Verifying Tree-Manipulating Programs via CHCs

Marco Faella[1][0000−0001−7617−5489] and Gennaro Parlato[2][0000−0002−8697−2980]

[1] University of Naples Federico II, Italy
m.faella@unina.it
[2] University of Molise, Italy
gennaro.parlato@unimol.it

Abstract. Programs that manipulate tree-shaped data structures often require complex, specialized proofs that are difficult to generalize and automate. This paper introduces a unified, foundational approach to verifying such programs. Central to our approach is the *knitted-tree encoding*, modeling each program execution as a tree structure capturing input, output, and intermediate states. Leveraging the compositional nature of knitted-trees, we encode these structures as constrained Horn clauses (CHCs), reducing verification to CHC satisfiability task. To illustrate our approach, we focus on *memory safety* and show how it naturally leads to simple, modular invariants.

1 Introduction

Ensuring automatic or semi-automatic verification of programs that manipulate dynamic memory (heaps) presents numerous challenges. First, heap can grow unboundedly in size and store unbounded data, requiring expressive logics to capture intricate invariants, preconditions, and postconditions. Next, dynamic shape changes complicate the maintenance of structural constraints, such as binary search tree ordering or balance. Aliasing further obscure these constraints and break invariants. Unbounded recursion and loops, common in tree algorithms, add complexity by making termination reasoning non-trivial. Finally, incomplete or missing specifications often force verifiers to infer properties on-the-fly, and integrating various verification techniques (e.g., SMT-solving, abstract interpretation, and interactive theorem proving) remains a nontrivial task.

This paper presents a foundational approach for automated analysis of heap-manipulating programs, particularly those involving tree data structures. By combining automata and logic-based methods, we reduce verification to checking satisfiability of constrained Horn clauses (CHCs). This reduction allows us to capitalize on advancements in CHC solvers [16,3]. We demonstrate our approach on the *memory safety problem*, ensuring that no execution causes crashes (e.g., null-pointer dereferences, use-after-free, or illegal frees) or nontermination.

The core of our methodology maps an entire program execution π on an input data tree T into a single tree data structure called a **knitted-tree**. This structure encapsulates the input, output, and all intermediate configurations of π. Its underlying tree, or *backbone*, is derived from T by adding a fixed number of inactive nodes to allow dynamic node allocation. Each node is labeled by a sequence of records, or *frames*, connected in a global linear sequence called the *lace*, where consecutive frames may belong to the same node or adjacent nodes, resembling a knitting tree. Each frame describes changes to the associated node (e.g., pointer updates) and records the current program state, preserving the backbone's original structure while also representing the final heap that may differ graph-wise. However, the knitted-tree's parameters – the number of extra nodes added to the backbone and the number of frames per node – may not capture every possible execution, potentially excluding some from our analysis.

```
      pointer $head, prev, cur, tmp$
      int $key$
 0 :  $cur := head$ ;
 1 :  while ($cur \neq$ nil && $cur \to val \neq key$) do
 2 :      $tmp := cur \to next$ ;
 3 :      $cur \to next := prev$ ;
 4 :      $prev := cur$ ;
 5 :      $cur := tmp$ ;
      od ;
 6 :  if ($cur \neq$ nil) then  We found the key
 7 :      $tmp := cur \to next$ ;
 8 :      $head \to next := tmp$ ; Rewind for $head$
 9 :      $head := cur$ ;  Rewind for $cur$
      else
10 :      $head := prev$ ;
      fi ;
11 :  exit ;
```

Fig. 1: Running example.

Example. We illustrate our encoding method with a simple program shown in Fig. 1 that manipulates a singly linked list, specifically designed to highlight the key features of our encoding methodology. The program takes a list of integers with *head* pointing to the first node and a value stored in *key*. It reverses the list up to and including the first node containing the key, then appends the remaining nodes. For example, given the input list $1 \to 2 \to 3 \to 4 \to 5$ and $key = 3$, the output list is: $3 \to 2 \to 1 \to 4 \to 5$. The knitted-tree corresponding to this program execution is shown in Fig. 2.

The knitted-tree's structure matches the input list. The label of each node is displayed next to it, with the lace being depicted by red arrows and numbers. In our example, the lace starts at the frame with ordinal 1 of node u_1, takes two local steps to the frames with ordinals 2 and 3, then moves to the frame numbered 4 in node u_2, and so on. Note that consecutive frames in the lace either belong to the same node, or to adjacent nodes. Due to space constraints, only a selection of the information contained within each frame is displayed.
Our encoding's main innovation is how it handles pointer fields and variables:

1. an update to a pointer field is stored in its node;
2. an update to a pointer variable is stored in the node it points to.

For example, the lace's first frame includes the event $\langle head := \textbf{here} \rangle$, indicating that *head* initially points to the first node of the input structure. This initial assignment is implicit. The second frame corresponds to the execution of $cur := head$ at line 0. According to rule 2, when a pointer is dereferenced, it may be necessary to traverse the lace backward to find its latest assignment, a process called *rewinding*. In our example, the first rewinding occurs at line 8 when the current value of *head* is needed. In the knitted-tree, frame 15 in node u_4 reaches line 8, but since the label of u_4 does not contain information about

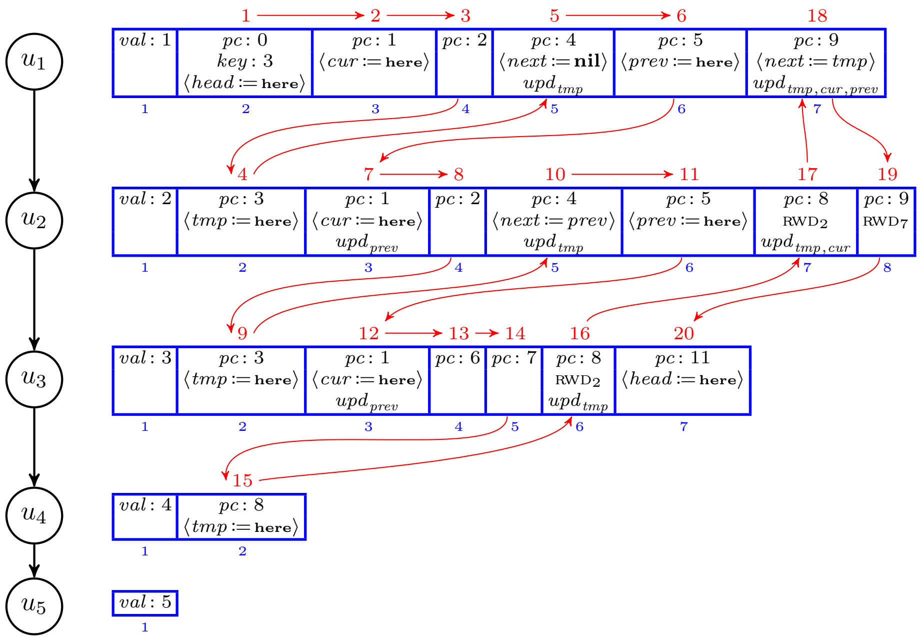


Fig. 2: A knitted-tree of the program in Fig. 1 on the input list $1 \to 2 \to 3 \to 4 \to 5$ and $key = 3$. Blue numbers below the frames represent positions within the label, while red numbers and arrows refer to the lace.

head, rewinding is triggered. Frames 16 and 17 are then added to the lace to go back to node u_1, where *head* currently points, and frame 18 in u_1 reports the effect of the instruction at line 8, consisting in the event $\langle next \coloneqq tmp \rangle$.

Knitted-trees enjoy **compositional properties** that are essential for our CHC-based verification method. These properties allow subtree replacement: a subtree rooted at node v in one knitted-tree can be swapped with a subtree rooted at node v' in another, provided that the labels of v and v' satisfy a local consistency condition expressible in a quantifier-free first-order data theory. This replacement rule enables us to build a CHC system whose minimal model precisely captures the set of valid node labels for the knitted-trees, allowing the detection of any reachable error configuration during execution. If our analysis reports no errors, we need to check whether any execution was truncated by the current choice of parameters. With a small modification, the same CHC system can also reveal whether that is the case; if so, we can increase the parameters to encompass more executions. While verification is undecidable and may not terminate, our method significantly broadens the class of programs and properties amenable to automation. In particular, executions of several well-known programs are fully captured by knitted-trees with suitable parameters, as they traverse nodes a bounded number of times and admit simple CHC solutions.

Organization of the paper. The rest of the paper is structured as follows. Section 2 introduces notation and definitions. Section 3 defines the knitted-tree encoding, and Section 4 details its compositionality. Section 5 describes our reduction to the CHC satisfiability problem, presents a sound procedure to solve the memory safety problem, and analyzes the structure and complexity of the required invariants. Section 6 reviews related work, and Section 7 concludes with future directions. The appendices include substantial supplementary material.

2 Preliminaries

Let $\mathbb{N}$ denote the natural numbers that include 0, $[i, j] \overset{\mathsf{def}}{=} \{k \in \mathbb{N} \mid i \leq k \leq j\}$, and $[j] \overset{\mathsf{def}}{=} [1, j]$.

Trees. A k-ary tree T is a finite, prefix-closed subset of $[k]^*$, where $k \in \mathbb{N}$. Each element in T is a *node*, with the *root* represented by the empty string ε. The tree *edge relation* is implicit: for any $d \in [k]$, if both v and $v.d$ are nodes in T, then $(v, v.d)$ is an *edge*, making $v.d$ the d^{th} child of v, and v the *parent* of $v.d$.

Data signatures. A *data signature* $\mathcal{S}$ consists of pairs $\{id_i : type_i\}_{i\in[n]}$, defining field names and their types (e.g., integers, Booleans $\mathbb{B}$). An *evaluation* ν of $\mathcal{S}$ assigns each field name id a type-specific value, denoted $\nu.id$. The *language* of $\mathcal{S}$, $L(\mathcal{S})$, is the set of all its evaluations.

Data trees. A *data tree* with data signature $\mathcal{S}$, or $\mathcal{S}$-*tree*, is a pair (T, λ) where T is a tree and λ is a labeling function $\lambda : T \to L(\mathcal{S})$ that assigns an evaluation of $\mathcal{S}$ to each node $t \in T$. To simplify notation, the value of a field id at node t can be written as $t.id$ when λ is clear from the context.

Constrained Horn clauses. We use standard first-order logic (FOL) with equality [45] and formulas from a many-sorted, quantifier-free first-order theory $\mathcal{D}$ that includes program-relevant data types like arithmetic, reals, and arrays. We refer to $\mathcal{D}$ as the *data theory.*

Definition 1. *Let R be a set of uninterpreted fixed-arity relation symbols representing unknowns. A* **constrained Horn clause** *(CHC) is a formula of the form $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$ where:* **(i)** *C is a* constraint*, a formula of the data theory $\mathcal{D}$ without symbols from R;* **(ii)** *each B_i is an application $r(v_1, \ldots, v_k)$ of a relation symbol $r \in R$ to first-order variables $v_1, \ldots, v_k$;* **(iii)** *H (the* head*) is either false, or an application $r(v_1, \ldots, v_k)$ as in B_i. A CHC is a* fact *if its body is only C and a* query *if its head is false. A finite set $\mathcal{C}$ of CHCs forms a* system *by conjoining all CHCs with free variables universally quantified. We assume the constraint semantics is predefined as a structure.* □

A CHC system $\mathcal{S}$ with relation symbols R is *satisfiable* if there exists an interpretation for each $r \in R$ that makes all clauses in $\mathcal{S}$ valid. Any such interpretation is called a *solution* of $\mathcal{S}$. The CHC *satisfiability problem* is the computational task of determining whether a given system $\mathcal{S}$ of CHCs is satisfiable.

Each CHC system $\mathcal{S}$ has a unique minimal model under subset ordering,[3] computable as the fixed-point of an operator derived from its clauses [19,35]. We use this fixed-point semantics to ensure the correctness of our reductions.

Heap-manipulating Programs. The *heap* is essential for dynamic memory allocation, allowing memory blocks (*nodes*) to be allocated and deallocated during execution. We assume that nodes have a single data field and one or more pointer fields. A *specific heap state* is defined as follows.

Definition 2. *A* **heap** *is a tuple* $\mathcal{H} = (N, \mathcal{S}, \mathbf{data}, PF)$ *where*

- N *is a finite set of nodes, including a unique element* **nil** *for free memory.*
- $\mathcal{S}$ *is a data signature defining the type of data that can be stored in a node.*
- $\mathbf{data} : N \setminus \{\mathbf{nil}\} \to L(\mathcal{S})$ *is a map modeling the data field of each node.*
- PF *is a finite sequence of distinct pointer fields, each defined as a function of type* $(N \setminus \{\mathbf{nil}\}) \to N$*, representing the pointers of each node.* □

We define k as the number of pointer fields. For example, in Fig. 1, $PF = \{next\}$ and $k = 1$, while for binary trees use $PF = \{left, right\}$ and $k = 2$.

$$\begin{array}{rcl}
\mathbf{Program} & \overset{\text{def}}{=} & decl\ \ block \\
decl & \overset{\text{def}}{=} & (\mathbf{pointer}\ id(,id)^*)^*\ (type\ id(,id)^*)^* \\
block & \overset{\text{def}}{=} & (pc : (ctrl\_stmt \mid heap\_stmt)\ ;)^+ \\
ctrl\_stmt & \overset{\text{def}}{=} & d := exp \mid d_{\text{bool}} := heap\_cond \mid \mathbf{skip} \mid \mathbf{exit} \\
 & & \mid \mathbf{if}\ cond\ \mathbf{then}\ block\ \mathbf{else}\ block\ \mathbf{fi} \\
 & & \mid \mathbf{while}\ cond\ \mathbf{do}\ block\ \mathbf{od} \mid \mathbf{goto}\ pc \\
heap\_stmt & \overset{\text{def}}{=} & \mathbf{new}\ p \mid \mathbf{free}\ p \mid p := \mathbf{nil} \mid p := q \mid p := q \to pfield \\
 & & \mid p \to pfield := \mathbf{nil} \mid p \to pfield := q \\
 & & \mid p \to dfield := exp \mid d := p \to dfield \\
exp & \overset{\text{def}}{=} & d \mid f(exp, \ldots, exp) \\
cond & \overset{\text{def}}{=} & r(exp, \ldots, exp) \mid (\neg)?\ heap\_cond \\
heap\_cond & \overset{\text{def}}{=} & p = q \mid p = \mathbf{nil} \mid p \to pfield = q \mid p \to pfield = \mathbf{nil}
\end{array}$$

Syntax. The syntax of our programming language is shown on the right. Programs begin with declarations of pointer and data variables, followed by labeled instructions. Instructions include assignments, control flow, and heap operations. Data assignments are of the form $d := exp$, where d is a data variable set to the value of the data expression exp. Data expressions are built from data variables and combined using function symbols of the data theory $\mathcal{D}$. Control flow instructions include **skip**, **exit**, **if-then-else** statements, and **while** loops. Boolean conditions ($cond$) are exclusively either *data conditions* or *heap conditions*. Heap conditions can be assigned to a Boolean variable with $d_{\text{bool}} := heap\_cond$, integrating them into Boolean theory. Heap operations include **new** p (creates a new node, initializes its fields to undefined or **nil**, and assigns it to p) and **free** p (deallocates the node pointed by p and sets all pointers pointing to the node to **nil**). We also allow assignment and retrieval of pointer fields (i.e., $pfield \in PF$) and data fields (i.e., $dfield \in \mathcal{S}$). Programs are **valid** if they are well-formed, type-correct,

[3] See [19] for logic programs and [35, Prop. 4.1] for constr. logic programs (or CHCs).

uniquely labeled, and terminate with **exit**. Fig. 1 shows an example of a program.[4] For a program P, PC_P, PV_P, and DV_P represent program counters, pointer variables, and data variables, respectively. The function *succ* defines the successor(s) of a program counter. Statements can have from 0 to 2 successors: most statements have a single successor, denoted by $succ(pc)$; the **exit** statement has no successor; **if-then-else** and **while** have two successors based on a Boolean condition: $succ(pc, true)$ and $succ(pc, false)$. The language does not support function calls directly: non-recursive calls are inlined, and limited recursion, typical in tree-based algorithms, can be simulated (see Appendix A for details).

Semantics. A program P operates on a specialized heap called a *P-heap*, that includes all its pointers and data. A *configuration* of P is a tuple $(\mathcal{H}, \nu_p, \nu_d, pc)$ consisting of a P-heap, an evaluation of the pointer variables, an evaluation of the data variables, and next instruction label. Focusing on tree-based programs, a configuration c is *initial* if it meets the following conditions:

– $\mathcal{H}$ is *isomorphic* to a data tree $\mathcal{T}$ via a bijection ρ that maps each node in $\mathcal{H}$ to a node in $\mathcal{T}$, such that for all nodes x, y in $\mathcal{H}$ and $i \in |PF|$, $y = pf_i(x)$ iff $\rho(y) = \rho(x).i$, where pf_i is the i-th pointer field in PF. We refer to $\mathcal{T}$ as the data tree of c, and may use $\mathcal{T}$ in place of $\mathcal{H}$.
– ν_p maps the first pointer variable declared in P, conventionally denoted by $\widehat{p}$, to the $\mathcal{H}$ node corresponding to the root of $\mathcal{T}$ and maps the other pointer variables to **nil**.
– ν_d assigns each variable a non-deterministic value.
– pc is the label of the first statement in P.

A *transition* $c \to_P c'$ in P occurs by executing the instruction at pc using standard semantics unless noted otherwise. If pc is an **exit** statement, c becomes a *final configuration* with no further transitions. Attempting to dereference or deallocate a **nil** pointer makes c an *error configuration.* An **execution** π of P is a (possibly infinite) sequence of configurations $c_0 c_1 \ldots$ where: **(i)** c_0 is initial, and **(ii)** $c_{i-1} \to_P c_i$ for each $i \in \mathbb{N}$. A finite π that ends in a final or error configuration is a *terminating* or *buggy execution*, respectively. We aim to solve the following (undecidable) problem:

Problem 1. A program P is *memory safe* if all its executions terminate without reaching an error configuration. The **memory safety problem** asks whether a given program is memory safe.

3 Knitted-Trees: Representing Executions as Data Trees

Our approach to solving the memory safety problem uses the knitted-tree encoding, which models a program execution as a single data tree capturing inputs, outputs, and intermediate configurations.

[4] The **while** condition goes beyond our syntax but is easily translatable into it.

We first fix notation and assumptions. The encoding uses two parameters $m, n \in \mathbb{N}$, explained later. For simplicity, we assume that heap nodes have a single data field *val* of an arbitrary type $\mathcal{D}$. We consider a fixed program P and omit it from most notations and statements. Let π be an execution of P starting from an initial configuration c_0, where $\mathcal{T} = (T, \lambda)$ is a k-ary data tree of c_0 with signature $\mathcal{S}^{\Sigma} = \{val : \mathcal{D}\}$, and $\widehat{p}$ points to the root of $\mathcal{T}$. The encoding maps π to a set of data trees $kt(\pi, m, n)$, called the (m,n)-*knitted-trees* of π. We now describe a generic knitted-tree $\mathcal{K} = (K, \mu)$ from the set $kt(\pi, m, n)$.

The backbone. The *backbone* K of $\mathcal{K}$ is the smallest tree such that: (i) the input tree T is a subset of K ($T \subset K$), (ii) all nodes from T are internal nodes in K, (iii) each internal node of K has degree $k+m$, and (iv) K has at least one internal node. Note that, in the special case where T is empty, the backbone K is a full tree of height 2, consisting of a root and its $k+m$ children.

Each node of K represents a distinct heap node. Initially, all nodes in T are active, and the rest are inactive; freeing an active node makes it inactive.

The node signature. The backbone of a knitted-tree depends only on the input data tree and parameters m, n, independent of the execution it represents. Each backbone node is labeled with a sequence of *frames* (a *log*) tracking changes along π. Frames form a doubly linked list called *lace*, to maintain chronological order and enable bidirectional navigation. When consecutive operations involve different nodes, frames are inserted along the backbone path connecting them. The data signature $\mathcal{S}_{\mathcal{K}}$ of an (m,n)-knitted-tree is the following:

$$\mathcal{S}_{\mathcal{K}} = \Big\{ \begin{array}{ll}
\mathit{avail}^i : \mathbb{B}, & \text{Is this frame available?} \\
\mathit{active}^i : \mathbb{B}, & \text{Is this node allocated?} \\
\mathit{val}^i : \mathcal{D}, & \text{Current value of this node's data field} \\
\mathit{pc}^i : PC_P, & \text{Program counter} \\
\{d^i : \mathcal{D}_d\}_{d \in DV_P}, & \text{Current value of the data variables} \\
\{\mathit{upd}^i_p : \mathbb{B}\}_{p \in PV_P}, & \text{Has } p \text{ been updated since the frame } i-1\text{?} \\
\{\mathit{isnil}^i_p : \mathbb{B}\}_{p \in PV_P}, & \text{Is } p \textbf{ nil}\text{?} \\
\mathit{event}^i : \mathit{Event}, & \text{A pointer update, rewind, or error} \\
\mathit{active\_child}^i : \mathbb{B}^{k+m} & \text{Is each child allocated?} \\
\mathit{next}^i : \mathit{Dir} \times [2, n+1], & \text{Link to the next frame} \\
\mathit{prev}^i : \mathit{Dir} \times [2, n] & \text{Link to the previous frame}
\end{array} \Big\}_{i \in [n+1]},$$

where $\mathit{Dir} = \{-, \uparrow\} \cup [k+m]$ encodes the position of an adjacent frame relative to the reference frame. Each node's label, or *log*, has $n+1$ indexed frames in time order. Once a frame f in a log σ is named, its fields are referenced without the index (e.g., $f.pc$ instead of $\sigma.pc^i$). The last frame of a log handles *label overflow* (if more than n frames are needed), via the $(n+1)^{\text{th}}$ frame. The *prev* field holds a value from $Dir \times [n]$, since a frame with index $n+1$ has no successor. The *event* field holds an *event* taken from:

$$\begin{aligned}Event = \big\{&\langle pfield := p\rangle,\ \langle pfield := \mathbf{nil}\rangle,\ \langle p := \mathbf{here}\rangle \mid pfield \in PF, p \in PV_P\big\}\\ &\cup\ \big\{\textsc{rwd}_i \mid i \in [n]\big\} \cup \big\{\textsc{rwd}_{i,p} \mid i \in [n], p \in PV_P\big\}\ \cup\ \big\{\textsc{nop}, \textsc{err}, \textsc{oom}\big\}\,.\end{aligned}$$

The first group represents updates to pointer fields and variables. $\textsc{rwd}_i$ and $\textsc{rwd}_{i,p}$ represent *lace rewinding* events. Other symbols denote the empty event (NOP), null-pointer dereference (ERR), and out-of-memory error (OOM) caused by excessive use of the statement **new**.

3.1 The Labeling Function

The labeling function μ of $\mathcal{K}$ is defined inductively on the length of π.

Base case. π consists of an initial configuration, say $(\mathcal{T}, \nu_p, \nu_d, pc)$. We encode the input tree $\mathcal{T} = (T, \lambda)$ into the backbone by setting the first frame of each node $t \in K$ as follows:

$$\mu(t).avail^1 = false, \qquad \mu(t).active^1 = \begin{cases} true & \text{if } t \in T \\ false & \text{otherwise} \end{cases}$$

$$\mu(t).val^1 = \begin{cases} \lambda(t).val & \text{if } t \in T \\ \text{unspecified} & \text{otherwise} \end{cases} \qquad \mu(t).active\_child^1_j = \begin{cases} true & \text{if } t.j \in T \\ false & \text{otherwise.} \end{cases}$$

All other fields of the first frame are unspecified. The root's *second* frame stores π's initial configuration: $\mu(\varepsilon).avail^2 = false$, $\mu(\varepsilon).prev^2 = (-, 2)$ (a self-loop), $\mu(\varepsilon).pc^2$ is the first statement's label in P, and $\mu(\varepsilon).isnil^2_q = true$ for all pointer variables q different from $\widehat{p}$. If T is not empty, $\mu(\varepsilon).event^2 = \langle \widehat{p} := \mathbf{here}\rangle$ and $\mu(\varepsilon).isnil^2_{\widehat{p}} = false$; otherwise, $\mu(\varepsilon).event^2 = \textsc{nop}$ and $\mu(\varepsilon).isnil^2_{\widehat{p}} = true$. All other frames are marked as available. The root's second frame also copies *active*, *val*, and *active_child* from the first frame.

Inductive case. We start with an overview of the encoding method, its properties, and the required notation. Let $\pi = \overline{\pi}c$, where c is a configuration and $\overline{\pi}$ is a non-empty execution. Assume that $\overline{\mathcal{K}} = (K, \overline{\mu})$ is a knitted-tree in $kt(\overline{\pi}, m, n)$. We define the labeling μ for π by extending the lace of $\overline{\mathcal{K}}$ based on the last instruction executed in π. To aid understanding, we list some invariants for all knitted-trees, providing an informal explanation for brevity.

The lace. Besides individual node logs, we maintain a chronological order of all frames across all nodes. All the unavailable frames in the knitted-tree with index greater than 1 form a doubly linked list called the *lace* using the *next* and *prev* fields of the frames. The first frame in the lace is the root's second frame. Each frame is identified by a pair (u, i), where u is a node and $i \in [n+1]$ is the frame's index. A frame (v, j) is the *lace successor* of frame (u, i), denoted $(u, i) \to_{next} (v, j)$ (and (u, i) is the *lace predecessor* of (v, j), written $(v, j) \to_{prev} (u, i)$) if $i, j > 1$ and one of the following holds:

- $u = v$, $j = i+1$, $\mu(u).\mathit{next}^i = (-, j)$, and $\mu(v).\mathit{prev}^j = (-, i)$;
- v is the l^{th} child of u, $\mu(u).\mathit{next}^i = (l, j)$, and $\mu(v).\mathit{prev}^j = (\uparrow, i)$;
- u is the l^{th} child of v, $\mu(u).\mathit{next}^i = (\uparrow, j)$, and $\mu(v).\mathit{prev}^j = (l, i)$.

Available frames' unspecified fields contribute to $kt(\pi, m, n)$'s non-determinism.

Properties of the frame fields. In our inductive definition, we obtain the knitted-tree for π by extending that of $\overline{\pi}$ by appending frames for π's final step. This helps us assign meanings to fields like upd_p, isnil_p, d, val, pc, and $\mathit{active\_child}$ for the unavailable frames. The upd_p flag tracks changes to the pointer p outside the current node: it is set to *true* in frame (u, i) (for $i > 1$) if p was assigned a non-**nil** value in the part of the lace between frames $(u, i-1)$ and (u, i) (excluding these frames). Thus, if frames $(u, i-1)$ and (u, i) are adjacent in the lace (i.e., $(u, i-1) \rightarrow_{next} (u, i)$, aka an *internal step*), all upd_p flags in (u, i) are *false*. The isnil_p flag is *true* in a frame if $p = \textbf{nil}$ at that point in the execution. The $\mathit{active\_child}$ flags help track the allocation of the child nodes of the backbone. Other fields preserve their usual meanings.

Pushing a frame. Appending a frame to a log involves: *(1)* finding the smallest index i with avail^i is *true*, and *(2)* adding the new frame at position i. Hence, a log behaves like a stack, with the bottom frame at index 1 and the *top frame* being the highest-index frame where *avail* is *false*.

Default values for a frame. Any frame pushed onto a log assumes default values unless specified otherwise. When pushing a frame f on a node u, default values come from the preceding frame in the lace, f^{prev}, or the frame below f in u's log, f^{below}. Note that f^{prev} and f^{below} can be the same. The default values for the fields of f are as follows: for all $p \in PV_P$ and $d \in DV_P$,

- $\mathit{avail} = \mathit{false}$, $\mathit{event} = \text{NOP}$, and $\mathit{upd}_p = \mathit{false}$;
- *active* and *val* are copied from f^{below};
- isnil_p, d, and pc are copied from f^{prev};
- $\mathit{active\_child}_j$ is copied from $f^{\text{prev}}.\mathit{active}$ if f^{prev} belongs to the j^{th} child of u; otherwise, it is copied from $f^{\text{below}}.\mathit{active\_child}_j$;
- *prev* points to f^{prev};
- *next* is unspecified and can take any value in different knitted-trees for the same execution.

Moreover, $f^{\text{prev}}.\mathit{next}$ is updated to point to f, eliminating the non-determinism of the *next* field of the previous frame.

Despite the non-determinism in the *next* field, identifying the last frame f in a lace can be done by checking f and its lace successor f'. Specifically, f is the last frame of the lace if f' is available or f' precedes f in the lace, which happens when field of $f'.\mathit{prev}$ does not point to f.

Henceforth, assume that $\overline{f}$ is the final lace last frame in $\overline{\mathcal{K}}$, located as the top frame of node $\bar{t}$. We first present the encoding of the statements that push a single new frame f on the current node $\bar{t}$. The fields of f are set to default values, except for those specified below.

Encoding of $p \coloneqq$ **nil:** $f.\mathit{pc} = \mathit{succ}(\overline{f}.\mathit{pc})$ and $f.\mathit{isnil}_p = \mathit{true}$.

Encoding of $d \coloneqq exp$: $f.pc = succ(\overline{f}.pc)$ and $f.d$ is set to the value of exp, with variables in DV_P evaluated using their values from $\overline{f}$.

Encoding of skip: $f.pc = succ(\overline{f}.pc)$.

The other statements may operate on different nodes in addition to $\overline{t}$. The main reasons to move to another node are to dereference a pointer or identify the node a pointer field refers to. To get this information, we *rewind the lace* by moving backward to find the most recent assignment to the relevant pointer. For example, to identify the node a pointer variable p points to, we rewind the lace until we find a frame with the event $\langle p \coloneqq \textbf{here}\rangle$.

Lace rewinding function. To enable rewinding, we define the auxiliary function $\mathit{find\_ptr}(p, id)$, which takes a pointer $p \in PV_P$ and a frame ID and returns a sequence of frame IDs by traversing the lace backward from id, until the most recent assignment to p. The sequence uses *shortcuts*, including only the IDs where the lace moves between nodes and where such moves are *relevant* to track p, as indicated by the upd_p flags. For example, in Fig. 2, rewinding from frame 15 to resolve *head* gives the following: [5]

$$\mathit{find\_ptr}(head, (u_4, 2)) = \mathit{find\_ptr}(head, 15) = \big((u_3, 2), (u_2, 2), (u_1, 2)\big) = (9, 4, 1).$$

Frames 9 and 4 have a predecessor that belongs to another node; moreover, they represent the earliest occurrence of their node in the lace. Contrast this with frame 12, which also follows a frame in another node, but is *not* in the sequence because 12 is not the first visit to u_3, and upd_{head} is false in 12. Frame 1 is included as the value of *head* in frame 15 was established in frame 1. Thus, rewinding from 15 adds frames $16, 17$ and 18 for these IDs.

Null pointer dereference. If the instruction located at $\overline{f}.pc$ dereferences a pointer p, and p is **nil**, it indicates an error. Thus, if the flag $isnil_p$ is *true* in $\overline{f}$, we push a new frame with the event ERR onto the current node $\overline{t}$ to indicate a runtime error. We now present the encoding for the other types of statement.

Encoding of $p \to pfield \coloneqq$ nil. We rewind the lace to find the latest assignment to p. Let $(\overline{t}, \overline{i})$ be the identifier of $\overline{f}$, and $id_1 = (u_1, i_1), \ldots, id_l = (u_l, i_l)$ be the sequence $\mathit{find\_ptr}\big(p, (\overline{t}, \overline{i})\big)$. We push a new frame for each element of the sequence $id_1, \ldots, id_l$, to keep a faithful record of the movements necessary to simulate the current statement: for each $j \in [l-1]$, we push a frame onto u_j with the event RWD_{i_j}, without advancing the program counter. Finally, we push a frame f onto u_l with $f.pc = succ(\overline{f}.pc)$ and $f.event = \langle pfield \coloneqq \textbf{nil}\rangle$.

Encoding of $p \to pfield \coloneqq q$. Same as the encoding for $p \to pfield \coloneqq$ **nil**, but the final frame's event is set to $\langle pfield \coloneqq q\rangle$.

Encoding of $p \coloneqq q$. If $isnil_q$ evaluates to *true* in $\overline{f}$, we push a new frame onto $\overline{t}$ with $isnil_p$ set to *true*. Otherwise, a rewinding operation takes the lace to the node pointed by q, where we push a frame with $\langle p \coloneqq \textbf{here}\rangle$. In either case, the last frame also updates the program counter.

[5] Using the global lace positions as frame IDs (red numbers in Fig. 2).

Encoding of new p. Let j be the smallest index in $[k+1, k+m]$ such that $\mathit{active\_child}_j$ is false. If no such index exists, we push a frame on $\bar{t}$ with the event OOM, representing an out-of-memory error. Otherwise, the lace moves to the j^{th} child of $\bar{t}$ and pushes a frame f there with $f.\mathit{pc} = \mathit{succ}(\bar{f}.\mathit{pc})$, $f.\mathit{event} = \langle p \coloneqq \mathbf{here}\rangle$, $f.\mathit{active} = \mathit{true}$, and $f.\mathit{isnil}_p = \mathit{false}$.

Encoding of free p. A rewinding operation takes the lace to the node u pointed by p, where we push a frame f with $\mathit{active} = \mathit{false}$ and an updated pc. In the new frame, isnil_q is set to true for every pointer q that currently points at u, including p. To find such pointers, let (u, i) be the id of f. Then, q points at u if the log of u contains another frame (u, j) such that: $j < i$, (u, j) contains $\langle q \coloneqq \mathbf{here}\rangle$, and the upd_q flag is false in all frames from $(u, j+1)$ to (u, i).

Encoding of $p \coloneqq q \to \mathit{pfield}$. First, the lace moves to the node u pointed by q through a rewinding process. Then, we search in u's log for the most recent assignment to pfield. We look for the largest index i s.t. the frame (u, i) is in use and contains an event of the form $\langle \mathit{pfield} \coloneqq \alpha\rangle$, for some α. If no such index exists, pfield is interpreted as having its default value pointing to a child in the original input tree. We then distinguish the following cases:

[$\alpha = \mathbf{nil}$] We push a frame with $\mathit{isnil}_p = \mathit{true}$ on u.

[$\alpha = r$, for some $r \in PV_P$] If isnil_r is true in the current frame, the lace pushes a frame with $\mathit{isnil}_p = \mathit{true}$. Otherwise, the lace moves again to the node pointed by r, and there it pushes a frame with $\langle p \coloneqq \mathbf{here}\rangle$.

[The log of u does not contain an explicit assignment to pfield] If pfield is the j^{th} field in PF, $u.j \in T$, and $u.j$ is active (as encoded in the flag $\mathit{active\_child}_j$), the lace moves to $u.j$ and pushes a frame with event $\langle p \coloneqq \mathbf{here}\rangle$ there. Otherwise, a frame with $\mathit{isnil}_p = \mathit{true}$ is pushed on u.

The last pushed frame always updates the program counter.

Encoding Boolean conditions and control-flow statements. Data conditions are evaluated locally using variable values in the current frame $\bar{f}$. For heap conditions, we may need to traverse the lace. We focus on conditions of the form $p = q$, since others (e.g., $p \to \mathit{pfield} = q$) can be reduced to this form using auxiliary variables. To evaluate $p = q$, we first check the isnil flags: if both pointers have their isnil flags set to true, the condition is true; if the flags differ, it is false. If this is inconclusive, we rewind the lace to find an assignment to p or q. For example, upon finding $\langle p \coloneqq \mathbf{here}\rangle$ in frame (u, i), we search in u's log for the largest index $j < i$ where frame (u, j) has $\langle q \coloneqq \mathbf{here}\rangle$. If none exists, the condition is false. If found, the condition holds if q was unchanged between (u, j) and (u, i), verified via $\neg \bigvee_{l \in [j+1, i]} \mathit{upd}^l_q$. A new frame is then pushed, updating the program counter accordingly. The process is symmetric if $\langle q \coloneqq \mathbf{here}\rangle$ is found first.

On the choice of parameters m and n. Parameter m bounds the number of allocations: $m = 0$ for programs without allocations, while $m = 1$ is adequate for programs that insert a single new node. The parameter n limits the passes and instructions executed per node; while some programs need unbounded labels, typical tree-like algorithms work with moderate n (usually ≤ 10).

Statement	**Movement**	**Information stored**
$p \to pfield := \mathbf{nil}$	Find p or fail	$\langle pfield := \mathbf{nil}\rangle$
$p \to pfield := q$	Find p or fail	$\langle pfield := q\rangle$ or $\langle pfield := \mathbf{nil}\rangle$
$p := q$	Find q	$\langle p := \mathbf{here}\rangle$ or set $isnil_p$
$p := q \to pfield$	Find q or fail, then find last assignment to *pfield*	$\langle p := \mathbf{here}\rangle$ or set $isnil_p$
$p \to val := exp$	Find p or fail	Update *val*
$d := p \to val$	Find p or fail	Update d
new p	Move to the first inactive child or fail	$\langle p := \mathbf{here}\rangle$ and set *active*
free p	Find p or fail	Reset *active*

Table 1: Summary of the encodings.

3.2 Relations with Program Executions

Our first result establishes that knitted-trees provide an accurate and faithful representation of program executions.

Theorem 1. *Given a program P and parameters m, n, $kt(\cdot, m, n)$ is computable. Moreover, there is a computable function exec such that, for any data tree $\mathcal{K}$ that is a knitted-tree of P, $exec(\mathcal{K})$ is an execution π of P s.t. $\mathcal{K} \in kt(\pi, m, n)$.*

Notice that the relation between executions π and knitted-trees $\mathcal{K}$ defined by $\mathcal{K} \in kt(\pi, m, n)$ is neither injective nor functional. It is not functional due to the non-determinism in the encoding. It is not injective because a knitted-tree ending in a label overflow represents only a prefix of an execution and thus relates to all executions sharing that prefix.

Exit status of a knitted-tree. To distinguish how knitted-trees terminate, we introduce the notion of *exit status* for individual frames and for the entire knitted-tree. Each frame f of a knitted-tree is assigned one of five statuses in $ExStatus = \{\mathbf{C}, \mathbf{E}, \mathbf{O}, \mathbf{M}, \mathbf{N}\}$, with the following meanings:

- **C**lean exit: The program counter (pc) of f points to an **exit** instruction.
- Runtime **E**rror: $f.event = \textsc{err}$, indicating a null-pointer dereference.
- Label **O**verflow: The index of f in its label is $n + 1$, indicating log overflow.
- Out of **M**emory: $f.event = \textsc{oom}$, indicating a failed attempt to allocate a node with the **new** statement due to the absence of inactive nodes.
- **N**one: Indicates that frame f is not a terminal frame.

A frame f is *terminal* if its status is not **N**one. Indeed, the exit statuses different from **N** terminate the lace, hence only the last frame in the lace may have an exit status different from **N**. The *exit status* of a knitted-tree $\mathcal{K}$, denoted $exit(\mathcal{K})$, is the status of the last frame in its lace. The theorem below links knitted-tree's exit status to its corresponding executions.

Theorem 2. *For all executions π and $\mathcal{K} \in kt(\pi, m, n)$, the following hold:*

1. *If $exit(\mathcal{K}) = \mathbf{E}$, then π ends in an error configuration.*
2. *If π ends in an error configuration, then $exit(\mathcal{K}) \in \{\mathbf{E}, \mathbf{O}, \mathbf{M}\}$.*
3. *If π is infinite, then $exit(\mathcal{K}) = \mathbf{O}$.*

4 Properties of Knitted-Trees

Prefix of a knitted-tree. An (m, n)-knitted-tree is a knitted-tree associated with an execution π in $kt(\pi, m, n)$. A *prefix* of a knitted-tree $\mathcal{K}$ is a data tree obtained from $\mathcal{K}$ by truncating its lace at a frame f, setting $f'.avail$ to *true* for all subsequent frames, and leaving $f.next$ unconstrained.

Locality. For a label σ and $i \in \mathbb{N}$, we define $\sigma^{<i}$ (resp., $\sigma^{\leq i}$) as the label obtained by setting *avail* to true in all frames of σ with indices $\geq i$ (resp., $> i$).

The following lemma states that each new frame in a knitted-tree depends only on *local* information from neighboring nodes. We use $f_1 \equiv f_2$ to indicate that frames f_1 and f_2 differ only in their *next* field.

Lemma 1 (Locality). *There exist functions Up, Down, Internal such that for logs $\sigma, \tau_1, \ldots, \tau_{k+m}$ of a node u and of its children in a knitted-tree prefix:*

- *For all steps $(u.j, b) \rightarrow_{next} (u, a)$ in the lace, it holds $\sigma^a \equiv Up(\tau_j^{\leq b}, j, \sigma^{<a})$.*
- *For all steps $(u, a) \rightarrow_{next} (u.j, b)$ in the lace, it holds $\tau_j^b \equiv Down(\sigma^{\leq a}, j, \tau_j^{<b})$.*
- *For all steps $(u, a) \rightarrow_{next} (u, a+1)$ in the lace, it holds $\sigma^{a+1} \equiv Internal(\sigma^{\leq a})$.*

Compositionality. Using the functions *Up* and *Down* from Lemma 1, we define the predicate *consistent_child*(τ, j, σ). This predicate is meant to check whether two logs σ and τ may belong to the same knitted-tree as the logs of a node and its j^{th} child. Specifically, it verifies that all pairs of consecutive frames, linked by *next* and *prev*, with one frame belonging to τ and the other frame belonging to σ, adhere to the functions *Up* and *Down*. A detailed definition of *consistent_child* is in Appendix C.4.

From *consistent_child* and knitted-tree prefix definitions, the next lemma follows, ensuring that *consistent_child* holds on all parent-child log pairs.

Lemma 2. *For all labels $\sigma, \tau \in L(\mathcal{S}_\mathcal{K})$ and indices $j \in [k + m]$, if there exists a knitted-tree prefix where σ and τ are the logs of a node and its j^{th} child respectively, then consistent_child(τ, j, σ) holds.*

The following lemma establishes a key property of *consistent_child* for our verification approach, enabling knitted-tree composition from different subtrees.

Lemma 3 (Compositionality). *Let $\sigma_1, \sigma_2 \in L(\mathcal{S}_\mathcal{K})$ be the logs of nodes t_1, t_2 in (m, n)-knitted-tree prefixes $\mathcal{K}_1, \mathcal{K}_2$. If consistent_child$(\sigma_2, j, \sigma_1)$ holds true for some $j \in [k+m]$, then there is an (m, n)-knitted-tree prefix $\mathcal{K}$ where σ_1 is the log of a node and σ_2 is the log of its j^{th} child. Moreover, $\mathcal{K}$ is obtained by replacing the j^{th} subtree of t_1 in $\mathcal{K}_1$ with the subtree rooted at t_2 in $\mathcal{K}_2$.*

Proof. Let $\mathcal{K}$ be the data tree obtained by replacing the subtree rooted at the j^{th} child of t_1 in $\mathcal{K}_1$ with the subtree rooted at t_2 in $\mathcal{K}_2$. We prove that $\mathcal{K}$ is a knitted-tree prefix by induction on the number of pairs (a, b) where frames σ_1^a and σ_2^b are adjacent in the lace, each such pair representing an interaction between the parent's and the child's labels.

When the above number is zero, there are no interactions between t_1 and its j^{th} child in $\mathcal{K}_1$. Let π_1 be an execution s.t. $\mathcal{K}_1$ is a prefix of a knitted-tree representing π_1. It is direct to show that $\mathcal{K}$ is a knitted-tree prefix for an execution π following the same steps as π_1, starting with the input tree of $\mathcal{K}$. Since π_1 never visits the j^{th} child of t_1, and this subtree is the only difference between $\mathcal{K}_1$ and $\mathcal{K}$, π is a valid execution of P.

For the inductive case, consider the last interaction (a, b) between σ_1^a and σ_2^b. First, assume that such interaction is a step *up* from frame σ_2^b to σ_1^a. Let $\mathcal{K}_1'$ be derived from $\mathcal{K}_1$ by truncating its lace to end just before σ_1^a. Clearly, $\mathcal{K}_1'$ is still a knitted-tree prefix, and the modified label σ_1' of t_1 is obtained from σ_1 by removing the frames with index at least a, by setting their *avail* flags to true. We can now apply the inductive hypothesis to the labels σ_1' and σ_2, because we have removed one interaction between them. Hence, we can assume that there is a single knitted-tree prefix $\mathcal{K}'$ containing both labels as the logs of a parent and its j^{th} child. We then obtain the desired knitted-tree prefix $\mathcal{K}$ from $\mathcal{K}'$ by reintroducing the sequence of frames removed from $\mathcal{K}_1$. We need to prove that adding those frames respects all rules of knitted-trees. The correctness of the first added frame, σ_1^a, is ensured by $\mathit{consistent\_child}(\sigma_2, j, \sigma_1)$, because it applies the function *Up* to all upward interactions between σ_2 and σ_1. In turn, Lemma 1 ensures that adhering to that function is sufficient to establish the correctness of the next frame. Subsequent frames can be reintroduced due to the unchanged surroundings. Lemma 1 ensures that no other information is relevant.

The other case to prove is when the last interaction is a step *down* from the parent's frame σ_1^a to the child's frame σ_2^b. Define σ_2' as the label obtained from σ_2 with the frames of indices b and above removed. Similar to the previous case, we apply the inductive hypothesis to the shortened label σ_2' and its shortened knitted-tree prefix $\mathcal{K}_2'$, resulting in a knitted-tree prefix $\mathcal{K}'$. We then reintroduce the frames removed from $\mathcal{K}_2$ into $\mathcal{K}'$. The correctness of the first reintroduced frame is ensured by *consistent_child* checking the function *Down* on every downward interaction. The subsequent reintroduced frames are still valid because there are no steps returning to the parent of t_2, and their surroundings remain unchanged from $\mathcal{K}_2$. □

Example 1. Consider the knitted-tree in Fig. 2, with node labels $\sigma_1, \ldots, \sigma_5$, and another knitted-tree of the same program on the input list $7 \to 8 \to 9 \to 3 \to 10 \to 11 \to 12$ and $\mathit{key} = 3$. Let v_4 be the node of the second knitted-tree with value 3, and let τ_4 be its label. By inspecting the second knitted-tree, one can observe that the occupied frames of τ_4 (i.e., those with $\mathit{avail} = \mathit{false}$) contain the same information as the occupied frames of σ_3. Therefore, $\mathit{consistent\_child}(\tau_4, 1, \sigma_2)$ holds, and by Lemma 3, the two knitted-trees can be composed at nodes u_2-v_4 to construct a knitted-tree for the input list $1 \to 2 \to 3 \to 10 \to 11 \to 12$.

(I) $\mathbf{Lab}(\sigma) \leftarrow \mathit{len}(\sigma, 1) \wedge \mathit{first\_frame}(\sigma^1)$ *Initializing non-root nodes*

(II) $\mathbf{Lab}(\sigma) \leftarrow \mathit{len}(\sigma, 2) \wedge \mathit{start}(\sigma)$ *Initializing the root node*

(III) $\mathbf{Lab}(\sigma) \leftarrow \mathit{len}(\sigma, i) \wedge \mathbf{Lab}(\sigma^{<i}) \wedge \Psi_{Internal}(\sigma^{<i}, \sigma^i)$ *Internal step*

A step from the j^{th} child to its parent

(IV) $\mathbf{Lab}(\sigma) \leftarrow \mathit{len}(\sigma, i) \wedge \mathbf{Lab}(\sigma^{<i}) \wedge \mathbf{Lab}(\tau)$
$\wedge\ \mathit{consistent\_child}(\tau, j, \sigma^{<i}) \wedge \Psi_{Up}(\tau, j, \sigma^{<i}, \sigma^i)$

A step from a node to its j^{th} child

(V) $\mathbf{Lab}(\tau) \leftarrow \mathit{len}(\tau, i) \wedge \mathbf{Lab}(\sigma) \wedge \mathbf{Lab}(\tau^{<i})$
$\wedge\ \mathit{consistent\_child}(\tau^{<i}, j, \sigma) \wedge \Psi_{Down}(\sigma, j, \tau^{<i}, \tau^i)$

(VI) $\perp \leftarrow \mathbf{Lab}(\sigma) \wedge \mathit{label\_exit}(\sigma, Ex)$ *Lace ends with status in Ex*

Fig. 3: CHCs (I)-(V) form the CHC system $\mathcal{C}_{\mathrm{kt}}(P, m, n)$, while the CHC system $\mathcal{C}_{\mathrm{ex}}(\mathcal{I})$ includes all the CHCs in the figure. Here, $i \in [2, n]$ and $j \in [k + m]$.

5 Reasoning about Knitted-Trees with CHCs

We introduce a CHC system $\mathcal{C}_{\mathrm{kt}}(P, m, n)$ for a program P with parameters $m, n \in \mathbb{N}$. It employs a single uninterpreted relation symbol, $\mathbf{Lab}(\sigma)$, where σ matches the data signature $\mathcal{S}_{\mathcal{K}}$ of knitted-trees, ensuring the following:

Theorem 3. *In the minimal model of $\mathcal{C}_{\mathrm{kt}}(P, m, n)$, $\mathbf{Lab}(\sigma)$ holds for a label σ iff σ labels a node in some (m, n)-knitted-tree prefix $\mathcal{K}$ of P.*

We define $\mathcal{C}_{\mathrm{kt}}(P, m, n)$ using the knitted-tree rules constructing constructing partial knitted-trees (impractical for enumerating all knitted-trees). Instead, we rely on the compositionality lemma (Lemma 3), which states that two consistent labels imply the existence of a knitted-tree where those labels are logs of an internal node and one of its children, and the locality lemma (Lemma 1) to extend the lace involving these nodes. This lemma entails that constructing a knitted-tree involves adding frames to node logs so that any two consecutive frames belong to the same node or to neighboring backbone nodes. We use this property in the CHC system, employing independent CHCs to simulate adding a single frame. In the CHC system, we simulate adding a single frame via *Up* (upward), *Down* (downward), and *Internal* (within the same log). We define predicates $\Psi_{Down}(\sigma, j, \tau_j, f), \Psi_{Up}(\tau_j, j, \sigma, f)$, and $\Psi_{Internal}(\sigma, f)$ to constrain logs accordingly, with f as the resulting frame.

Figure 3 details the CHCs of $\mathcal{C}_{\mathrm{kt}}(P, m, n)$. While describing each CHC, we establish the "only if" direction of Theorem 3, by induction on the number of CHC applications needed to insert σ into the minimal interpretation of $\mathbf{Lab}$. For completeness, the proof of the "if" direction appears in Appendix B.3.

Before detailing the CHCs, we introduce some notation. Let $\sigma \in L(\mathcal{S}_{\mathcal{K}})$ and $i \in \mathbb{N}$. The formula $len(\sigma, i)$ is true if all frames with indices in $[i]$ are unavailable and all other frames are available, i.e., $len(\sigma, i) \overset{\text{def}}{=} \neg\sigma^i.avail \wedge \bigwedge_{j=i+1}^{n+1} \sigma^j.avail$. With an abuse of notation, we write $\sigma^{<i}$ in a CHC as a shorthand for a fresh variable θ, together with the conjunct $\bigwedge_{\ell\in[i-1]}(\theta^\ell = \sigma^\ell) \wedge \bigwedge_{\ell\in[i,n+1]} \theta^\ell.avail$.

CHCs I and **II** ensure that **Lab** includes labels of all knitted-trees of 0-length executions of P, forming the base case for the "only if" direction of Theorem 3, since both are facts. CHC I defines non-root labels with all but the first frame available, and consistent *active*/*active_child*, while CHC II defines root labels by disabling the first two frames and using $start(\sigma)$ for the second frame, as per the base-case labeling (Section 3.1).

The remaining CHCs extend each node's log frame by frame, following the inductive knitted-tree label definition. For Theorem 3 ("only if" direction), we assume inductively that the body labels satisfy the claim, i.e., they label a node in a (m, n)-knitted-tree prefix, and show that the head label does too.

CHC III handles *internal steps*, where σ^i follows σ^{i-1} in the lace. Here, $\mathbf{Lab}(\sigma^{<i})$ ensures that $\sigma^{<i}$ labels a node in an (m, n)-knitted-tree prefix, while $\Psi_{Internal}(\sigma^{<i}, \sigma^i)$ constrains σ^i to encode the next internal step, as per Lemma 1.

CHC IV handles cases where the lace extends with a new frame pushed to the parent of the previous frame, typically during a rewind phase. The predicate *consistent_child* ensures that $\sigma^{<i}$ and τ belong to the same knitted-tree prefix, as the log of a parent and its jth child (Lemma 3). Then, Ψ_{Up} extends the lace by adding a frame to $\sigma^{<i}$, following the topmost frame of τ.

CHC V handles the reverse of CHC IV, where the current lace extends from a parent to its jth child. Using Ψ_{Down}, it ensures that τ_j correctly extends $\tau_j^{<i}$ with a new frame for the latest step.

The Exit Status Problem. We present a method to check whether a program can lead to a memory safety error via an execution that can be represented by an (m, n)-knitted-tree. This reduces to solving a CHC system: if unsatisfiable, such an execution exists. We formalize this as the following decision problem.

Problem 2. An instance of the **exit status problem** is a tuple (P, m, n, Ex), where P is a program, $m, n \in \mathbb{N}$, and Ex is a set of exit statuses excluding $\mathbf{N}$. The exit status problem asks whether there exists an (m, n)-knitted-tree of P whose exit status is in Ex.

We solve the exit status problem using the CHC system $\mathcal{C}_{\text{ex}}(\mathcal{I})$ (Figure 3), which includes (i) all CHCs from $\mathcal{C}_{\text{kt}}\big(P, m, n\big)$, crucial for Theorem 3, and (ii) a single query, CHC VI, to check for a knitted-tree corresponding to a program execution with an exit status in Ex. Combining Theorem 3 with the definition of CHC VI yields the main result of this section.

Theorem 4. *Let $\mathcal{I}$ be an instance of the exit-status problem. Then, $\mathcal{I}$ admits a positive answer if and only if the CHC system $\mathcal{C}_{\text{ex}}(\mathcal{I})$ is unsatisfiable.*

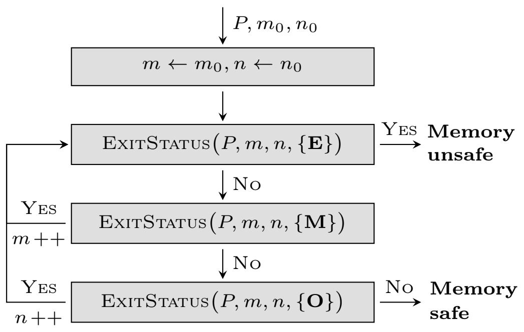
5.1 Verifying Memory Safety

We begin by establishing two key theorems linking the exit status problem to memory safety, forming the basis for our method's correctness.

Theorem 5. *If the answer to the exit status problem* $(P, m, n, \{\mathbf{E}\})$ *is positive, then the answer to the memory safety for* P *is negative.*

Theorem 6. *If the answer to the exit status problem* $(P, m, n, \{\mathbf{O}, \mathbf{M}, \mathbf{E}\})$ *is negative, then the answer to the memory safety problem for* P *is positive.*

Algorithm MemSafety: We outline our algorithm on the right. We are given a program P and initial values m_0 and n_0 for the two parameters m and n of knitted-trees. Verification starts by solving the problem $\textsc{ExitStatus}\big(P, m, n, \{\mathbf{E}\}\big)$ to detect null-pointer dereference errors. If the answer is positive, by Theorem 5 P violates memory safety. Otherwise, memory safety is not guaranteed, as the current values of m and n may not cover all executions. To address this, we solve $\textsc{ExitStatus}\big(P, m, n, X\big)$ with:


P, m_0, n_0
$m \leftarrow m_0, n \leftarrow n_0$
$\textsc{ExitStatus}(P, m, n, \{\mathbf{E}\})$
Yes
Memory unsafe
No
Yes
m++
$\textsc{ExitStatus}(P, m, n, \{\mathbf{M}\})$
No
Yes
n++
$\textsc{ExitStatus}(P, m, n, \{\mathbf{O}\})$
No
Memory safe

1. $X = \{\mathbf{M}\}$ to detect out-of-memory failure from **new** allocations, and
2. $X = \{\mathbf{O}\}$ to detect label overflow errors.

If both instances are negative, Theorem 6 ensures that P is memory-safe. Otherwise, we increment m or n to broaden coverage and restart. This may continue indefinitely if the semi-algorithm never terminates (due to undecidability) or no parameter values suffice to establish memory safety.

Theorem 7. *Algorithm* MemSafety *is a sound solution to the memory safety problem, i.e., if it terminates, it yields the correct answer.*

5.2 Invariant Structure

In this section, we examine the essential properties that a solution to the CHC system presented in Fig. 3 must satisfy, with particular emphasis on the structure and complexity of the required invariants. To ground the discussion, we refer to the running example introduced in Section 1 and shown in Fig. 1, representative of a broad class of procedures manipulating tree data structures. This example highlights both the challenges and the recurring structural patterns encountered in the synthesis of suitable invariants for CHC systems.

The minimal model of **Lab** for logs of knitted trees (from lists containing the *key* value) results in labels that can be classified as follows:

Initial Node: the first node of the lists is shaped as u_1 (see Fig. 2).
Intermediate Nodes: one or more nodes like u_2 (depending on the length of the input list).
Target Node: the node containing the special value, u_3.
Post-Target Node: the node immediately following the special value, u_4.
Subsequent Nodes: remaining nodes, similar to u_5.

Labels associated with nodes of the same category differ only in their data fields (*val* and *key*), while all other fields (e.g., program counters) are equal across nodes (for instance, nodes like u_2 consistently exhibit program counter 3 in the second frame). Only the data fields *val* and *key* depend on the input list, and their constraints are simple:

- In each node, both *val* and *key* retain the same value across all frames.
- In an intermediate node, $val \neq key$;
- In the target node (u_3), $val = key$.

Input lists that do not contain the *key* induce additional node types, subject to analogous constraints. In Section 7, we briefly outline how the regularities discussed above could be exploited in an implementation of our framework.

6 Related Work

Our work is related to many works in the literature in different ways. Here we focus on those that seem to be the closest to the results presented in this paper.

Our approach uses CHC engines for backend analysis, similar to other verification methods [12,25,27,29,31,37,38,49,28,23,21,33,24]. CHCs serve as an intermediate language, allowing us to focus on proof rules while solvers implement algorithms within a standard framework. A primary challenge is encoding heap-allocated mutable data structures. While array theory is often used (e.g., [39,15]), it can result in complex CHCs. Our approach uses simple theories for basic data types, avoiding array theory unless necessary. Traditional heap program analysis often relies on abstractions like shape analysis [58] to scale. Refinement types and invariants can be used to transform complex data structures, avoiding array theory (e.g. [57,10,50,36]). This can lead to over-approximation in CHCs and false positive by replacing heap operations with local object assertions, potentially missing global invariants but enabling efficient verification when local invariants suffice. A recent proposal suggests using an SMT-LIB theory of heaps for CHCs to standardize heap data-structure representation [20].

Our technique relates to tree automata, automata with auxiliary storage, and bounded tree-width graphs representing their executions. It also relates to Courcelle's theorem (proof), which reduces analysis to tree automata emptiness [26]. Inspired by Alur and Madhusudan's nested words to represent pushdown automata executions [2], and their extensions for multistack and distributed automata by Madhusudan and Parlato [43], we represent tree-manipulating program executions as knitted-trees, where nodes are frames and edges are *next* and *prev* frame fields. Similar to La Torre et al. [40] for multistack pushdown automata, our approach provides tree decompositions with a bounded tree width. Instead of using tree automata emptiness for reachability analysis, we leverage CHC solvers to enable a tree automata-like method with enhanced data reasoning. Additionally, like Heizmann et al. [30], we use automata for program analysis but replace counterexample-guided abstraction refinement with precise knitted-tree representations and CHC solvers for approximation and refinement.

Our work extends decidable methods for bounded-pass heap-manipulating programs by supporting a broader range of properties, potentially at the cost of undecidability. Mathur et al. [48] achieve decidable memory safety for forest-like initial heaps and single-pass traversals, building on uninterpreted coherent programs [46,47]. They handle memory freeing but leave support for more complex postconditions for future work. Alur and Černý [1] reduce assertion checking of single-pass list-processing programs to data string transducers, achieving decidability with a single advancing variable. This approach is less flexible than Mathur et al.'s, as it doesn't address memory safety or heap shape changes and is limited to data ordering and equality without handling explicit memory freeing.

Heap verification has been extensively studied using decidable logics such as first-order logic with reachability [42], LISBQ in the HAVOC tool [41], and fragments of separation logic [7,54,18]. Some approaches interpret bounded tree width data structures on trees [32,44]. While these logics are often restrictive, others methods handle undecidable cases using heuristics, lemma synthesis, and programmer annotations [4,6,5,9,8,11,13,14,17,34,51,52,53,55,59]. In contrast, our knitted-tree encoding promotes a separation of concerns, offloading the algorithmic burden to the underlying CHC solver.

7 Conclusions and Directions for Future Research

We presented a foundational, compositional approach to verifying programs that manipulate tree data structures. By modeling executions as knitted-trees and encoding them as CHCs, verification reduces to CHC satisfiability. This enables modular reasoning and supports simple invariants. Overall, our method offers a uniform and scalable framework for automating the verification of a broad class of tree-manipulating programs.

Future Work. Our approach opens multiple research directions.

Efficient Implementation. Labels are currently handled by a single predicate, **Lab**. Performance can be improved via case splitting—encoding enumerated fields into predicate names—to simplify invariants (see Section 5.2). Moreover, precomputing all possible configurations of the enumerated fields arising in every possible frame of the program, together with (an overapproximation of) the within-node and across-neighbor adjacency relations between frames would produce a larger set of significantly simplified CHCs that enforce consistency of the data with the enumerated structure of the knitted tree.

Beyond Memory Safety. Full correctness requires verifying structural and functional properties. Using *symbolic data-tree automata* (SDTAS), which integrate well with CHC-based verification [22], we can formally specify pre/postconditions–e.g., that inputs form a red-black tree and outputs a sorted list. Preconditions are easy to encode as they involve only the first frame of each node in the knitted-tree, while postconditions require more effort due to the complexity of encoding the output heap within the knitted-tree structure. For set-like trees, it is also important to verify that operations such as insertion, deletion, and search preserve key invariants, which can be checked via knitted-tree logs. Termination can be verified by ensuring that labels do not overflow.

Deductive Verification. Our methodology is particularly suited for deductive verification of procedures with linear-time complexity—i.e., those that traverse each node a bounded number of times. We aim to develop a verification framework where program correctness is established by breaking down verification conditions into preconditions and postconditions for code segments, with each segment provably executable in linear time. This would bridge the gap between our approach and classical deductive verification techniques.

More Structures. Our approach naturally extends to programs manipulating multiple data structures, especially those with bounded treewidth. While there is no general method for all combinations, many can be handled with suitable encodings. For example, a program that traverses a red-black tree in-order and inserts values into a singly linked list can be modeled using knitted trees with bounded labels; our method can then verify that the output list contains the input values in non-increasing order. Some scenarios require more inventive encodings. For instance, checking equality of two lists via separate dummy roots leads to unbounded log growth. Instead, modeling both as a single list of paired elements keeps the log size bounded and tractable.

As noted in the previous section, the graphs induced by knitted-trees have bounded treewidth, suggesting applicability to a broad range of structures, including arrays, doubly-linked lists, trees with parent pointers, and, more generally, any structure with bounded treewidth and a canonical tree decomposition.

Program Synthesis. We also plan to explore syntax-guided synthesis (SyGuS) of tree-manipulating code. By expressing correctness properties as SDTAs and reducing synthesis to CHC solving, we aim to generate correct-by-construction procedures, extending extending recent work on synthesis from specifications [56].

Acknowledgments. We sincerely thank the anonymous reviewers for their insightful feedback and constructive suggestions, which significantly improved the quality of this paper. This work was partially supported by INdAM-GNCS Project 2025 - CUP: E53C24001950001, and the MUR projects SOP (Securing sOftware Platforms, CUP: H73C22000890001) as part of the SERICS project (n. PE00000014 - CUP: B43C22000750006), and MUR project FAIR (Future AI Research, CUP: E63C220 02150007).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alur, R., Cerný, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 599–610. ACM (2011). `https://doi.org/10.1145/1926385.1926454`
2. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM **56**(3), 16:1–16:43 (2009). `https://doi.org/10.1145/1516512.1516518`
3. Angelis, E.D., K, H.G.V.: CHC-COMP 2022: Competition report. Electronic Proceedings in Theoretical Computer Science **373**, 44–62 (nov 2022). `https://doi.org/10.4204/eptcs.373.5`
4. Banerjee, A., Barnett, M., Naumann, D.A.: Boogie meets regions: A verification experience report. In: Shankar, N., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5295, pp. 177–191. Springer (2008). `https://doi.org/10.1007/978-3-540-87873-5_16`
5. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5142, pp. 387–411. Springer (2008). `https://doi.org/10.1007/978-3-540-70592-5_17`
6. Banerjee, A., Naumann, D.A., Rosenberg, S.: Local reasoning for global invariants, part I: region logic. J. ACM **60**(3), 18:1–18:56 (2013). `https://doi.org/10.1145/2485982`
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3328, pp. 97–109. Springer (2004). `https://doi.org/10.1007/978-3-540-30538-5_9`
8. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 115–137. Springer (2005). `https://doi.org/10.1007/11804192_6`
9. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3780, pp. 52–68. Springer (2005). `https://doi.org/10.1007/11575467_5`
10. Bjørner, N.S., McMillan, K.L., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7935, pp. 105–125. Springer (2013). `https://doi.org/10.1007/978-3-642-38856-9_8`
11. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011). `https://doi.org/10.1145/2049697.2049700`

12. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: Ice-based refinement type discovery for higher-order functional programs. J. Autom. Reason. **64**(7), 1393–1418 (2020)
13. Chin, W., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**(9), 1006–1036 (2012). https://doi.org/10.1016/J.SCICO.2010.07.004
14. Chu, D., Jaffar, J., Trinh, M.: Automatic induction proofs of data-structures in imperative programs. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 457–466. ACM (2015). https://doi.org/10.1145/2737924.2737984
15. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Program verification using constraint handling rules and array constraint generalizations. Fundam. Informaticae **150**(1), 73–117 (2017). https://doi.org/10.3233/FI-2017-1461
16. De Angelis, E., K., H.G.V.: CHC-COMP 2023: Competition report. CoRR **abs/2404.14923** (2024). https://doi.org/10.48550/ARXIV.2404.14923
17. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3920, pp. 287–302. Springer (2006). https://doi.org/10.1007/11691372_19
18. Echenim, M., Iosif, R., Peltier, N.: Unifying decidable entailments in separation logic with inductive definitions. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12699, pp. 183–199. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_11
19. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. J. ACM **23**(4), 733–742 (1976)
20. Esen, Z., Rümmer, P.: An SMT-LIB theory of heaps. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022. CEUR Workshop Proceedings, vol. 3185, pp. 38–53. CEUR-WS.org (2022)
21. Esen, Z., Rümmer, P.: Tricera: Verifying C programs using the theory of heaps. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 380–391. IEEE (2022). https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_45
22. Faella, M., Parlato, G.: Reasoning about data trees using chcs. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13372, pp. 249–271. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_13
23. Faella, M., Parlato, G.: Reachability games modulo theories with a bounded safety player. In: Williams, B., Chen, Y., Neville, J. (eds.) Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on In-

novative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023. pp. 6330–6337. AAAI Press (2023). https://doi.org/10.1609/AAAI.V37I5.25779

24. Faella, M., Parlato, G.: A unified automata-theoretic approach to $\mathrm{ltl}_{\mathrm{f}}$ modulo theories. In: Endriss, U., Melo, F.S., Bach, K., Diz, A.J.B., Alonso-Moral, J.M., Barro, S., Heintz, F. (eds.) ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024). Frontiers in Artificial Intelligence and Applications, vol. 392, pp. 1254–1261. IOS Press (2024). https://doi.org/10.3233/FAIA240622
25. Fedyukovich, G., Ahmad, M.B.S., Bodík, R.: Gradual synthesis for static parallelization of single-pass array-processing programs. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 572–585. ACM (2017)
26. Flum, J., Grohe, M.: Parameterized Complexity Theory. Texts in Theoretical Computer Science. An EATCS Series, Springer (2006). https://doi.org/10.1007/3-540-29953-X
27. Garoche, P., Kahsai, T., Thirioux, X.: Hierarchical state machines as modular horn clauses. In: Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016. EPTCS, vol. 219, pp. 15–28 (2016)
28. Gurfinkel, A.: Program verification with constrained horn clauses (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 19–29. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_2
29. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. LNCS, vol. 9206, pp. 343–361. Springer (2015)
30. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 36–52. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
31. Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems - tool paper. In: FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. LNCS, vol. 7436, pp. 247–251. Springer (2012)
32. Iosif, R., Rogalewicz, A., Simácek, J.: The tree width of separation logic with recursive definitions. In: Bonacina, M.P. (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7898, pp. 21–38. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_2
33. Itzhaky, S., Shoham, S., Vizel, Y.: Hyperproperty verification as CHC satisfiability. In: Weirich, S. (ed.) Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II. Lecture Notes

in Computer Science, vol. 14577, pp. 212–241. Springer (2024). https://doi.org/10.1007/978-3-031-57267-8_9
34. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
35. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program. **19/20**, 503–581 (1994)
36. Kahsai, T., Kersten, R., Rümmer, P., Schäf, M.: Quantified heap invariants for object-oriented programs. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 368–384. EasyChair (2017). https://doi.org/10.29007/ZRCT
37. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: A framework for verifying java programs. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. LNCS, vol. 9779, pp. 352–358. Springer (2016)
38. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 222–233. ACM (2011)
39. Komuravelli, A., Bjørner, N.S., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015. pp. 89–96. IEEE (2015). https://doi.org/10.1109/FMCAD.2015.7542257
40. La Torre, S., Napoli, M., Parlato, G.: A unifying approach for multistack pushdown automata. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8634, pp. 377–389. Springer (2014). https://doi.org/10.1007/978-3-662-44522-8_32
41. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 171–182. ACM (2008). https://doi.org/10.1145/1328438.1328461
42. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. Log. Methods Comput. Sci. **5**(2) (2009)
43. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 283–294. ACM (2011). https://doi.org/10.1145/1926385.1926419
44. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 611–622. ACM (2011). https://doi.org/10.1145/1926385.1926455

45. Manna, Z., Zarba, C.G.: Combining decision procedures. In: Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers. LNCS, vol. 2757, pp. 381–422. Springer (2002)
46. Mathur, U., Madhusudan, P., Viswanathan, M.: Decidable verification of uninterpreted programs. Proc. ACM Program. Lang. **3**(POPL), 46:1–46:29 (2019). `https://doi.org/10.1145/3290359`
47. Mathur, U., Madhusudan, P., Viswanathan, M.: What's decidable about program verification modulo axioms? In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 158–177. Springer (2020). `https://doi.org/10.1007/978-3-030-45237-7_10`
48. Mathur, U., Murali, A., Krogmeier, P., Madhusudan, P., Viswanathan, M.: Deciding memory safety for single-pass heap-manipulating programs. Proc. ACM Program. Lang. **4**(POPL), 35:1–35:29 (2020). `https://doi.org/10.1145/3371103`
49. Matsushita, Y., Tsukada, T., Kobayashi, N.: Rusthorn: Chc-based verification for rust programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021). `https://doi.org/10.1145/3462205`
50. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free horn clauses. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 361–382. Springer (2016). `https://doi.org/10.1007/978-3-662-53413-7_18`
51. Murali, A., Peña, L., Blanchard, E., Löding, C., Madhusudan, P.: Model-guided synthesis of inductive lemmas for FOL with least fixpoints. Proc. ACM Program. Lang. **6**(OOPSLA2), 1873–1902 (2022). `https://doi.org/10.1145/3563354`
52. Nguyen, H.H., Chin, W.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5123, pp. 355–369. Springer (2008). `https://doi.org/10.1007/978-3-540-70545-1_34`
53. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in C using separation logic. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 440–451. ACM (2014). `https://doi.org/10.1145/2594291.2594325`
54. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 773–789. Springer (2013). `https://doi.org/10.1007/978-3-642-39799-8_54`
55. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 711–728. Springer (2014). `https://doi.org/10.1007/978-3-319-08867-9_47`

56. Polikarpova, N., Sergey, I.: Structuring the synthesis of heap-manipulating programs. Proc. ACM Program. Lang. **3**(POPL), 72:1–72:30 (2019). https://doi.org/10.1145/3290385
57. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 159–169. ACM (2008). https://doi.org/10.1145/1375581.1375602
58. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Appel, A.W., Aiken, A. (eds.) POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999. pp. 105–118. ACM (1999). https://doi.org/10.1145/292540.292552
59. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated mutual explicit induction proof in separation logic. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 659–676 (2016). https://doi.org/10.1007/978-3-319-48989-6_40

A Encoding Restricted Recursion Mechanisms in the Programming Language

In Section 2, we introduced a simple programming language for heap-manipulating programs that does not directly support procedure calls. For non-recursive procedures, we can handle calls by inlining their code at each call site. In this section, we describe an approach to simulating a specific form of recursive procedures, commonly used in procedures that manipulate tree data structures, as found in the literature.

We focus on procedures where the first parameter (later denoted by *first*) is a pointer. The corresponding argument that is passed at runtime must satisfy the following conditions:

1. It references a node that has not been previously used for invocation.
2. It corresponds to a pointer field of the current node (i.e., the node on which the procedure was last invoked).
3. Its value is unchanged from its original value at the beginning of the program.

The remaining parameters can be any data values or pointer variables. Additionally, if the procedure is invoked with a **nil** pointer as the first parameter, it must not call another procedure before returning. The code handling this case must be guarded by an **if** statement that checks whether the first argument of the function is equal to or different from **nil**. These constraints are typically satisfied by functions that manipulate Binary Search Trees (BSTs), red-black trees, AVL trees, and similar structures.

We assume that procedures do not return data or pointer values; instead, any return values are managed via global variables. We also assume that the **main** (i.e., outmost) procedure is never called.

Under these constraints, recursive programs can be transformed into a program without procedures by extending the input tree's structure to include additional fields for local data and pointer variables used within the procedures. We also include additional fields to orchestrate the simulation of the control flow during calls and returns, thereby eliminating the need for a system stack. To formalize this approach, we define an extended heap as follows:

Definition 3. *An* **extended heap** *is a tuple* $\mathcal{H}_{ext} = (N, \mathcal{S}, \mathbf{data}, PF, \mathcal{S}_{ext}, \mathbf{data}_{ext}, PF_{ext})$ *where* $(N, \mathcal{S}, \mathbf{data}, PF)$ *and* $(N, \mathcal{S}_{ext}, \mathbf{data}_{ext}, PF_{ext})$ *are heaps.*

The extended data signature $\mathcal{S}_{ext}$ includes:

Local Variable Fields: Fields for the distinct local variables of the procedures (including the procedure arguments). The types of these fields match the types of the corresponding local variables.

Program Counter: A field named *return_lab* to keep track of the next statement to execute within a procedure after returning from the last ongoing procedure call.

Parent Field: A pointer field that points to the parent node in the input tree, facilitating the return control. This field is initialized accordingly and is never modified in the translated code.

A.1 Code to Code Translation

Here, we outline a method to rewrite the code and entirely eliminate recursive procedures, resulting in a program that behaves equivalently to the original.

Initialize the local variables of the root: We add a fresh assignment to set the local variables related the main procedure to point at $\widehat{p}$. This establishes the property that the first argument of the procedure refers to the node on which it is invoked. For other nodes, this assignment is performed before the simulation starts.

Local variable replacement: Within a procedure's body, replace all occurrences of local variables as follows. If loc is a local variable, replace it with $first \rightarrow loc$. This modification is applied throughout the procedure code except for the code block executed when the procedure is called with **nil** as the first parameter. This ensures that local variables correctly reference the appropriate heap node.

Consolidate into a single procedure: Combine all existing functions into a single procedure by removing individual function declarations. We assume that all statement labels, variable names, and function parameter names are distinct to avoid naming conflicts.

Replace procedure calls: This process is divided into four phases. Consider a procedure call statement, and assume that the first parameter of the calling procedure is $first$, whereas the arguments of this call are $arg_1, \ldots, arg_\ell$. Recall that, according to Condition 2 at the beginning of this section, $arg_1 = first \rightarrow pfield$, for some pointer field $pfield$. We then perform the following steps, that add new statements in place of the procedure call.

Phase 1 – Argument Assignment: Assign the values of the arguments of the called procedure to the corresponding fields of the node referenced by arg_1. Specifically, let $param_1, \ldots, param_\ell$ be the list of parameters of the callee, for each $i \in [\ell]$, add an assignment of the form:

$$arg_1 \rightarrow param_i := arg_i.$$

Phase 2 – Set Return Label Assignment: Assign the label of the statement immediately following the current function call to $arg_1 \rightarrow return\_lab$. This sets up the point to which the program should jump after the simulated procedure execution corresponding to the current call:

$$arg_1 \rightarrow return\_lab := \langle \text{label of the next statement in the orginal code} \rangle$$

Phase 3 – Jump to the First Statement of the Called Procedure: Transfer control in the monolithic program to the first statement of the called procedure via a **goto** statement:

$$\textbf{goto}\ \langle \text{label of the first statement of the called function} \rangle.$$

Phase 4 – Jump Back to the Caller Procedure: Replace **return** statements in the original code with a **goto** statement that uses the label stored in the auxiliary field of the current node set at the time of procedure invocation, namely:

$$\textbf{goto } \mathit{first} \to \mathit{return\_lab}.$$

This systematic transformation effectively simulates recursive procedure calls without using actual function calls or a system stack. By extending the heap with additional fields to store local variables, return addresses, and parent pointers, the program maintains the necessary state to manage control flow and variable references during execution.

B Additional Proofs

B.1 Proofs for Section 3

Theorem 1. *Given a program P and parameters m, n, $kt(\cdot, m, n)$ is computable. Moreover, there is a computable function exec such that, for any data tree $\mathcal{K}$ that is a knitted-tree of P, $exec(\mathcal{K})$ is an execution π of P s.t. $\mathcal{K} \in kt(\pi, m, n)$.*

Proof. The first part of the statement (i.e., the computability of *kt*) can be shown by inspecting the encoding described in Section 3, with further details in Appendix C. One can clearly see that the definition of knitted-tree is entirely constructive. In fact, the content of each new frame in the lace can be computed from the program and the log of the current node and of the neighboring nodes.

The second part of the statement requires a function *exec* that maps knitted-trees back to executions. To this aim, we first prove two supporting claims.

Claim 1: We can retrieve the state of the heap from any knitted-tree prefix. We show that we can recover the state of the heap at each point of the lace of a knitted-tree. We select the nodes where *active* is *true* in their top frame. We then examine the logs of the selected nodes to determine the values of their fields. For any node u, the current value of its data field *val* is the value of the *val* field in the topmost frame of the log. The current value of a pointer field *pfield* (assumed to be the jth field in *PF*) can be recovered as follows:

– If *pfield* has never been updated, $u.\mathit{pfield}$ points to the jth child of u, if $u.j$ exists and is active in the knitted-tree; otherwise, $u.\mathit{pfield}$ points to **nil**.
– If the last update in u's log is $\langle \mathit{pfield} \coloneqq \textbf{nil} \rangle$, then that pointer field is also **nil**.
– Otherwise, the last update is $\langle \mathit{pfield} \coloneqq p \rangle$ for some pointer variable p. We navigate the lace backward using *find_ptr* until finding a frame that reports $\langle p \coloneqq \textbf{here} \rangle$, say in the log of node v, and set $u.\mathit{pfield}$ to v.

We denote by $heap(\mathcal{K})$ the heap corresponding to the knitted-tree (prefix) $\mathcal{K}$, as described above.

Claim 2: We can retrieve the value of all pointer variables from any knitted-tree prefix. The *isnil* fields in the last frame of the knitted-tree prefix indicate which pointer variables are **nil**. We can find the current value of the other pointer variables $p \in PV_P$, by retracing the lace backward until the most recent assignment to it (i.e., $\langle p \coloneqq \textbf{here} \rangle$). We denote by $pv(\mathcal{K})$ the map that assigns each $p \in PV_P$ to the node of $heap(\mathcal{K})$ pointed to by p at the end of the lace of $\mathcal{K}$.

We can now prove the second part of the statement. Let $\mathcal{K} = (K, \mu)$ be a data tree that is the knitted-tree of some execution. To define $exec(\mathcal{K})$, we start by recovering the initial program configuration $c_0 = (\mathcal{H}_0, \nu_p, \nu_d, pc_0)$, where the initial heap $\mathcal{H}_0$ can be obtained from Claim 1 above, and the initial value of the pointer variables ν_p from Claim 2. The initial value of the program counter is simply the label of the first line in the program, and the initial value for the data variables ν_d can be read from the d fields in the second frame of the root of $\mathcal{K}$. To obtain each subsequent program configuration, it is sufficient to scan the lace chronologically, iteratively identify the frame f that marks the end of each statement[6], and at that point build a new program configuration using Claims 1 and 2 above, and reading the current value of the data variables and program counter from f. □

Theorem 2. *For all executions π and $\mathcal{K} \in kt(\pi, m, n)$, the following hold:*

1. *If $exit(\mathcal{K}) = \mathbf{E}$, then π ends in an error configuration.*
2. *If π ends in an error configuration, then $exit(\mathcal{K}) \in \{\mathbf{E}, \mathbf{O}, \mathbf{M}\}$.*
3. *If π is infinite, then $exit(\mathcal{K}) = \mathbf{O}$.*

Proof. **Claim 1.** To prove the first statement, assume that $\mathcal{K}$ has exit status $\mathbf{E}$, occurring when the last frame f in $\mathcal{K}$'s lace has the event ERR. By Thm 1, the last configuration of π can be recovered from $\mathcal{K}$ via the *exec* function. By definition of knitted-trees, the event ERR arises only from a null pointer dereference, so c is an error configuration.

Claim 2. For the second statement, assume that π ends in an error configuration and the exit status of $\mathcal{K}$ is neither $\mathbf{O}$ nor $\mathbf{M}$. We show that the exit status of $\mathcal{K}$ is $\mathbf{E}$. By Thm 1, the last configuration of π can be recovered from $\mathcal{K}$ via the *exec* function. Since π ends in an error configuration, the program counter in the last frame of the lace of $\mathcal{K}$ points to an instruction causing a null pointer dereference, leading to the event ERR and hence to the exit status $\mathbf{E}$.

Claim 3. Each step of an execution adds at least one frame to the lace of its knitted-tree. Since the total number of nodes and frames in a knitted-tree is finite, the lace will eventually occupy a frame with index $n+1$ in the log hosting that frame, resulting in the exit status $\mathbf{O}$. □

B.2 Proofs for Section 4

Lemma 2. *For all labels $\sigma, \tau \in L(\mathcal{S}_{\mathcal{K}})$ and indices $j \in [k+m]$, if there exists a knitted-tree prefix where σ and τ are the logs of a node and its j^{th} child respectively, then consistent_child(τ, j, σ) holds.*

[6] Recall that the execution of a statement may require multiple frames in the lace, especially if a rewind is involved.

Proof. We follow the definition of *consistent_child* presented in Appendix C.4. The *consistent_first_frames*(τ, j, σ) predicate holds true because it follows the definition of the *active* and *active_child* fields in the first frame of every node, as detailed in Sec. 3.1. The next two blocks in the definition of *consistent_child* are responsible for checking that every pair of frames connected by the *prev* and *next* fields and belonging to different labels encodes a step in the current instruction, according to the rules detailed in §3.1. The two connected frames may represent a step downward (first block) or upward (second block) in the knitted-tree.

Consider the downward direction, as the other case is symmetrical. A downward connection between the frames σ^a and τ^b is detected in two cases:

1. First, if σ^a is occupied (i.e., not available), linked to τ^b through the *next* field, and τ^b is also occupied. Then, we apply the predicate Ψ_{Down} to check that the step is valid. Note that it is possible that in a valid knitted-tree τ^b is available, even though $\sigma^a.next$ points to it, but only when σ^a is the last frame of the lace.
2. Second, when τ^b is occupied and linked to σ^a via its *prev* field. In that case, σ^a *must* be occupied in any valid knitted-tree, and we again check the correctness of the step using Ψ_{Down}. □

B.3 Proofs for Section 5

Theorem 3. *In the minimal model of* $\mathcal{C}_{\text{kt}}(P, m, n)$, $\mathbf{Lab}(\sigma)$ *holds for a label* σ *iff* σ *labels a node in some* (m, n)*-knitted-tree prefix* $\mathcal{K}$ *of* P.

Proof. We present only the "if" direction of the theorem; the "only if" direction is provided in Section 5.

Let t be a node in $\mathcal{K}$ with log σ. We proceed by induction on the length ℓ of the lace of $\mathcal{K}$.

Base case ($\ell = 1$): The root of $\mathcal{K}$ has one frame in the lace. CHC I or CHC II inserts σ into **Lab**, depending on whether 1 or 2 frames are unavailable in σ.

Inductive step ($\ell > 1$): Let $\mathcal{K}_{\ell-1}$ be $\mathcal{K}$ with its the last frame of the lace removed. By inductive hypothesis, its logs are in **Lab**. If σ is a log of $\mathcal{K}_{\ell-1}$, the claim holds. Otherwise, σ includes the last frame in the lace of $\mathcal{K}$, meaning that the logs of t's neighboring nodes (its parent and children) remain unchanged from $\mathcal{K}_{\ell-1}$. By inductive hypothesis those logs belong to **Lab**. Depending on the direction taken in the last step, σ is added to **Lab** via CHC III or CHC V, as detailed in the above description of the individual CHCs. □

Theorem 4. *Let* $\mathcal{I}$ *be an instance of the exit-status problem. Then,* $\mathcal{I}$ *admits a positive answer if and only if the CHC system* $\mathcal{C}_{\text{ex}}(\mathcal{I})$ *is unsatisfiable.*

Proof. Let $\mathcal{I} = (P, k, m, n, Ex)$. We prove the theorem by establishing the equivalent statement: $\mathcal{I}$ *admits a negative answer if and only if* $\mathcal{C}_{\text{ex}}(\mathcal{I})$ *is satisfiable.*

For the "only if" direction, if $\mathcal{I}$ has a negative answer, no (m, n)-knitted-tree of P has an exit status in Ex. Consider the interpretation of **Lab** containing exactly the node labels from all (m, n)-knitted-tree prefixes of P. By Theorem 3, this is the minimal model of $\mathcal{C}_{\mathrm{kt}}\big(P, m, n\big)$ and satisfies CHCs I-V. Since none of these labels have an exit status in Ex, the interpretation also satisfies the query VI, proving that $\mathcal{C}_{\mathrm{ex}}(\mathcal{I})$ is satisfiable.

For the "if" direction, if $\mathcal{C}_{\mathrm{ex}}(\mathcal{I})$ is satisfiable, the interpretation of **Lab** in any model of $\mathcal{C}_{\mathrm{ex}}(\mathcal{I})$ has the property that no label in **Lab** has an exit status in Ex (CHC VI). Since $\mathcal{C}_{\mathrm{ex}}(\mathcal{I})$ implies $\mathcal{C}_{\mathrm{kt}}\big(P, m, n\big)$, by Theorem 3 such interpretation of **Lab** contains at least all the logs of the (m, n)-knitted-tree (prefixes) of P. We conclude that the answer to the exit status problem is negative. □

B.4 Proofs for Section 5.1

Theorem 5 is an immediate consequence of Claim 1 of Theorem 2.

Theorem 6. *If the answer to the exit status problem* $(P, m, n, \{\mathbf{O}, \mathbf{M}, \mathbf{E}\})$ *is negative, then the answer to the memory safety problem for* P *is positive.*

Proof. Since no knitted-tree has exit status **O**, Claim 3 of Theorem 2 ensures that no execution is infinite. Similarly, since no knitted-tree has an exit status in $\{\mathbf{O}, \mathbf{M}, \mathbf{E}\}$, Claim 2 guarantees no execution ends in an error. Thus, all executions reach a final configuration. □

C Detailed Encoding

In this section, we describe in detail the predicates and functions appearing as constraints in the CHCs that recognize knitted-trees.

Consider again the system of CHCs described in Figure 3. Those CHCs are based on the predicates *first_frame*, *start*, *consistent_child*, $\Psi_{Internal}$, Ψ_{Down}, Ψ_{Up}, and *label_exit*, each described in one of the following sections.

C.1 The First Frame of Non-Root Nodes

The predicate in this section is meant to constrain the first frame of every non-root node of a knitted-tree. The only property to be enforced is a relationship between the fields *active* and *active_child*: an auxiliary node ($active = false$) that is not the root must be a leaf of the knitted-tree. Hence, no allocation of new nodes can be performed in this node. This is enforced by setting the $active\_child_j$ flags to *true* for all indices $j \in [k+1, k+m]$. Vice versa, all active nodes start with m auxiliary children with indices $k+1, \ldots, k+m$, available for allocation. We obtain the following predicate:

$$\underline{\mathit{first\_frame}}(f) \overset{\mathsf{def}}{=} \Big(f.\mathit{active} \wedge \textstyle\bigwedge_{j\in[k+1,k+m]} \neg f.\mathit{active\_child}_j\Big) \vee \\ \Big(\neg f.\mathit{active} \wedge \textstyle\bigwedge_{j\in[k+1,k+m]} f.\mathit{active\_child}_j\Big) .$$

C.2 The Start of the Lace

The predicate *start* sets constraints on the first two frames at the root of any knitted-tree. These frames must agree on the fields *active* and *val*. The *prev* field of the second frame points to itself through the value $(-, 2)$. For non-empty input trees, the first frame must be active and the second frame makes $\widehat{p}$ point to the root node, and *active_child* is non-deterministic on the first k children, constraining only the children from index $k+1$ to $k+m$ as inactive. For empty input trees, the first frame is inactive and the second frame sets $\widehat{p}$ to **nil**, and *active_child* requires all children to be inactive. In both cases, pc^2 is set to the first program statement label, with all the other pointer variables set to **nil**.

$$
\begin{aligned}
\underline{\mathit{start}}(\sigma) \overset{\mathsf{def}}{=} & \mathit{initial}(\sigma) \wedge \\
& \sigma.\mathit{active}^2 = \sigma.\mathit{active}^1 \wedge \sigma.\mathit{val}^2 = \sigma.\mathit{val}^1 \wedge \\
& \sigma.\mathit{pc}^2 = 0 \wedge \textstyle\bigwedge_{p \in PV_P \setminus \{\widehat{p}\}} \sigma.\mathit{isnil}^2_p \wedge \\
& \Big[\Big(\sigma.\mathit{active}^1 \wedge \sigma.\mathit{event}^2 = \langle \widehat{p} := \mathbf{here} \rangle \wedge \neg \sigma.\mathit{isnil}^2_{\widehat{p}} \wedge \qquad \text{Non-empty input tree} \\
& \quad \textstyle\bigwedge_{j \in [k+1, k+m]} \neg \sigma.\mathit{active\_child}^2_j \Big) \vee \\
& \quad \Big(\neg \sigma.\mathit{active}^1 \wedge \sigma.\mathit{event}^2 = \textsc{nop} \wedge \sigma.\mathit{isnil}^2_{\widehat{p}} \wedge \qquad \text{Empty input tree} \\
& \quad \textstyle\bigwedge_{j \in [k+m]} \neg \sigma.\mathit{active\_child}^2_j \Big) \Big],
\end{aligned}
$$

where the *initial* predicate describes the distinguishing feature of the initial frame of a lace – namely, having itself as predecessor:

$$
\underline{\mathit{initial}}(\sigma) \overset{\mathsf{def}}{=} \neg \sigma.\mathit{avail}^2 \wedge (\sigma.\mathit{prev}^2 = (-, 2)) .
$$

C.3 Lace Termination

The function *label_exit*(σ) returns the exit status of any frame in σ, assuming that at most one frame in a label is terminal. In the following, we denote by $P(i)$ the instruction located at program counter i in the program P.

$$\underline{continues}(f) \stackrel{\text{def}}{=} \neg f.avail \wedge frame\_exit(f) = N$$

$$\underline{frame\_exit}(f) \stackrel{\text{def}}{=} \begin{cases} C & \text{if } P(f.pc) = \mathbf{exit} \\ E & \text{if } f.event = \text{ERR} \\ M & \text{if } f.event = \text{OOM} \\ N & \text{otherwise.} \end{cases}$$

$$\underline{label\_exit}(\sigma) \stackrel{\text{def}}{=} \begin{cases} C & \text{if } \bigvee_{i\in[2,n]} \big(\neg\sigma.avail^i \wedge frame\_exit(\sigma^i) = C\big) \\ E & \text{if } \bigvee_{i\in[2,n]} \big(\neg\sigma.avail^i \wedge frame\_exit(\sigma^i) = E\big) \\ M & \text{if } \bigvee_{i\in[2,n]} \big(\neg\sigma.avail^i \wedge frame\_exit(\sigma^i) = M\big) \\ O & \text{if } \neg\sigma.avail^{n+1} \\ N & \text{otherwise.} \end{cases}$$

$$\underline{label\_exit}(\sigma, Ex) \stackrel{\text{def}}{=} \big(label\_exit(\sigma) \in Ex\big)$$

C.4 Checking Parent-Child Consistency

The *consistent_child*(τ, j, σ) predicate checks that the label τ of the j-th child is consistent with the label σ of the parent. It uses the predicates *consistent_first_frames*, Ψ_{Down}, and Ψ_{Up}, defined later.

$$\underline{consistent\_child}(\tau, j, \sigma) \stackrel{\text{def}}{=} consistent\_first\_frames(\tau, j, \sigma) \wedge$$

Consistency of steps, when going down:

$$\bigwedge_{\substack{a\in[2,n]\\ b\in[2,n+1]}} \Big[\Big(\big(\neg\sigma.avail^a \wedge \sigma.next^a = (j,b) \wedge \neg\tau.avail^b\big) \vee \big(\neg\tau.avail^b \wedge \tau.prev^b = (\uparrow, a)\big) \Big) \rightarrow \Psi_{Down}(\sigma^{\leq a}, j, \tau^{<b}, \tau^b)\Big] \wedge$$

Consistency of steps, when going up:

$$\bigwedge_{\substack{a\in[2,n]\\ b\in[2,n+1]}} \Big[\Big(\big(\neg\tau.avail^a \wedge \tau.next^a = (\uparrow,b) \wedge \neg\sigma.avail^b\big) \vee \big(\neg\sigma.avail^b \wedge \sigma.prev^b = (j, a)\big) \Big) \rightarrow \Psi_{Up}(\tau^{\leq a}, j, \sigma^{<b}, \sigma^b)\Big].$$

It is worth pointing out that different child labels may be consistent with the same parent label, just like two different parent labels may be consistent with the same child label. First, *consistent_child* only constrains frames that are directly involved in an interaction between parent and child, i.e., pairs of frames belonging to these different nodes and adjacent in the lace. Now, consider a pair of frames that are adjacent in the lace and belong to two different nodes (parent and child). Here is a description of how *consistent_child* constrains each field in those frames:

avail: this field must be false, because frames involved in an interaction belong to the lace;

active: this field is directly constrained only in the first frame of each label, where its value must follow the backbone rules described in Section 3.1; this is ensured by the auxiliary predicate *consistent_first_frames*; the value in the subsequent frames of each label is only constrained indirectly, by the semantics of the **new** and **free** instructions;

val: the default constraint for this field is to have the same value as the preceding frame in the same log; the only exception is the last step of the instruction $p \to \textbf{data} := exp$, when the current value of p has been ascertained via rewinding: in such a step the value of the *val* field of the new frame takes the value of the expression *exp*;

pc: the program counter must be consistent with the corresponding *pc* on the other side of the interaction; i.e., the two pc's must be equal if a rewind operation is ongoing, and otherwise the instruction being executed is complete and the pc advances according to the control-flow graph of P;

d: the value of this field in the two frames of an interaction is the same, except in the last step of an instruction of the type $d := p \to \textbf{data}$, when the current value of p has been ascertained via rewinding;

upd_p: the auxiliary predicates Ψ_* constrain these fields to follow the rules described on page 9;

$isnil_p$: this field generally keeps its value in an interaction, except in the last step of an assignment to p; specifically, it can switch from *true* to *false* in the last step of $p := q$ or $p := q \to pfield$, when q is not **nil** and rewinding was needed to find its current value; moreover, it can switch from *false* to *true* in the last step of $p := q \to pfield$, when q is not **nil**, rewinding was needed, and then $q \to pfield$ is found to be **nil**;

event: the event in the second frame of an interaction is dictated by the program instruction being executed and by the meaning of each event; for example, during a rewind process, the events RWD_i and $\text{RWD}_{i,p}$ are used; the end of a rewinding may be marked by a event of the types $\langle pfield := p\rangle$, $\langle pfield := \textbf{nil}\rangle$, or $\langle p := \textbf{here}\rangle$; in some cases, the final event is simply NOP;

next and *prev*: in a pair of interacting frames these fields must connect the two frames with each other; i.e., the *next* field of the first frame points to the second frame, and the *prev* field of the second frame points to the first one.

The following auxiliary predicates are used within *consistent_child*. The *consistent_first_frames* predicate describes the initial value of the *active* field, that starts as true on all nodes of the knitted-tree that correspond to nodes of the input tree, and false elsewhere.

$$\underline{consistent\_first\_frames}(\tau, j, \sigma) \overset{\text{def}}{=} \big(initial(\sigma) \lor \sigma.active^1\big) \land \big(\tau.active^1 \to \sigma.active^1\big) \land \big(j > k \to \neg\tau.active^1\big) \land \sigma.active\_child^1_j = \tau.active^1 .$$

We now present the three predicates that describe the next frame in the lace, depending on the direction of the next step. The first predicate, $\Psi_{Internal}$, describes internal steps. It invokes the general predicate *step* that checks a single

step in a lace, passing two adjacent frames from the same log. In addition, it makes sure that the upd_p flags are all false after an internal step, according to the meaning of those flags, as described on page 9.
In the following formulas, for a label σ and a frame f we write $\sigma \cdot f$ to denote the label obtained by the replacing the first available frame of σ with f.

$$\underline{\Psi_{Internal}}(\sigma, f) \overset{\text{def}}{=} len(\sigma, a) \land continues(\sigma^a) \land \sigma.next^a = (-, a+1) \land \\ f.prev = (-, a) \land step(\sigma, a; \sigma \cdot f, a+1) \land \textstyle\bigwedge_{p \in PV_P} \neg f.upd_p \,.$$

The following predicates Ψ_{Down} and Ψ_{Up} describe the next frame in the lace, when the current step involves a change of node, either from a node down to its j-th child or from a node up to its parent. In particular, these predicates constrain the *upd* flags in the new frame to respect their meaning, as described on page 9.

Consider how the last two lines in $\Psi_{Up}(\tau, j, \sigma, f)$ relate to the intended behavior of the *upd* flags. In that context, τ^a is a frame in the j-th child of a parent node with log σ, and the next frame in the lace is f, to be appended on top of σ. Note that in this situation the previous frame in the log of the parent (i.e., σ^{b-1}) is followed in the lace by a frame in the j-th child. Now, for every pointer variable $p \in PV_P$, the upd_p flag when going back to the parent is true iff p is not **nil** at that time and we observe a marker for an update to p within the frames of the child located between the step down and the step back up. If the update to p was performed in one of the frames of the child, the marker is a direct event of the type $\langle p := \textbf{here} \rangle$. If instead the update occurs further below in the subtree rooted in the child, the marker observed in the child is a upd_p field set to true, as encoded in the last lines of Ψ_{Up}.

Finally, note that Ψ_{Up} is also responsible for synchronizing the *active_child* field in the parent with the *active* field in the child; these fields may become out-of-sync after a **free** operation performed on the child.

$$\underline{\Psi_{Down}}(\sigma, j, \tau, f) \overset{\text{def}}{=} len(\sigma, a) \land continues(\sigma^a) \land \sigma.next^a = (j, b) \land len(\tau, b-1) \land \\ f.prev = (\uparrow, a) \land step(\sigma, a; \tau \cdot f, b) \land \\ \bigwedge_{p \in PV_P} \Big[f.upd_p \leftrightarrow \Big(\neg f.isnil_p \land b > 2 \land \tau.next^{b-1} = (\uparrow, a') \land \\ \Big(\textstyle\bigvee_{c \in [a', a]} \big(\sigma.event^c = \langle p := \textbf{here} \rangle\big) \lor \textstyle\bigvee_{c \in [a'+1, a]} \sigma.upd_p^c \Big) \Big) \Big]$$

$$\underline{\Psi_{Up}}(\tau, j, \sigma, f) \overset{\text{def}}{=} len(\tau, a) \land continues(\tau^a) \land \tau.next^a = (\uparrow, b) \land len(\sigma, b-1) \land \\ f.prev = (j, a) \land step(\tau, a; \sigma \cdot f, b) \land f.active\_child = \tau.active^a \land \\ \bigwedge_{p \in PV_P} \Big[f.upd_p \leftrightarrow \Big(\neg f.isnil_p \land b > 2 \land \sigma.next^{b-1} = (j, a') \land \\ \Big(\textstyle\bigvee_{c \in [a', a]} \big(\tau.event^c = \langle p := \textbf{here} \rangle\big) \lor \textstyle\bigvee_{c \in [a'+1, a]} \tau.upd_p^c \Big) \Big) \Big].$$

C.5 Individual Statements

The predicate $\mathit{step}(\sigma, a; \tau, b)$ holds if τ^b is the logically correct frame to follow σ^a in a lace. The structure of this predicate comprises an implication for each type of instruction in our programming language, except for **exit**. The **exit** instruction does not produce a step, because just having the program counter point to it identifies a terminating execution.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\underline{\mathit{step}}(\sigma, a; \tau, b) \overset{\text{def}}{=}} \\
\quad P(\sigma.pc^a) = \textbf{skip} & \rightarrow \mathit{step\_skip}(\sigma^a, \tau^b) \wedge \\
\quad P(\sigma.pc^a) = (p \coloneqq \textbf{nil}) & \rightarrow \mathit{step\_assgn\_nil}(\sigma^a, \tau^b, p) \wedge \\
\quad P(\sigma.pc^a) = (d \coloneqq \mathit{exp}) & \rightarrow \mathit{step\_assgn\_exp}(\sigma^a, \tau^b, d, \mathit{exp}) \wedge \\
\quad P(\sigma.pc^a) = (d_{\text{bool}} \coloneqq (p = q)) & \rightarrow \mathit{step\_assgn\_cond}(\sigma, a, \tau, b, d_{\text{bool}}, p, q) \wedge \\
\quad P(\sigma.pc^a) = (p \coloneqq q) & \rightarrow \mathit{step\_assgn\_ptr}(\sigma, a, \tau, b, p, q) \wedge \\
\quad P(\sigma.pc^a) = (p \coloneqq q \rightarrow \mathit{pfield}) & \rightarrow \mathit{step\_assgn\_from\_field}(\sigma, a, \tau, b, p, q, \mathit{pfield}) \wedge \\
\quad P(\sigma.pc^a) = (p \rightarrow \mathit{pfield} \coloneqq q) & \rightarrow \mathit{step\_assgn\_to\_field}(\sigma, a, \tau, b, p, q, \mathit{pfield}) \wedge \\
\quad P(\sigma.pc^a) = (p \rightarrow \mathit{val} \coloneqq \mathit{exp}) & \rightarrow \mathit{step\_assgn\_to\_data}(\sigma, a, \tau, b, p, \mathit{exp}) \wedge \\
\quad P(\sigma.pc^a) = (d \coloneqq p \rightarrow \mathit{val}) & \rightarrow \mathit{step\_assgn\_to\_var}(\sigma, a, \tau, b, d, p) \wedge \\
\quad P(\sigma.pc^a) = (\textbf{new}\ p) & \rightarrow \mathit{step\_new}(\sigma^a, \tau^{b-1}, \tau^b, p) \wedge \\
\quad P(\sigma.pc^a) = (\textbf{free}\ p) & \rightarrow \mathit{step\_free}(\sigma, a, \tau, b, p) \wedge \\
\quad \big(P(\sigma.pc^a) = (\textbf{while}\ p = q) \vee & \\
\quad\ \ P(\sigma.pc^a) = (\textbf{if}\ p = q)\big) & \rightarrow \mathit{step\_cmp\_ptr}(\sigma, a, \tau, b, p, q, \mathit{false}) \wedge \\
\quad \big(P(\sigma.pc^a) = (\textbf{while}\ \neg(p = q)) \vee & \\
\quad\ \ P(\sigma.pc^a) = (\textbf{if}\ \neg(p = q))\big) & \rightarrow \mathit{step\_cmp\_ptr}(\sigma, a, \tau, b, p, q, \mathit{true}) \wedge \\
\quad \big(P(\sigma.pc^a) = (\textbf{while}\ r(\mathit{exp}_1, \ldots, \mathit{exp}_l)) \vee & \\
\quad\ \ P(\sigma.pc^a) = (\textbf{if}\ r(\mathit{exp}_1, \ldots, \mathit{exp}_l))\big) & \rightarrow \mathit{step\_local\_branch}(\sigma, a, \tau, b, r, \mathit{exp}_1, \ldots, \mathit{exp}_l)\,.
\end{array}
$$

Default values. Regardless of any specific instruction, most fields in any new frame of the lace are set to their default values, as described in Section 3.1. The following *default* predicate represents the full case, where all fields that have default values are constrained. In the subsequent predicates, we add subscripts to *default* to specify which fields follow the default rules.

$$
\begin{array}{lll}
\underline{\mathit{default}}(f^{prev}, f^{below}, f) \overset{\text{def}}{=} & \neg f.\mathit{avail} \wedge & \mathit{avail} \\
& f.\mathit{active} = f^{below}.\mathit{active} \wedge & \mathit{active} \\
& f.\mathit{val} = f^{below}.\mathit{val} \wedge & \mathit{val} \\
& \bigwedge_{d \in DV_P} f.d = f^{prev}.d \wedge & d \\
& \bigwedge_{p \in PV_P} f.\mathit{isnil}_p = f^{prev}.\mathit{isnil}_p \wedge & \mathit{isnil} \\
& f.\mathit{event} = \text{NOP} \wedge & \mathit{event} \\
& f.pc = f^{prev}.pc \wedge & pc \\
& \mathit{default}_{\mathit{active\_child}}(f^{prev}, f^{below}, f). & \mathit{active\_child}
\end{array}
$$

The default for the *active_child* flags requires a little more care:

$$\begin{aligned}
&\underline{\mathit{default}_{\mathit{active\_child}}}(f^{\mathit{prev}}, f^{\mathit{below}}, f) \overset{\mathsf{def}}{=} \\
&\qquad f.\mathit{prev} = (j,*) \to \Big(f.\mathit{active\_child}_j = f^{\mathit{prev}}.\mathit{active} \wedge \\
&\qquad\qquad \textstyle\bigwedge_{j \neq i} \big(f.\mathit{active\_child}_i = f^{\mathit{below}}.\mathit{active\_child}_i \big) \Big) \wedge \\
&\qquad f.\mathit{prev} \neq (j,*) \to \textstyle\bigwedge_{i \in [k+m]} \big(f.\mathit{active\_child}_i = f^{\mathit{below}}.\mathit{active\_child}_i \big) .
\end{aligned}$$

Internal instructions. Internal instructions push a single new frame on the current node of the knitted-tree. As our first type of internal instruction, we describe the *step_assgn_nil* predicate. As explained earlier, its encoding simply pushes a new frame f_2 on the current node, with the isnil_p flag set to *true*. All the other fields of the new frame take their default value, except the program counter, which advances to the next instruction:

$$\begin{aligned}
&\underline{\mathit{step\_assgn\_nil}}(f_1, f_2, p) \overset{\mathsf{def}}{=} (f_1.\mathit{next} = -) \wedge \qquad\qquad p := \mathbf{nil} \\
&\qquad f_2.\mathit{isnil}_p \wedge \textstyle\bigwedge_{q \in PV_P \setminus \{p\}} (f_2.\mathit{isnil}_q = f_1.\mathit{isnil}_q) \wedge \\
&\qquad \mathit{advance\_pc}(f_1, f_2) \wedge \mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{event},\mathit{active\_child}}(f_1, f_1, f_2).
\end{aligned}$$

The following auxiliary predicate encodes the advancement of the program counter:

$$\underline{\mathit{advance\_pc}}(f_1, f_2) \overset{\mathsf{def}}{=} \big(f_2.\mathit{pc} = \mathit{succ}(f_1.\mathit{pc}) \big).$$

Next, we present the predicate that encodes the **new** instruction. Note that a failed **new** is only justified if all children of the current node with position in $[k+1, k+m]$ are active (i.e., currently allocated).

$$\begin{aligned}
&\underline{\mathit{step\_new}}(f, f', f'', p) \overset{\mathsf{def}}{=} \qquad\qquad \mathbf{new}\ p \\
&\qquad \text{Case 1: Out-of-memory error} \\
&\qquad \Big(\textstyle\bigwedge_{j \in [k+1,k+m]} f.\mathit{active\_child}_j \wedge f.\mathit{next} = (-,*) \wedge f''.\mathit{instr} = \textsc{oom} \wedge \\
&\qquad\quad \mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{pc},\mathit{isnil},\mathit{active\_child}}(f, f', f'') \Big) \quad \vee \\
&\qquad \text{Case 2: Normal case} \\
&\qquad \Big(\bigvee_{j \in [k+1,k+m]} \big(\neg f.\mathit{active\_child}_j \wedge \textstyle\bigwedge_{i \in [k+1,j-1]} f.\mathit{active\_child}_i \wedge f.\mathit{next} = (j,*) \big) \wedge \\
&\qquad \neg f''.\mathit{isnil}_p \wedge \bigwedge_{q \in PV_P \setminus \{p\}} \big(f''.\mathit{isnil}_q = f.\mathit{isnil}_q \big) \wedge \\
&\qquad f''.\mathit{event} = \langle p := \mathbf{here} \rangle \wedge f''.\mathit{active} \wedge \mathit{advance\_pc}(f, f'') \wedge \\
&\qquad \mathit{default}_{\mathit{avail},\mathit{val},d,\mathit{active\_child}}(f, f', f'') \Big) .
\end{aligned}$$

The following predicate handles assignments of arbitrary data expressions to data variables.

$$\underline{\mathit{step\_assgn\_exp}}(f_1, f_2, d, \mathit{exp}) \overset{\mathsf{def}}{=} \qquad d \coloneqq \mathit{exp}$$
$$\big(f_1.\mathit{next} = (-, *)\big) \wedge f_2.d = \mathit{exp}[d' \mapsto f_1.d']_{d' \in DV_P} \wedge$$
$$\textstyle\bigwedge_{d' \in DV_P \setminus \{d\}} (f_2.d' = f_1.d') \wedge \mathit{advance\_pc}(f_1, f_2) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},\mathit{isnil},\mathit{event},\mathit{active\_child}}(f_1, f_1, f_2)\,.$$

Finally, the following is the straightforward encoding of the **skip** statement.

$$\underline{\mathit{step\_skip}}(f_1, f_2, p) \overset{\mathsf{def}}{=} \big(f_1.\mathit{next} = (-, *)\big) \wedge \mathit{advance\_pc}(f_1, f_2) \wedge \qquad \textbf{skip}$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{isnil},\mathit{event},\mathit{active\_child}}(f_1, f_1, f_2).$$

Walking instructions. The following *walking instructions* may add multiple frames to different nodes of the knitted-tree. The first such instruction is the assignment of the form $p \coloneqq q$, which may require rewinding the lace to find the node currently pointed by q. In fact, the encoding distinguishes three cases: (1) when q is **nil**, (3) when q points elsewhere, and we need to start or keep rewinding the lace, (2) when q points to the current node (i.e., the node with label σ). The predicates governing the rewinding operation are presented later in Section C.7.

$$\underline{\mathit{step\_assgn\_ptr}}(\sigma, a, \tau, b, p, q) \overset{\mathsf{def}}{=} \qquad p \coloneqq q$$

Case 1: q is **nil**; use the encoding for $p \coloneqq$ **nil**

$$\Big(\sigma^a.\mathit{isnil}_q \wedge \mathit{step\_assgn\_nil}(\sigma^a, \tau^b, p)\Big) \quad \vee$$

Case 2: q points elsewhere; start or keep rewinding

$$\mathit{rewind}(\sigma, a, \tau, b, q) \quad \vee$$

Case 3: q points to the current node

$$\Big(\mathit{stop\_rewind}(\sigma, a, q) \wedge \mathit{set\_ptr\_here}(\sigma^a, \tau^b, p)\Big)\,.$$

The auxiliary predicate *set_ptr_here* pushes a frame on the current node with $\langle p \coloneqq \textbf{here}\rangle$ and updates *isnil* accordingly.

$$\underline{\mathit{set\_ptr\_here}}(f_1, f_2, p) \overset{\mathsf{def}}{=} \big(f_1.\mathit{next} = (-, *)\big) \wedge \mathit{advance\_pc}(f_1, f_2) \wedge$$
$$f_2.\mathit{event} = \langle p \coloneqq \textbf{here}\rangle \wedge$$
$$\neg f_2.\mathit{isnil}_p \wedge \textstyle\bigwedge_{q \in PV_P \setminus \{p\}} \big(f_2.\mathit{isnil}_q = f_1.\mathit{isnil}_q\big) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{active\_child}}(f_1, f_1, f_2)\,.$$

The following predicate encodes the statements of the form $p \to \mathit{pfield} := q$.

$$\underline{\mathit{step\_assgn\_to\_field}}(\sigma, a, \tau, b, p, q, \mathit{pfield}) \overset{\mathsf{def}}{=} \mathit{find\_or\_fail}(\sigma, a, \tau, b, p) \vee \qquad p \to \mathit{pfield} := q$$
$$p \text{ points to the current node:}$$
$$\Big(\mathit{stop\_rewind}(\sigma, a, p) \wedge \sigma^a.\mathit{next} = (-, a+1) \wedge \mathit{advance\_pc}(\sigma^a, \tau^b) \wedge$$
$$\big((\neg\sigma^a.\mathit{isnil}_q \wedge \tau^b.\mathit{event} = \langle \mathit{pfield} := q \rangle) \vee$$
$$(\sigma^a.\mathit{isnil}_q \wedge \tau^b.\mathit{event} = \langle \mathit{pfield} := \mathbf{nil} \rangle) \big) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{isnil},\mathit{active\_child}}(\sigma^a, \tau^{b-1}, \tau^b) \Big).$$

The predicate *step_assgn_nil_to_field*, corresponding to the statements of the form $p \to \mathit{pfield} := \mathbf{nil}$, is a simple variant of the above, that only sets the event $\langle \mathit{pfield} := \mathbf{nil} \rangle$.

Next, we deal with the two instructions that write or read the data field of a node.

$$\underline{\mathit{step\_assgn\_to\_data}}(\sigma, a, \tau, b, p, \mathit{exp}) \overset{\mathsf{def}}{=} \mathit{find\_or\_fail}(\sigma, a, \tau, b, p) \vee \qquad p \to \mathit{val} := \mathit{exp}$$
$$p \text{ points to the current node:}$$
$$\Big(\mathit{stop\_rewind}(\sigma, a, p) \wedge \big(\sigma^a.\mathit{next} = (-, a+1)\big) \wedge \mathit{advance\_pc}(\sigma^a, \tau^b) \wedge$$
$$\tau^b.\mathit{val} = \mathit{exp}\big[d \mapsto \sigma^a.d\big]_{d \in DV_P} \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},d,\mathit{isnil},\mathit{event},\mathit{active\_child}}(\sigma^a, \tau^{b-1}, \tau^b) \Big).$$

$$\underline{\mathit{step\_assgn\_to\_var}}(\sigma, a, \tau, b, d, p) \overset{\mathsf{def}}{=} \mathit{find\_or\_fail}(\sigma, a, \tau, b, p) \vee \qquad d := p \to \mathit{val}$$
$$p \text{ points to the current node:}$$
$$\Big(\mathit{stop\_rewind}(\sigma, a, p) \wedge \big(\sigma^a.\mathit{next} = (-, a+1)\big) \wedge \mathit{advance\_pc}(\sigma^a, \tau^b) \wedge$$
$$\tau^b.d = \sigma^a.\mathit{val} \wedge \textstyle\bigwedge_{d' \in DV_P \setminus \{d\}} (\tau^b.d' = \sigma^a.d') \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},\mathit{isnil},\mathit{event},\mathit{active\_child}}(\sigma^a, \tau^{b-1}, \tau^b) \Big).$$

The following predicate handles the deallocation of the node pointed by a given pointer.

$$\underline{\mathit{step\_free}}(\sigma, a, \tau, b, p) \overset{\text{def}}{=} \mathit{find\_or\_fail}(\sigma, a, \tau, b, p) \vee \qquad \textbf{free } p$$

p points to the current node:

$$\Big(\mathit{stop\_rewind}(\sigma, a, p) \wedge \big(\sigma^a.\mathit{next} = (-, *)\big) \wedge \mathit{advance\_pc}(\sigma^a, \tau^b) \wedge$$
$$\neg\tau^b.\mathit{active} \wedge$$
$$\bigwedge_{q \in PV_P} \Big(\big(\mathit{points\_here}(\sigma, a, q) \to \tau^b.\mathit{isnil}_q\big) \wedge$$
$$\big(\neg\mathit{points\_here}(\sigma, a, q) \to \tau^b.\mathit{isnil}_q = \sigma^a.\mathit{isnil}_q\big) \Big) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{val},d,\mathit{event},\mathit{active\_child}}(\sigma^a, \tau^{b-1}, \tau^b)\Big) .$$

The following auxiliary predicates handle the search for the current value of a pointer variable p, including issuing an error if p is **nil**.

$$\underline{\mathit{find\_or\_fail}}(\sigma, a, \tau, b, p) \overset{\text{def}}{=} \Big(\sigma^a.\mathit{isnil}_p \wedge \mathit{error}(\sigma^a, \tau^b)\Big) \vee \mathit{rewind}(\sigma, a, \tau, b, p)$$

$$\underline{\mathit{error}}(f_1, f_2) \overset{\text{def}}{=} \big(f_1.\mathit{next} = (-, *)\big) \wedge (f_2.\mathit{event} = \text{ERR}) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{isnil},\mathit{pc},\mathit{active\_child}}(f_1, f_1, f_2) .$$

Next, we move to the instruction that assigns to a Boolean variable the result of the comparison between two pointers. If at least one of the two pointers is **nil**, the comparison can be resolved locally. Otherwise, a rewind operation may be necessary. The auxiliary predicates *rewind2*, *stop_rewind2*, and *are_equal_after_rewind* are described in Sec. C.7.

$$\underline{\mathit{step\_assgn\_cond}}(\sigma, a, \tau, b, d_{\text{bool}}, p, q) \overset{\text{def}}{=} \qquad d_{\text{bool}} := (p = q)$$

Case 1: at least one pointer is **nil**

$$\Big(\big(\sigma^a.\mathit{isnil}_p \vee \sigma^a.\mathit{isnil}_q\big) \wedge$$
$$\big(\sigma^a.\mathit{next} = (-, *)\big) \wedge \mathit{advance\_pc}(\sigma^a, \tau^b) \wedge$$
$$\tau^b.d_{\text{bool}} = \big(\sigma^a.\mathit{isnil}_p \leftrightarrow \sigma^a.\mathit{isnil}_q\big)\big) \wedge$$
$$\bigwedge\nolimits_{d \in DV_P \setminus \{d_{\text{bool}}\}} (\tau^b.d = \sigma^a.d) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},\mathit{isnil},\mathit{event},\mathit{active\_child}}\big(\sigma^a, \tau^{b-1}, \tau^b\big) \Big) \quad \vee$$

Case 2: start or keep rewinding

$$\mathit{rewind2}(\sigma, a, \tau, b, p, q) \quad \vee$$

Case 3: stop rewinding

$$\Big(\mathit{stop\_rewind2}(\sigma, a, p, q) \wedge \big(\sigma^a.\mathit{next} = (-, *)\big) \wedge \mathit{advance\_pc}(\sigma^a, \tau^b) \wedge$$
$$\tau^b.d_{\text{bool}} = \mathit{are\_equal\_after\_rewind}(\sigma, a, p, q)\big) \wedge$$
$$\bigwedge\nolimits_{d \in DV_P \setminus \{d_{\text{bool}}\}} (\tau^b.d = \sigma^a.d) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},\mathit{isnil},\mathit{event},\mathit{active\_child}}\big(\sigma^a, \tau^{b-1}, \tau^b\big)\Big) .$$

The most complex walking instruction is the assignment of the form $p := q \to \mathit{pfield}$, because it may involve *two* consecutive rewinding phases.

$$\underline{\mathit{step\_assgn\_from\_field}}(\sigma, a, \tau, b, p, q, \mathit{pfield}) \overset{\text{def}}{=} \qquad\qquad p := q \to \mathit{pfield}$$

Case 1: q is **nil**; null pointer dereference

$$\big(\sigma^a.\mathit{isnil}_q \;\wedge\; \mathit{error}(\sigma^a, \tau^b)\big) \quad \vee$$

Case 2a: phase I; q points elsewhere; start or keep rewinding

$$\big(\sigma^a.\mathit{event} \neq \mathrm{RWD}_{*,*} \;\wedge\; \mathit{rewind}(\sigma, a, \tau, b, q)\big) \quad \vee$$

Case 2b: phase II; r points elsewhere; start or keep rewinding

$$\big(\sigma^a.\mathit{event} = \mathrm{RWD}_{i,r} \;\wedge\; \mathit{rewind\_special}(\sigma, a, \tau, b, r, i)\big) \quad \vee$$

Case 3a: end of phase I; q points to the current node

$$\begin{aligned}
&\Big(\sigma^a.\mathit{event} \neq \mathrm{RWD}_{*,*} \wedge \mathit{stop\_rewind}(\sigma, a, q) \wedge \\
&\;\Big(\big(\mathit{is\_pfield\_nil}(\sigma, a, \mathit{pfield}) \vee \\
&\quad (\mathit{is\_pfield\_implicit}(\sigma, a, \mathit{pfield}) \wedge \neg\sigma^a.\mathit{active\_child}_{\mathit{index}(\mathit{pfield})})\big) \to \\
&\qquad \mathit{step\_assign\_nil}(\sigma^a, \tau^b, p)\Big) \wedge \\
&\;\Big(\big(\mathit{is\_pfield\_implicit}(\sigma, a, \mathit{pfield}) \;\wedge\; \sigma^a.\mathit{active\_child}_{\mathit{index}(\mathit{pfield})}\big) \to \\
&\qquad \big(\sigma^a.\mathit{next} = (\mathit{index}(\mathit{pfield}), b) \;\wedge\; \mathit{advance\_pc}(\sigma^a, \tau^b) \wedge \\
&\qquad\; \tau^b.\mathit{event} = \langle p := \textbf{here}\rangle \wedge \\
&\qquad\; \mathit{default}_{\mathit{avail}, \mathit{active}, \mathit{val}, d, \mathit{isnil}, \mathit{active\_child}}(\sigma^a, \tau^{b-1}, \tau^b)\big)\Big) \wedge \\
&\bigwedge_{r \in PV_P, i \in [2,n]} \Big(\big(\mathit{is\_pfield\_ptr}(\sigma, a, \mathit{pfield}, r, i) \wedge \mathit{points\_here}(\sigma, i, r)\big) \to \\
&\qquad\qquad \mathit{set\_ptr\_here}(\sigma^a, \tau^b, p)\Big) \wedge \\
&\bigwedge_{r \in PV_P, i \in [2,n]} \Big(\big(\mathit{is\_pfield\_ptr}(\sigma, a, \mathit{pfield}, r, i) \wedge \neg\mathit{points\_here}(\sigma, i, r)\big) \to \\
&\qquad\qquad \mathit{rewind\_special}(\sigma, a, \tau, b, r, i)\Big)\Big) \quad \vee
\end{aligned}$$

Case 3b: end of phase II; r points to the current node

$$\Big(\sigma^a.\mathit{event} = \mathrm{RWD}_{i,r} \wedge \mathit{points\_here}(\sigma, i, r) \;\wedge\; \mathit{set\_ptr\_here}(\sigma^a, \tau^b, p)\Big)\,.$$

The following auxiliary predicates check the value of the node field *pfield* at (σ, a).

pfield is **nil** at the frame (σ, a) of the lace:

$$\underline{\mathit{is\_pfield\_nil}}(\sigma, a, \mathit{pfield}) \overset{\text{def}}{=}$$

$$\bigvee_{i \in [2,a]} \Big(\sigma^i.\mathit{event} = \langle \mathit{pfield} := \textbf{nil}\rangle \wedge \bigwedge_{j \in [i+1,a]} \sigma^j.\mathit{event} \neq \langle \mathit{pfield} := *\rangle\Big)$$

pfield has the same value as r at the frame (σ, a) of the lace:

$$\underline{\mathit{is\_pfield\_ptr}}(\sigma, a, \mathit{pfield}, r, i) \overset{\text{def}}{=}$$
$$\sigma^i.\mathit{event} = \langle \mathit{pfield} := r\rangle \wedge \bigwedge_{j\in[i+1,a]} \sigma^j.\mathit{event} \neq \langle \mathit{pfield} := *\rangle$$

pfield has never been assigned up to the frame (σ, a) of the lace:

$$\underline{\mathit{is\_pfield\_implicit}}(\sigma, a, \mathit{pfield}) \overset{\text{def}}{=} \bigwedge_{j\in[2,a]} \sigma^j.\mathit{event} \neq \langle \mathit{pfield} := *\rangle\,.$$

C.6 Boolean Conditions and Control-Flow Instructions

Control-flow statements **if** and **while** have two successors, depending on the value of their Boolean condition. To support those statements, we introduce a 3-argument version of the predicate that advances the program counter:

$$\underline{\mathit{advance\_pc}}(f_1, f_2, \mathit{cond}) \overset{\text{def}}{=} \big(f_2.\mathit{pc} = \mathit{succ}(f_1.\mathit{pc}, \mathit{cond})\big).$$

The next predicate handles the Boolean conditions of the form $p = q$:

$$\underline{\mathit{step\_cmp\_ptr}}(\sigma, a, \tau, b, p, q, \mathit{neg}) \overset{\text{def}}{=} \qquad \text{if } p = q$$

Case 1: at least one pointer is **nil**

$$\Big(\big(\sigma^a.\mathit{isnil}_p \vee \sigma^a.\mathit{isnil}_q\big) \wedge \big(\sigma^a.\mathit{next} = (-, a+1)\big) \wedge$$
$$\mathit{advance\_pc}(\sigma^a, \tau^b, \mathrm{xor}(\mathit{neg}, \sigma^a.\mathit{isnil}_p \leftrightarrow \sigma^a.\mathit{isnil}_q)) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{isnil},\mathit{event},\mathit{active\_child}}\big(\sigma^a, \tau^{b-1}, \tau^b\big) \Big) \quad \vee$$

Case 2: start or keep rewinding

$$\mathit{rewind2}(\sigma, a, \tau, b, p, q) \quad \vee$$

Case 3: stop rewinding

$$\Big(\mathit{stop\_rewind2}(\sigma, a, p, q) \wedge \big(\sigma^a.\mathit{next} = (-, a+1)\big) \wedge$$
$$\mathit{advance\_pc}\big(\sigma^a, \tau^b, \mathrm{xor}(\mathit{neg}, \mathit{are\_equal\_after\_rewind}(\sigma, a, p, q))\big) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{isnil},\mathit{event},\mathit{active\_child}}(\sigma^a, \tau^{b-1}, \tau^b)\Big)\,.$$

The next predicate handles the Boolean conditions based on the content of the data variables. Those conditions can always be resolved locally.

$$\underline{\mathit{step\_local\_branch}}(f_1, f_2, r, \mathit{exp}_1, \ldots, \mathit{exp}_l) \overset{\text{def}}{=} \qquad \text{if } r(\mathit{exp}_1, \ldots, \mathit{exp}_l)$$
$$\big(f_1.\mathit{next} = (-, *)\big) \wedge$$
$$\mathit{advance\_pc}\big(f_1, f_2, r(\mathit{exp}_1[d \mapsto f_1.d]_{d\in DV_P}, \ldots, \mathit{exp}_l[d \mapsto f_1.d]_{d\in DV_P})\big) \wedge$$
$$\mathit{default}_{\mathit{avail},\mathit{active},\mathit{val},d,\mathit{isnil},\mathit{event},\mathit{active\_child}}(f_1, f_1, f_2)\,.$$

C.7 Rewinding the Lace

The following predicate is true when the pointer q is not **nil** at (σ, a) and its value cannot be ascertained by analyzing the label σ alone. In that case, it is necessary to perform at least one rewinding step, represented by the frame (τ, b).

$$\begin{aligned}
&\underline{\mathit{rewind}}(\sigma, a, \tau, b, q) \overset{\text{def}}{=} \\
&\qquad \sigma^a.\mathit{next} = (\mathit{dir}, b) \wedge \neg\sigma^a.\mathit{isnil}_q \wedge a' = \mathit{cur\_rewind\_pos}(\sigma, a) \wedge \\
&\qquad \neg \mathit{points\_here}(\sigma, a', q) \wedge a'' = \mathit{last\_upd}(\sigma, a', q) \wedge \\
&\qquad \sigma^{a''}.\mathit{prev} = (\mathit{dir}, b') \wedge \tau^{b'}.\mathit{next} = (\mathit{dir}', a'') \wedge \\
&\qquad b = \mathit{last\_visit}(\tau, \mathit{dir}', a') + 1 \wedge \tau^b.\mathit{event} = \textsc{rwd}_{b'} \wedge \\
&\qquad \mathit{default}_{\mathit{avail},\mathit{active},\mathit{pc},\mathit{val},d,\mathit{isnil},\mathit{active\_child}}(\sigma^a, \tau^{b-1}, \tau^b)\,.
\end{aligned}$$

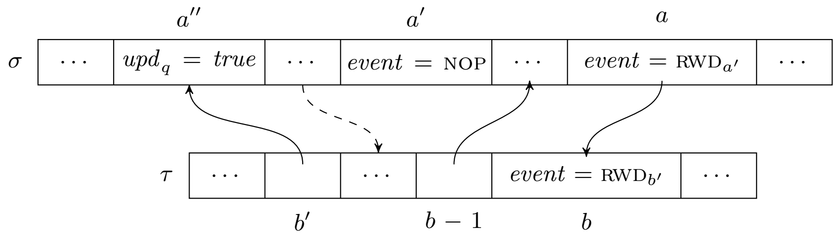

a''
a'
a
σ
$\cdots$
$\mathit{upd}_q = \mathit{true}$
$\mathit{event} = \textsc{nop}$
$\mathit{event} = \textsc{rwd}_{a'}$
τ
$\mathit{event} = \textsc{rwd}_{b'}$
b'
$b-1$
b

Fig. 4: An example of the $\mathit{rewind}(\sigma, a, \tau, b, q)$ predicate. Arrows connect a frame to its successor in the lace.

The above predicate uses several auxiliary functions and predicates. The function $\mathit{cur\_rewind\_pos}(\sigma, a)$ returns the position within σ of the current rewinding. If the rewinding is just starting, this position will simply be a. Otherwise, the current instruction will be of the type $\textsc{rwd}_b$, and the current rewinding position will be b.

$$\underline{\mathit{cur\_rewind\_pos}}(\sigma, a) \overset{\text{def}}{=} \begin{cases} b & \text{if } \sigma^a.\mathit{event} = \textsc{rwd}_b \text{ for some } b \in [n], \\ a & \text{otherwise.} \end{cases}$$

The predicate $\mathit{points\_here}(\sigma, a, q)$ holds when q points to the node labeled with σ when the lace is at (σ, a). This occurs when an instruction of the type $\langle q := \textbf{here} \rangle$ is found in σ at position at most a, and all upd_q flags from that position to position a are false.

$$\underline{\mathit{points\_here}}(\sigma, a, q) \overset{\text{def}}{=} \bigvee_{i \in [2,a]} \Big(\sigma^i.\mathit{event} = \langle q := \textbf{here} \rangle \wedge \bigwedge_{j \in [i+1,a]} \neg\sigma^j.\mathit{upd}_q \Big)\,.$$

When the value of pointer q cannot be ascertained by the current label σ, we search for its most recent assignment by identifying the latest position in σ that comes before position a and contains the flag $upd_q = true$. This is the job of the function $last\_upd(\sigma, a, q)$. If no position in σ before a contains $upd_q = true$, this function returns 2, because in that case the rewinding operation must go back to the frame that led to the first visit to the current node.

$$\underline{last\_upd}(\sigma, a, q) \overset{\text{def}}{=} \begin{cases} \max\{i \in [2,a] \mid \sigma^i.upd_q = true\} & \text{if } \{i \in [2,a] \mid \sigma^i.upd_q = true\} \neq \emptyset \\ 2 & \text{otherwise.} \end{cases}$$

The last auxiliary function, $last\_visit(\tau, dir, j)$, returns the position of the last frame from τ whose $next$ field is of the form (dir, c) for some $c \le j$.

$$\underline{last\_visit}(\tau, dir, j) \overset{\text{def}}{=} \max\{i \in [n] \mid \tau^i.next = (dir, c), c \le j\}\,.$$

Searching for two pointers. We describe a variant of the predicate *rewind*, for the instructions that search for *two* different pointers at the same time. In fact, this only happens in the encoding of the Boolean condition $q_1 = q_2$.

$$\begin{aligned}
&\underline{rewind2}(\sigma, a, \tau, b, q_1, q_2) \overset{\text{def}}{=} \\
&\quad \sigma^a.next = (dir, b) \;\wedge\; \neg\sigma^a.isnil_{q_1} \;\wedge\; \neg\sigma^a.isnil_{q_2} \wedge \\
&\quad a' = cur\_rewind\_pos(\sigma, a) \wedge \\
&\quad \neg points\_here(\sigma, a', q_1) \;\wedge\; \neg points\_here(\sigma, a', q_2) \wedge \\
&\quad a'' = \max\big\{last\_upd(\sigma, a', q_1), last\_upd(\sigma, a', q_2)\big\} \wedge \\
&\quad \sigma^{a''}.prev = (dir, b') \;\wedge\; \tau^{b'}.next = (dir', a'') \wedge \\
&\quad b = last\_visit(\tau, dir', a') + 1 \;\wedge\; \tau^b.event = \mathrm{RWD}_{b'} \wedge \\
&\quad default_{avail,active,pc,val,d,isnil,active\_child}(\sigma^a, \tau^{b-1}, \tau^b)\,.
\end{aligned}$$

Next, the version of rewinding used by the second phase of the statement $p :=$ $q \to pfield$.

$$\begin{aligned}
&\underline{rewind\_special}(\sigma, a, \tau, b, r, a') \overset{\text{def}}{=} \\
&\quad \sigma^a.next = (dir, b) \;\wedge\; \neg\sigma^a.isnil_r \wedge \\
&\quad \neg points\_here(\sigma, a', r) \;\wedge\; a'' = last\_upd(\sigma, a', r) \wedge \\
&\quad \sigma^{a''}.prev = (dir, b') \;\wedge\; \tau^{b'}.next = (dir', a'') \wedge \\
&\quad b = last\_visit(\tau, dir', a') + 1 \;\wedge\; \tau^b.event = \mathrm{RWD}_{b',r} \wedge \\
&\quad default_{avail,active,pc,val,d,isnil,active\_child}(\sigma^a, \tau^{b-1}, \tau^b)\,.
\end{aligned}$$

The following predicates check whether the search for one or two pointers is finished because those variables point to the current node (i.e., the node with label σ):

$$\begin{aligned}
\underline{stop\_rewind}(\sigma, a, q) &\overset{\text{def}}{=} \neg\sigma^a.isnil_q \;\wedge\; a' = cur\_rewind\_pos(\sigma, a) \wedge \\
&\qquad points\_here(\sigma, a', q) \\
\underline{stop\_rewind2}(\sigma, a, q_1, q_2) &\overset{\text{def}}{=} stop\_rewind(\sigma, a, q_1) \vee stop\_rewind(\sigma, a, q_2).
\end{aligned}$$

The following predicate is used by the instructions that have already searched for two pointers and want to check whether they are equal.

$$\begin{aligned}&\underline{\mathit{are\_equal\_after\_rewind}}(\sigma, a, p, q) \overset{\mathsf{def}}{=}\\&\quad a' = \mathit{cur\_rewind\_pos}(\sigma, a) \,\wedge\, \mathit{points\_here}(\sigma, a', p) \wedge \mathit{points\_here}(\sigma, a', q)\,.\end{aligned}$$