element in L and $\mu_M(x_{\mathtt{nd}})$ is equal to the first value of the first list element in M. Similarly, the values of the last list elements are identified with 0, as in L and M.

After merging s and s' to a generalized state $\overline{s}$, we continue symbolic execution from $\overline{s}$. The next time we reach the same program position, we might have to merge the corresponding states again. As described in [25], we use a heuristic for constructing the SEG which ensures that after a finite number of iterations, a state is reached that only represents concrete states that are also represented by an *already existing* (more general) state in the SEG. Then symbolic execution can continue from this more general state instead. So with this heuristic, the construction always ends in a complete SEG or an SEG containing the state *ERR*.

We formalized the concept of "generalization" by a symbolic execution rule in [25]. Here, the state $\overline{s}$ is a generalization of s if the conditions $(g1)-(g6)$ hold.

Condition $(g1)$ prevents cycles consisting only of refinement and generalization edges in the graph. Condition $(g2)$ states that the instantiation $\mu : \mathcal{V}_{sym}(\overline{s}) \to \mathcal{V}_{sym}(s) \cup \mathbb{Z}$ maps symbolic variables from the more general state $\overline{s}$ to their counterparts from the more specific state s such that they correspond to the same program variable. Conditions $(g3)$–$(g6)$ ensure that all knowledge present in $\overline{KB}$, $\overline{AL}$, $\overline{PT}$, and $\overline{LI}$ still holds in s with the applied instantiation.

generalization with instantiation μ

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{\overline{s} = (p,\ \overline{LV},\ \overline{AL},\ \overline{PT},\ \overline{LI},\ \overline{KB})} \quad \text{if}$$

$(g1)$ s has an incoming evaluation edge
$(g2)$ $domain(LV) = domain(\overline{LV})$ and $LV(\texttt{var}) = \mu(\overline{LV}(\texttt{var}))$ for all $\texttt{var} \in \mathcal{V_P}$ where LV and $\overline{LV}$ are defined
$(g3)$ $\models \langle s \rangle \Rightarrow \mu(\overline{KB})$
$(g4)$ if $[\![x_1,\, x_2]\!] \in \overline{AL}$, then $[\![v_1,\, v_2]\!] \in AL$ with $\models \langle s \rangle \Rightarrow v_1 = \mu(x_1) \wedge v_2 = \mu(x_2)$
$(g5)$ if $(x_1 \hookrightarrow_{\texttt{ty}} x_2) \in \overline{PT}$,
then $(v_1 \hookrightarrow_{\texttt{ty}} v_2) \in PT$ with $\models \langle s \rangle \Rightarrow v_1 = \mu(x_1) \wedge v_2 = \mu(x_2)$
$(g6)$ if $(x_{\texttt{ad}} \overset{x_\ell}{\hookrightarrow}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : x_i..\hat{x}_i)]_{i=1}^{n}) \in \overline{LI}$,
then either $(v_{\texttt{ad}} \overset{v_\ell}{\hookrightarrow}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^{n}) \in LI$ with

- $\models \langle s \rangle \Rightarrow v_{\texttt{ad}} = \mu(x_{\texttt{ad}}) \wedge v_\ell = \mu(x_\ell)$ and
- $\models \langle s \rangle \Rightarrow v_i = \mu(x_i) \wedge \hat{v}_i = \mu(\hat{x}_i)$ for all $1 \leq i \leq n$,

or v_1^{start} points to a list of type $\texttt{ty}$ and length ℓ with allocations $[\![v_k^{start},\, v_k^{end}]\!]$ and values $v_{k,i}$ (for $1 \leq k \leq \ell, 1 \leq i \leq n$) such that

- $\models \langle s \rangle \Rightarrow v_1^{start} = \mu(x_{\texttt{ad}}) \wedge \ell = \mu(x_\ell)$,
- $\models \langle s \rangle \Rightarrow v_{1,i} = \mu(x_i) \wedge v_{\ell,i} = \mu(\hat{x}_i)$ for all $1 \leq i \leq n$, and
- if $(z_1 \hookrightarrow_{\texttt{ty}} z_2) \in \overline{PT}$,
then $\models \langle s \rangle \Rightarrow \mu(z_1) < v_k^{start} \vee \mu(z_1) > v_k^{end}$ for all $1 \leq k \leq \ell$.

Condition $(g6)$ is new compared to [25] and takes list invariants into account. So for every list invariant $\overline{l}$ of $\overline{s}$ there is either a corresponding list invariant l in s such that lists represented by l in s are also represented by $\overline{l}$ in $\overline{s}$, or there is a concrete list in s that is represented by $\overline{l}$ in $\overline{s}$. The last condition of the latter case ensures that disjointness between the memory domains of $\overline{PT}$ and $\overline{LI}$ is preserved. See [18] for the soundness proof of the extended generalization rule, i.e., that every concrete state represented by s is also represented by $\overline{s}$.

Our merging technique always yields generalizations according to this rule, i.e., the edges from L and M to O in Fig. 3 are generalization edges. Here, one chooses μ_L and μ_M such that $\mu_L(x_{\texttt{mem}}) = v_{\texttt{mem}}$, $\mu_L(x_\ell) = 1$, $\mu_L(x_{\texttt{nd}}) = v_{\texttt{nd}}$, $\mu_L(\hat{x}_{\texttt{nd}}) = v_{\texttt{nd}}$, $\mu_L(x_{\texttt{next}}) = 0$, $\mu_M(x_{\texttt{mem}}) = w_{\texttt{mem}}$, $\mu_M(x_\ell) = 2$, $\mu_M(x_{\texttt{nd}}) = w_{\texttt{nd}}$, $\mu_L(\hat{x}_{\texttt{nd}}) = v_{\texttt{nd}}$, and $\mu_M(x_{\texttt{next}}) = v_{\texttt{mem}}$. In both cases, all conditions of the second case of $(g6)$ with $\ell_L = 1$ and $\ell_M = 2$ are satisfied. With $\mu_L(x_{\texttt{kinc}}) = 1$ resp. $\mu_M(x_{\texttt{kinc}}) = 2$, we also have $\models \langle L \rangle \Rightarrow \mu_L(x_\ell) = \mu_L(x_{\texttt{kinc}})$ resp. $\models \langle M \rangle \Rightarrow \mu_M(x_\ell) = \mu_M(x_{\texttt{kinc}})$.

3.2 Adapting List Invariants

To handle and modify list invariants, three of our symbolic execution rules have to be changed. Sect. 3.2.1 presents a variant of the `store` rule where the list invariant is *extended* by an element. In Sect. 3.2.2, we adapt the `load` rule to load values from the first list element and we present a variant of the `getelementptr` rule for list *traversal*. Soundness of our new rules is proved in [18]. For all other instructions, the symbolic execution rules from [25] remain unchanged.

$$\boxed{\begin{array}{l}(\texttt{bodyF}, 7),\ \{\texttt{tail} = x_{\texttt{mem}},\ \texttt{curr\_next} = y_{\texttt{cn}},\ \ldots\},\ \{[\![y_{\texttt{mem}},\ y^{end}_{\texttt{mem}}]\!],\ \ldots\},\ \{y_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} y_{\texttt{nd}},\ \ldots\},\\ \{x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\},\ \{x_{\texttt{cn}} = x_{\texttt{mem}} + 8,\ y_{\texttt{cn}} = y_{\texttt{mem}} + 8,\ \ldots\}\end{array}} \quad P$$

$$\downarrow$$

$$\boxed{\begin{array}{l}(\texttt{bodyF}, 8),\ \{\texttt{tail} = x_{\texttt{mem}},\ \texttt{curr\_next} = y_{\texttt{cn}},\ \ldots\},\ \{\ldots\},\ \{\ldots\},\\ \{y_{\texttt{mem}} \overset{y_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : y_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{mem}}..0)]\},\ \{y_{\texttt{cn}} = y_{\texttt{mem}} + 8,\ y_\ell = x_\ell + 1,\ \ldots\}\end{array}} \quad Q$$

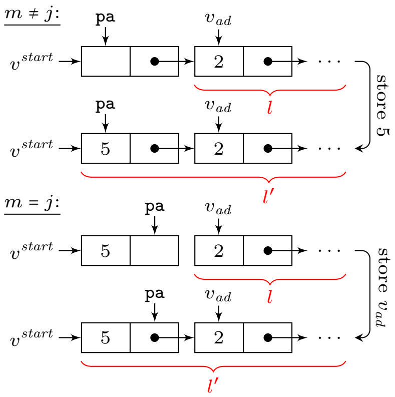
Fig. 4: Extending a List Invariant

3.2.1 List Extension

After merging L and M, symbolic execution continues from the more general state O in Fig. 3. Here, the values of `k` and `kinc` and the length of the list are not concrete but any positive (resp. non-negative) value with $x_\ell = x_{\texttt{kinc}} = x_{\texttt{k}} + 1$. The symbolic execution of O is similar to the steps from B to J in Sect. 3 (see Fig. 1). First, the value $x_{\texttt{kinc}}$ stored at `k_ad` is loaded to `k`. To distinguish whether `k < n` still holds, the next state is refined. From the refined state with `k < n`, we enter the loop body again. A new block $[\![y_{\texttt{mem}}, y^{end}_{\texttt{mem}}]\!]$ of 16 bytes is allocated and $y_{\texttt{mem}}$ is assigned to `mem` and `curr`. Then, a new unknown value $y_{\texttt{nd}}$ is assigned to `nondet`. The address of the `i32` value of the current element (equal to $y_{\texttt{mem}}$) is computed by the first `getelementptr` instruction of the loop and the value $y_{\texttt{nd}}$ of `nondet` is stored at it. The second `getelementptr` instruction computes the address $y_{\texttt{cn}}$ of the recursive field and results in State P in Fig. 4, where $y_{\texttt{cn}} = y_{\texttt{mem}} + 8$ is added to KB^P. Now, `store` sets the address of the `next` field to the head of the list created in the previous iteration. Since this instruction extends the list by an element, instead of adding $y_{\texttt{cn}} \hookrightarrow_{\texttt{list*}} x_{\texttt{mem}}$ to PT^Q, we extend the list invariant: The length is set to y_ℓ and identified with $x_\ell + 1$ in KB^Q. The pointer $x_{\texttt{mem}}$ to the first element is replaced by $y_{\texttt{mem}}$, while the first recursive field in the list gets the value $x_{\texttt{mem}}$. Since $(y_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} y_{\texttt{nd}}) \in PT^P$, $y_{\texttt{nd}}$ is the value of the first `i32` integer in the list. We remove all entries from PT^Q that are already contained in the new list invariant, e.g., $y_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} y_{\texttt{nd}}$.

To formalize this adaption of list invariants, we introduce a modified rule for `store` in addition to the one in [25]. It handles the case where there is a concrete list at some address v^{start}, `pa` points to the m-th field of this list's first element, one wants to store a value t at the address `pa`, and one already has a list invariant l for the "tail" of the list in the j-th field (if $m \neq j$) resp. for the list at the address t (if $m = j$). In all other cases, the ordinary `store` rule is applied.

More precisely, let the list invariant l describe a list of length v_l at the address v_{ad}. Then l is replaced by a new list invariant l' which describes the list at the address v^{start} after storing t at the address `pa`. Irrespective of whether $m \neq j$ or $m = j$, the resulting list at v^{start} has the list at v_{ad} as its "tail" and thus, its length v'_ℓ is $v_\ell + 1$. We prevent sharing of different elements by removing the allocation $[\![v^{start}, v^{end}]\!]$ of the list and all points-to information of pointers in $[\![v^{start}, v^{end}]\!]$.



list extension (p: "`store ty` t, `ty* pa`", $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{N}$, `pa` $\in \mathcal{V}_{\mathcal{P}}$)

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{s' = (p^+,\ LV,\ AL\backslash\{[\![v^{start},\ v^{end}]\!]\},\ PT',\ LI\backslash\{l\} \cup \{l'\},\ KB')} \quad \text{if}$$

- there is $l = (v_{ad} \overset{v_\ell}{\hookrightarrow}_{\texttt{lty}} [(\mathit{off}_i : \texttt{lty}_i : w_i..\hat{w}_i)]_{i=1}^n) \in LI$ with $\texttt{lty}_j = \texttt{lty}*$
- there is $[\![v^{start}, v^{end}]\!] \in AL$ with $\models \langle s \rangle \Rightarrow v^{end} = v^{start} + \mathit{size}(\texttt{lty}) - 1$
- there exists $1 \le m \le n$ such that $\texttt{ty} = \texttt{lty}_m$ and $\models \langle s \rangle \Rightarrow LV(\texttt{pa}) = v^{start} + \mathit{off}_m$
- $\models \langle s \rangle \Rightarrow v_{ad} = v_j$ if $m \neq j$ and $\models \langle s \rangle \Rightarrow v_{ad} = LV(t)$ if $m = j$
- for all $1 \le i \le n$ with $i \neq m$ there exist $v_i^{start}, v_i \in \mathcal{V}_{sym}$ with $\models \langle s \rangle \Rightarrow v_i^{start} = v^{start} + \mathit{off}_i$ and $(v_i^{start} \hookrightarrow_{\texttt{lty}_i} v_i) \in PT$
- $PT' = \{(x_1 \hookrightarrow_{\texttt{sy}} x_2) \in PT \mid \models \langle s \rangle \Rightarrow (v^{end} < x_1) \vee (x_1 + \mathit{size}(\texttt{sy}) - 1 < v^{start})\}$
- $l' = (v^{start} \overset{v'_\ell}{\hookrightarrow}_{\texttt{lty}} [(\mathit{off}_i : \texttt{lty}_i : v_i..\hat{w}_i)]_{i=1}^n)$
- $KB' = KB \cup \{v_m = LV(t),\ v'_\ell = v_\ell + 1\}$, where v_m, v'_ℓ are fresh

3.2.2 List Traversal After the current element $y_{\texttt{mem}}$ is stored at $x_{\texttt{tp}}$ and the value $x_{\texttt{kinc}}$ of `k` is incremented to $y_{\texttt{kinc}}$ and stored at $x_{\texttt{k\_ad}}$, we reach a state R at position $(\texttt{cmpF}, 0)$ by the branch instruction. However, our already existing state O is more general than R, i.e., we can draw a generalization edge from R to O using the generalization rule with the instantiation μ_R where $\mu_R(x_{\texttt{mem}}) = y_{\texttt{mem}}$, $\mu_R(x_{\texttt{nd}}) = y_{\texttt{nd}}$, $\mu_R(x_{\texttt{cn}}) = y_{\texttt{cn}}$, $\mu_R(x_{\texttt{k}}) = x_{\texttt{kinc}}$, $\mu_R(x_{\texttt{kinc}}) = y_{\texttt{kinc}}$, $\mu_R(x_\ell) = y_\ell$, $\mu_R(\hat{x}_{\texttt{nd}}) = \hat{x}_{\texttt{nd}}$, and $\mu_R(x_{\texttt{next}}) = x_{\texttt{mem}}$. Thus, the cycle of the first loop closes here.

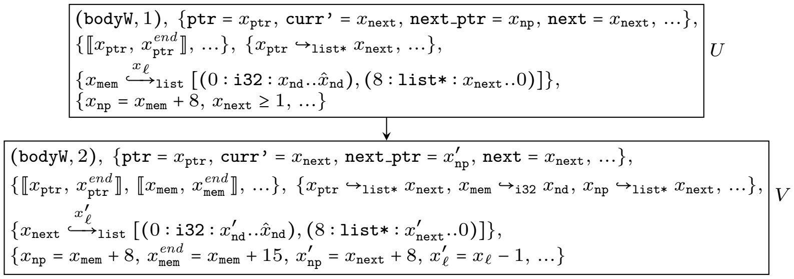
As mentioned, in the path from O to R there is a state at position $(\texttt{cmpF}, 1)$ which is refined (similar to State C). If `k < n` holds, we reach R. The other path with `k` $\not<$ `n` leads out of the `for` loop to the block `initPtr` followed by the `while` loop (see State S and the corresponding LLVM code on the side). The value $x_{\texttt{mem}}$ at address `tail_ptr` is loaded to `tail'` and stored at a new pointer variable `ptr`. State T is reached after the first iteration of the `while` loop body. Here, block `cmpW` loads the value $x_{\texttt{mem}}$ stored at `ptr` to `str`. Since it is not the null pointer, we enter `bodyW`, which corresponds to the body of the `while` loop. First, $x_{\texttt{mem}}$ is cast to an `i8` pointer. Then `getelementptr` computes a pointer $x_{\texttt{np}}$ to the next element by adding 8 bytes to $x_{\texttt{mem}}$. After another cast back to a `list*` pointer, we load the content of the new pointer to `next`. To this end, we need the following new variant of the `load` rule to load values that are described by a list invariant.

$$\begin{tikzcd}
\boxed{\begin{array}{l}(\texttt{initPtr}, 0),\ \{\texttt{tail\_ptr} = x_{\texttt{tp}},\ \ldots\},\ \{\ldots\},\ \{x_{\texttt{tp}} \hookrightarrow_{\texttt{list*}} x_{\texttt{mem}},\ \ldots\},\\ \{x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\},\ \{\ldots\}\end{array}}\ S \arrow[d, dotted] \\
\boxed{\begin{array}{l}(\texttt{cmpW}, 0),\ \{\texttt{ptr} = x_{\texttt{ptr}},\ \texttt{curr'} = x_{\texttt{mem}},\ \texttt{next\_ptr} = x_{\texttt{np}},\\ \texttt{next} = x_{\texttt{next}},\ \ldots\},\ \{[\![x_{\texttt{ptr}},\ x_{\texttt{ptr}}^{end}]\!],\ \ldots\},\ \{x_{\texttt{ptr}} \hookrightarrow_{\texttt{list*}} x_{\texttt{next}},\ \ldots\},\\ \{x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\},\\ \{x_{\texttt{np}} = x_{\texttt{mem}} + 8,\ \ldots\}\end{array}}\ T
\end{tikzcd}$$

```
initPtr:
  0: tail' = load list*, list** tail_ptr
  1: store list* tail', list** ptr
  2: br label cmpW

cmpW:
  0: str = load list*, list** ptr
  1: notnull = icmp ne list* str, null
  2: br i1 notnull, label bodyW, label ret

bodyW:
  0: curr' = bitcast list* str to i8*
  1: next_ptr = getelementptr i8, i8* curr', i64 8
  2: next_ptr' = bitcast i8* next_ptr to list**
  3: next = load list*, list** next_ptr'
  4: store list* next, list** ptr
  5: br label cmpW
```



$$(\texttt{bodyW}, 1),\ \{\texttt{ptr} = x_{\texttt{ptr}},\ \texttt{curr'} = x_{\texttt{next}},\ \texttt{next\_ptr} = x_{\texttt{np}},\ \texttt{next} = x_{\texttt{next}},\ \ldots\},$$
$$\{[\![x_{\texttt{ptr}},\ x^{end}_{\texttt{ptr}}]\!],\ \ldots\},\ \{x_{\texttt{ptr}} \hookrightarrow_{\texttt{list*}} x_{\texttt{next}},\ \ldots\},$$
$$\{x_{\texttt{mem}} \overset{x_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x_{\texttt{next}}..0)]\},$$
$$\{x_{\texttt{np}} = x_{\texttt{mem}} + 8,\ x_{\texttt{next}} \geq 1,\ \ldots\}$$

U

$$(\texttt{bodyW}, 2),\ \{\texttt{ptr} = x_{\texttt{ptr}},\ \texttt{curr'} = x_{\texttt{next}},\ \texttt{next\_ptr} = x'_{\texttt{np}},\ \texttt{next} = x_{\texttt{next}},\ \ldots\},$$
$$\{[\![x_{\texttt{ptr}},\ x^{end}_{\texttt{ptr}}]\!],\ [\![x_{\texttt{mem}},\ x^{end}_{\texttt{mem}}]\!],\ \ldots\},\ \{x_{\texttt{ptr}} \hookrightarrow_{\texttt{list*}} x_{\texttt{next}},\ x_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} x_{\texttt{nd}},\ x_{\texttt{np}} \hookrightarrow_{\texttt{list*}} x_{\texttt{next}},\ \ldots\},$$
$$\{x_{\texttt{next}} \overset{x'_\ell}{\hookrightarrow}_{\texttt{list}} [(0 : \texttt{i32} : x'_{\texttt{nd}}..\hat{x}_{\texttt{nd}}), (8 : \texttt{list*} : x'_{\texttt{next}}..0)]\},$$
$$\{x_{\texttt{np}} = x_{\texttt{mem}} + 8,\ x^{end}_{\texttt{mem}} = x_{\texttt{mem}} + 15,\ x'_{\texttt{np}} = x_{\texttt{next}} + 8,\ x'_\ell = x_\ell - 1,\ \ldots\}$$

V

Fig. 5: Traversing a List Invariant

`load` **from list invariant** (p : "`x = load ty, ty* ad_i`", `x`, `ad_i` $\in \mathcal{V}_\mathcal{P}$)

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

- there is $l = (v_{ad} \overset{v_\ell}{\hookrightarrow}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI$
- there exists $1 \leq i \leq n$ such that $\texttt{ty} = \texttt{ty}_i$ and $\models \langle s \rangle \Rightarrow LV(\texttt{ad\_i}) = v_{ad} + \mathit{off}_i$

With this new `load` rule, the content of the new pointer is identified as $x_{\texttt{next}}$. It is loaded to `next` and stored at $x_{\texttt{ptr}}$. Then we return to the block `cmpW` (State T). Merging T with its predecessor at the same program position is not possible yet since the domains of the respective LV functions do not coincide. Now, $x_{\texttt{next}}$ is loaded to `str` and compared to the null pointer. Since we do not have information about $x_{\texttt{next}}$, T's successor state is refined to a state with $x_{\texttt{next}} = 0$ (which starts a path out of the loop to a return state), and to a state with $x_{\texttt{next}} \geq 1$, which reaches U after a few evaluation steps, see Fig. 5. Now, `getelementptr` computes the pointer $x'_{\texttt{np}} = x_{\texttt{next}} + 8$ to the third element of the list, which is assigned to `next_ptr`. $\langle U \rangle$ contains $x_\ell \geq 2$ since the first and the last pointer value are known to be different ($x_{\texttt{next}} \neq 0$). This information is crucial for creating a new list invariant starting at $x_{\texttt{next}}$, which is used in the next iteration of the loop. Therefore, if our list invariant did not contain variables for the first and the last pointer, we could not prove termination of the program. In such a case where the pointer to the third element of a list invariant is computed and the length of the list is at least two, we *traverse* the list invariant to retain the correspondence between the computed pointer $x'_{\texttt{np}}$ and the new list invariant. In the resulting state V, we represent the first list element by an allocation $[\![x_{\texttt{mem}},\ x^{end}_{\texttt{mem}}]\!]$ and preserve all knowledge about this element that was encoded in the list invariant ($x^{end}_{\texttt{mem}} = x_{\texttt{mem}} + 15$, $x_{\texttt{mem}} \hookrightarrow_{\texttt{i32}} x_{\texttt{nd}}$, $x_{\texttt{np}} \hookrightarrow_{\texttt{list*}} x_{\texttt{next}}$). Moreover, we adapt the list invariant such that it now represents the list at $x_{\texttt{next}}$ (i.e., without its first element) starting with the value $x'_{\texttt{nd}}$. We also relate the length of the new list invariant to the length of the former one ($x'_\ell = x_\ell - 1$).

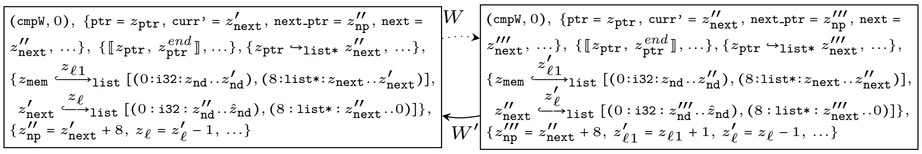
Thus, in addition to the rule for `getelementptr` in [25], we now introduce rules for list traversal via `getelementptr`. The rule below handles the case where the address calculation is based on the type `i8` and the `getelementptr` instruction adds the number of bytes given by the term t to the address `pa`. Here, the offsets in our list invariants are needed to compute the address of the accessed field. We also have similar rules for list traversal via field access (i.e., where the next element is accessed using `curr'->next` as in the `for` loop) and for the case where we cannot prove that the length v_ℓ of the list is at least 2, see [18].

list traversal (p: "`pb = getelementptr i8, i8* pa, i`m t", $t \in \mathcal{V}_\mathcal{P} \cup \mathbb{N}$, $\texttt{pa}, \texttt{pb} \in \mathcal{V}_\mathcal{P}$)

$$\frac{s = (p,\ LV,\ AL,\ PT,\ LI,\ KB)}{s' = (p^+,\ LV[\texttt{pb} := w_j^{start}],\ AL \cup [\![v^{start},\ v^{end}]\!],\ PT',\ LI \backslash \{l\} \cup \{l'\},\ KB')} \quad \text{if}$$

- there is $l = (v_{ad} \xhookrightarrow{v_\ell}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : v_i..\hat{v}_i)]_{i=1}^n) \in LI$ with $\texttt{ty}_j = \texttt{ty*}$, $\models \langle s \rangle \Rightarrow LV(\texttt{pa}) = v_j$, $\models \langle s \rangle \Rightarrow LV(t) = \mathit{off}_j$, and $\models \langle s \rangle \Rightarrow v_\ell \geq 2$
- $PT' = PT \cup \{(v_i^{start} \hookrightarrow_{\texttt{ty}_i} v_i) \mid 1 \leq i \leq n\}$
- $l' = (w^{start} \xhookrightarrow{w_\ell}_{\texttt{ty}} [(\mathit{off}_i : \texttt{ty}_i : w_i..\hat{v}_i)]_{i=1}^n)$
- $KB' = KB \cup \{v^{start} = v_{ad},\ v^{end} = v^{start} + \mathit{size}(\texttt{ty}) - 1,\ w^{start} = v_j,\ w_\ell = v_\ell - 1,$ $w_j^{start} = w^{start} + \mathit{off}_j\} \cup \{v_i^{start} = v_{ad} + \mathit{off}_i \mid 1 \leq i \leq n\}$
- $v^{start}, v^{end}, v_1^{start}, \ldots, v_n^{start}, w^{start}, w_\ell, w_j^{start}, w_1, \ldots, w_n \in \mathcal{V}_{sym}$ are fresh

We continue the symbolic execution of State V in our example and finally obtain a complete SEG with a path from a state W at the position $(\texttt{cmpW}, 0)$ to the next state W' at this position, and a generalization edge back from W' to W using an instantiation $\mu_{W'}$. Both W and W' contain a list invariant similar to T where instead of the length x_ℓ in T, we have the symbolic variables z_ℓ and z'_ℓ in W and W', where $\mu_{W'}(z_\ell) = z'_\ell$ (see [18] for more details).



$(\texttt{cmpW}, 0)$, $\{\texttt{ptr} = z_{\texttt{ptr}},\ \texttt{curr'} = z'_{\texttt{next}},\ \texttt{next\_ptr} = z''_{\texttt{np}},\ \texttt{next} = z''_{\texttt{next}},\ \ldots\}$, $\{[\![z_{\texttt{ptr}},\ z_{\texttt{ptr}}^{end}]\!], \ldots\}$, $\{z_{\texttt{ptr}} \hookrightarrow_{\texttt{list*}} z''_{\texttt{next}}, \ldots\}$, $\{z_{\texttt{mem}} \xhookrightarrow{z_{\ell 1}}_{\texttt{list}} [(0{:}\texttt{i32}{:}z_{\texttt{nd}}..z'_{\texttt{nd}}), (8{:}\texttt{list*}{:}z_{\texttt{next}}..z'_{\texttt{next}})],$ $z'_{\texttt{next}} \xhookrightarrow{z_\ell}_{\texttt{list}} [(0 : \texttt{i32} : z''_{\texttt{nd}}..\hat{z}_{\texttt{nd}}), (8 : \texttt{list*} : z''_{\texttt{next}}..0)]\}$, $\{z''_{\texttt{np}} = z'_{\texttt{next}} + 8,\ z_\ell = z'_\ell - 1,\ \ldots\}$

$(\texttt{cmpW}, 0)$, $\{\texttt{ptr} = z_{\texttt{ptr}},\ \texttt{curr'} = z''_{\texttt{next}},\ \texttt{next\_ptr} = z'''_{\texttt{np}},\ \texttt{next} = z'''_{\texttt{next}},\ \ldots\}$, $\{[\![z_{\texttt{ptr}},\ z_{\texttt{ptr}}^{end}]\!], \ldots\}$, $\{z_{\texttt{ptr}} \hookrightarrow_{\texttt{list*}} z'''_{\texttt{next}}, \ldots\}$, $\{z_{\texttt{mem}} \xhookrightarrow{z'_{\ell 1}}_{\texttt{list}} [(0{:}\texttt{i32}{:}z_{\texttt{nd}}..z''_{\texttt{nd}}), (8{:}\texttt{list*}{:}z_{\texttt{next}}..z''_{\texttt{next}})],$ $z''_{\texttt{next}} \xhookrightarrow{z'_\ell}_{\texttt{list}} [(0 : \texttt{i32} : z'''_{\texttt{nd}}..\hat{z}_{\texttt{nd}}), (8 : \texttt{list*} : z'''_{\texttt{next}}..0)]\}$, $\{z'''_{\texttt{np}} = z''_{\texttt{next}} + 8,\ z'_{\ell 1} = z_{\ell 1} + 1,\ z'_\ell = z_\ell - 1,\ \ldots\}$

4 Proving Termination

To prove termination of a program $\mathcal{P}$, as in [25] the cycles of the SEG are translated to an integer transition system whose termination implies termination of $\mathcal{P}$. The edges of the SEG are transformed into ITS transitions whose application conditions consist of the state formulas $\langle s \rangle$ and equations to identify corresponding symbolic variables of the different states. For evaluation and refinement edges, the symbolic variables do not change. For generalization edges, we use the instantiation μ to identify corresponding symbolic variables. In our example, the ITS has cyclic transitions of the following form:

$$\begin{array}{rcll}
O(x_{\texttt{n}}, x_{\texttt{k}}, x_{\texttt{kinc}}, \ldots) & \to^+ & R(x_{\texttt{n}}, x_{\texttt{k}}, x_{\texttt{kinc}}, \ldots) & \mid \quad x_{\texttt{kinc}} = x_{\texttt{k}} + 1 \wedge x_{\texttt{n}} > x_{\texttt{k}} \wedge \ldots \\
R(x_{\texttt{n}}, x_{\texttt{k}}, x_{\texttt{kinc}}, \ldots) & \to & O(x_{\texttt{n}}, x_{\texttt{kinc}}, \ldots) & \\
W(z_\ell, z'_\ell, \ldots) & \to^+ & W'(z_\ell, z'_\ell, \ldots) & \mid \quad z_\ell = z'_\ell - 1 \wedge z_\ell \geq 1 \wedge \ldots \\
W'(z_\ell, z'_\ell, \ldots) & \to & W(z'_\ell, \ldots) &
\end{array}$$

The first cycle resulting from the generalization edge from R to O terminates since `k` is increased until it reaches `n`. The generalization edge yields a condition identifying $x_{\texttt{kinc}}$ in R with $x_{\texttt{k}}$ in O, since $\mu_R(x_{\texttt{k}}) = x_{\texttt{kinc}}$. With the conditions $x_{\texttt{kinc}} = x_{\texttt{k}} + 1$ and $x_{\texttt{n}} > x_{\texttt{k}}$ (from KB^O), the resulting transitions of the ITS are terminating. The second cycle from the generalization edge from W' to W terminates since the length of the list starting with `curr'` decreases. Although there is no program variable for the length, due to our list invariants the states contain variables for this length, which are also passed to the ITS. Thus, the ITS contains the variable z_ℓ (where z_ℓ in W is identified with z'_ℓ in W' due to $\mu_{W'}(z_\ell) = z'_\ell$). Since the condition $z'_\ell = z_\ell - 1$ is obtained on the path from W to W' and $z_\ell \geq 1$ is part of $\langle W \rangle$ due to the list invariant with length z_ℓ in LI^W, the resulting transitions of the ITS clearly terminate. Analogous to [25, Cor. 11 and Thm. 13], we obtain the following theorem. To prove that a complete SEG represents all program paths, in [25] we used the LLVM semantics defined by the Vellvm project [26]. One now also has to prove soundness of those symbolic execution rules which were modified due to the new concept of list invariants (i.e., generalization, list extension, and list traversal), see [18].

Theorem 1 (Memory Safety and Termination). *Let $\mathcal{P}$ be a program with a complete SEG $\mathcal{G}$. Since a complete SEG does not contain ERR, $\mathcal{P}$ is memory safe for all concrete states represented by the states in $\mathcal{G}$.*[4] *If the ITS corresponding to $\mathcal{G}$ is terminating, then $\mathcal{P}$ is also terminating for all states represented by $\mathcal{G}$.*

5 Conclusion, Related Work, and Evaluation

We presented a new approach for automated proofs of memory safety and termination of C/LLVM-programs on lists. It first constructs a symbolic execution graph (SEG) which overapproximates all program runs. Afterwards, an integer transition system (ITS) is generated from this graph whose termination is proved using standard techniques. The main idea of our new approach is the extension of the states in the SEG by suitable *list invariants*. We developed techniques to infer and modify list invariants automatically during the symbolic execution.

During the construction of the SEG, the list invariants abstract from a concrete number of memory allocations to a list of allocations of variable length while preserving knowledge about some of the contents (the values of the fields of the first and the last element) and the list shape (the start address of the first element, the list length, and the content of the last recursive pointer which allows us to distinguish between cyclic and acyclic lists). They also contain information on the memory arrangement of the list fields which is needed for programs that access fields via pointer arithmetic. The symbolic variables for the list length and the first and last values of list elements are preserved when generating an ITS from the SEG. Thus, they can be used in the termination proof of the ITS (e.g., the variables for list length can occur in ranking functions).

In [5,6,22] we developed a technique for termination analysis of Java, based on a program transformation to *integer term rewrite systems* instead of ITSs. This approach does not require specific list invariants as recursive data structures on the heap are abstracted to terms. However, these terms are unsuitable for C, since they cannot express memory allocations and the connection to their contents.

Separation logic predicates for termination of list programs were also used in [1], but their list predicates only consider the list length and the recursive field,

[4] Our approach can only *prove* but not *disprove* memory safety, i.e., a SEG with the state *ERR* just means that we failed in showing memory safety.

but no other fields or offsets. The tools Cyclist [24] and HipTNT+ [19] are integrated in separation logic systems which also allow to define heap predicates. However, they require annotations and hints which parameters of the list predicates are needed as a termination measure. The tool 2LS [20] also provides basic support for dynamic data structures. But all these approaches are not suitable if termination depends on the contents or the shape of data structures combined with pointer arithmetic. In [10], programs can be annotated with arithmetic and structural properties to reason about termination. In contrast, our approach does not need hints or annotations, but finds termination arguments fully automatically.

We implemented our approach in AProVE [25]. While C programs with lists are very common, existing tools can hardly prove their termination. Therefore, the current benchmark collections for termination analysis contain almost no list programs. In 2017, a benchmark set[5] of 18 typical C-programs on lists was added to the *Termination* category of the *Competition on Software Verification* (*SV-COMP*) [3], where 9 of them are terminating. Two of these 9 programs do not need list invariants, because they just create a list without operating on it afterwards. The remaining seven terminating programs create a list and then traverse it, search for a value, or append lists and compute the length afterwards. Only few tools in *SV-COMP* produced correct termination proofs for programs from this set: HipTNT+ and 2LS failed for all of them. CPAchecker [2] and PeSCo [23] proved termination and non-termination for one of these programs in 2020. UAutomizer [8] proved termination for two and non-termination for seven programs. The termination proofs of CPAchecker, PeSCo, and UAutomizer only concern the programs that just create a list. Our new version of AProVE is the only termination prover[6] that succeeds if termination depends on the shape or contents of a list after its creation. Note that for non-termination, a proof is a single non-terminating program path, so here list invariants are less helpful.

For the *Termination Competition* [15] 2022, we submitted 18 terminating C programs on lists[7] (different from the ones at *SV-COMP*), where two of them just create a list. Three traverse it afterwards (by a loop or recursion), and ten search for a value, where for nine, also the list contents are relevant for termination. Three programs perform common operations like inserting or deleting an element. UAutomizer proves termination for a program that just creates a list but not for programs operating on the list afterwards. With our approach, AProVE succeeds on 17 of the 18 programs. Overall, AProVE and UAutomizer were the two most powerful tools for termination of C in *SV-COMP* 2022 and the *Termination Competition* 2022, with UAutomizer winning the former and AProVE winning the latter competition. To download AProVE, run it via its web interface, and for details on our experiments, see https://aprove-developers.github.io/recursive_structs.

	SV-C T.	**SV-C Non-T.**	**TermCmp T.**
AProVE	7 (of 9)	5 (of 9)	17 (of 18)
UAutomizer	2 (of 9)	7 (of 9)	1 (of 18)

[5] https://github.com/sosy-lab/sv-benchmarks/tree/master/c/termination-memory-linkedlists

[6] We did not compare with the tool VeriFuzz [21], since it does not prove termination but only tests for non-termination and thus, it is unsound for inferring termination.

[7] https://github.com/TermCOMP/TPDB/tree/master/C/Hensel_22

References

1. J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV '06*, LNCS 4144, pages 386–400, 2006. doi:10.1007/11817963_35.
2. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV '11*, LNCS 6806, pages 184–190, 2011. doi:10.1007/978-3-642-22110-1_16.
3. D. Beyer. Progress on software verification: *SV-COMP 2022*. In *Proc. TACAS '22*, LNCS 13244, pages 375–402, 2022. For the results of *SV-COMP* '22, see https://sv-comp.sosy-lab.org/2022/. doi:10.1007/978-3-030-99527-0_20.
4. M. Bozga, R. Iosif, and S. Perarnau. Quantitative separation logic and programs with lists. *J. Aut. Reasoning*, 45(2):131–156, 2010. doi:10.1007/s10817-010-9179-9.
5. M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proc. RTA '11*, LIPIcs 10, pages 155–170, 2011. doi:10.4230/LIPIcs.RTA.2011.155.
6. M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. CAV '12*, LNCS 7358, pages 105–122, 2012. doi:10.1007/978-3-642-31424-7_13.
7. M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: Temporal property verification. In *Proc. TACAS '16*, LNCS 9636, pages 387–393, 2016. doi:10.1007/978-3-662-49674-9_22.
8. Y.-F. Chen, M. Heizmann, O. Lengál, Y. Li, M.-H. Tsai, A. Turrini, and L. Zhang. Advanced automata-based algorithms for program termination checking. In *Proc. PLDI '18*, pages 135–150, 2018. doi:10.1145/3192366.3192405.
9. Clang: https://clang.llvm.org.
10. C. David, D. Kroening, M. Lewis, and J. Vitek. Propositional reasoning about safety and termination of heap-manipulating programs. In *Proc. ESOP '15*, LNCS 9032, pages 661–684, 2015. doi:10.1007/978-3-662-46669-8_27.
11. F. Emrich, J. Hensel, and J. Giesl. AProVE: Modular termination analysis of memory-manipulating C programs. *CoRR*, abs/2302.02382, 2023. doi:10.48550/arXiv.2302.02382.
12. F. Frohn and J. Giesl. Proving non-termination via loop acceleration. In *Proc. FMCAD '19*, pages 221–230, 2019. doi:10.23919/FMCAD.2019.8894271.
13. F. Frohn and J. Giesl. Proving non-termination and lower runtime bounds with LoAT (system description). In *Proc. IJCAR '22*, LNCS 13385, pages 712–722, 2022. doi:10.1007/978-3-031-10769-6_41.
14. J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *J. Aut. Reasoning*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9389-x.
15. J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. The termination and complexity competition. In *Proc. TACAS '19*, LNCS 11429, pages 156–166, 2019. For the results of *TermComp* '22, see https://termination-portal.org/wiki/Termination_Competition_2022. doi:10.1007/978-3-030-17502-3_10.
16. J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *Journal of Logical and Algebraic Methods in Programming*, 97:105–130, 2018. doi:10.1016/j.jlamp.2018.02.004.

17. J. Hensel, C. Mensendiek, and J. Giesl. AProVE: Non-termination witnesses for C programs (competition contribution). In *Proc. TACAS '22*, LNCS 13244, pages 403–407, 2022. doi:10.1007/978-3-030-99527-0_21.
18. J. Hensel and J. Giesl. Proving termination of C programs with lists. *CoRR*, abs/2305.12159, 2023. doi:10.48550/arXiv.2305.12159.
19. T. C. Le, S. Qin, and W. Chin. Termination and non-termination specification inference. In *Proc. PLDI '15*, pages 489–498, 2015. doi:10.1145/2737924.2737993.
20. V. Malík, Š. Martiček, P. Schrammel, M. Srivas, T. Vojnar, and J. Wahlang. 2LS: Memory safety and non-termination. In *Proc. TACAS '18*, LNCS 10806, pages 417–421, 2018. doi:10.1007/978-3-319-89963-3_24.
21. R. Metta, P. Yeduru, H. Karmarkar, and R. K. Medicherla. VeriFuzz 1.4: Checking for (non-)termination (competition contribution). In *Proc. TACAS '23*, LNCS 13994, pages 594–599, 2023. doi:10.1007/978-3-031-30820-8_42.
22. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010. doi:10.4230/LIPIcs.RTA.2010.259.
23. C. Richter and H. Wehrheim. PeSCo: Predicting sequential combinations of verifiers. In *Proc. TACAS '19*, LNCS 11429, pages 229–233, 2019. doi:10.1007/978-3-030-17502-3_19.
24. R. N. S. Rowe and J. Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *Proc. CPP '17*, pages 53–65, 2017. doi:10.1145/3018610.3018623.
25. T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *J. Aut. Reasoning*, 58(1):33–65, 2017. doi:10.1007/s10817-016-9389-x.
26. J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. POPL '12*, pages 427–440, 2012. doi:10.1145/2103656.2103709.