Deciding Memory Safety for Single-Pass Heap-Manipulating Programs

UMANG MATHUR, University of Illinois, Urbana-Champaign, USA
ADITHYA MURALI, University of Illinois, Urbana-Champaign, USA
PAUL KROGMEIER, University of Illinois, Urbana-Champaign, USA
P. MADHUSUDAN, University of Illinois, Urbana-Champaign, USA
MAHESH VISWANATHAN, University of Illinois, Urbana-Champaign, USA

We investigate the decidability of automatic program verification for programs that manipulate heaps, and in particular, decision procedures for proving memory safety for them. We extend recent work that identified a decidable subclass of uninterpreted programs to a class of alias-aware programs that can update maps. We apply this theory to develop verification algorithms for memory safety— determining if a heap-manipulating program that allocates and frees memory locations and manipulates heap pointers does not dereference an unallocated memory location. We show that this problem is decidable when the initial allocated heap forms a forest data-structure and when programs are *streaming-coherent*, which intuitively restricts programs to make a single pass over a data-structure. Our experimental evaluation on a set of library routines that manipulate forest data-structures shows that common single-pass algorithms on data-structures often fall in the decidable class, and that our decision procedure is efficient in verifying them.

CCS Concepts: • **Theory of computation → Logic and verification**; *Automated reasoning*.

Additional Key Words and Phrases: Memory Safety, Program Verification, Aliasing, Decidability, Uninterpreted Programs, Streaming-Coherence, Forest Data-Structures

ACM Reference Format:
Umang Mathur, Adithya Murali, Paul Krogmeier, P. Madhusudan, and Mahesh Viswanathan. 2020. Deciding Memory Safety for Single-Pass Heap-Manipulating Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 35 (January 2020), 29 pages. https://doi.org/10.1145/3371103

1 INTRODUCTION

The problem of automatic (safety) verification is to ascertain whether a program satisfies its assertions on all inputs and on all executions. The standard technique for proving programs correct involves writing inductive invariants in terms of loop invariants and pre/post conditions, and proving the resulting verification conditions valid [Floyd 1967; Hoare 1969]. While there has been tremendous progress in identifying decidable fragments for checking validity of verification conditions (Nelson-Oppen combinations of decidable theories realized by efficient SMT solvers [Bradley

Authors' addresses: Umang Mathur, umathur3@illinois.edu, Department of Computer Science, University of Illinois, Urbana-Champaign, USA; Adithya Murali, adithya5@illinois.edu, Department of Computer Science, University of Illinois, Urbana-Champaign, USA; Paul Krogmeier, paulmk2@illinois.edu, Department of Computer Science, University of Illinois, Urbana-Champaign, USA; P. Madhusudan, madhu@illinois.edu, Department of Computer Science, University of Illinois, Urbana-Champaign, USA; Mahesh Viswanathan, vmahesh@illinois.edu, Department of Computer Science, University of Illinois, Urbana-Champaign, USA.

2475-1421/2020/1-ART35
https://doi.org/10.1145/3371103

and Manna 2007]), decidable program verification without annotations has been elusive. Apart from programs over finite domains, very few natural decidable classes are known.

In a recent paper [Mathur et al. 2019a], a class of *uninterpreted programs* was identified and shown to have a decidable verification problem. Uninterpreted programs work over arbitrary data domains; the domains give meaning to the constants, relations, and functions in the program, interpreting equality using its natural definition. A program is deemed correct only if it satisfies its assertions in all executions and for all data domains. The authors show that for a class of programs that satisfies a *coherence* condition, verification is decidable. The decision procedure relies on a streaming congruence closure algorithm realized as automata. The results of [Mathur et al. 2019a] are, however, purely theoretical; there are no general application domains identified where uninterpreted program verification would be useful, nor is there any implementation.

The goal of this paper is to study completely automated verification for *heap-manipulating programs*. When modeling the heap, we can treat pointers as *unary uninterpreted functions*; this is a natural modeling choice that does not involve any abstraction (as pointer fields are really arbitrary, unary functions). Thus, it is reasonable to hope that uninterpreted program verification techniques could be useful in this setting.

However, there is a fundamental challenge that we need to overcome: heap-manipulating programs *modify* heap pointers. With pointers modeled as functions, these programs modify functions/maps. It turns out that the theory of uninterpreted program verification developed in [Mathur et al. 2019a] is severely inadequate for tackling programs that modify maps.

In this paper, we undertake a fundamental study of verification for programs that have updatable maps/functions. We then apply this theory to show that checking memory safety of heap-manipulating programs (where one is given a recursively-described set of allocated locations and asked to check whether a program only dereferences locations that are within this set) is decidable for a subclass of programs, and evaluate the algorithms with an implementation and experiments.

1.1 Alias Awareness and the Notion of Congruence for Programs with Updatable Maps

We can think of uninterpreted programs as computating *terms* using the function and constants symbols appearing in the code—in the beginning, each variable stores a constant, and after executing an assignment statement of the form "$x := f(y)$", the term stored in x is $f(t_y)$, where t_y is the term stored in y before executing this statement. As the program executes, in addition to computing terms, it places equality and disequality constraints on the terms it computes (inherited from the conditionals in **if**-**then**-**else** and **while** constructs).

However, when the program updates maps, this fundamental property, that the program computes terms, is destroyed. Consider an example where there are two locations pointed to by variables x and y, but we do not know whether x and y point to the same location or not. If we update a pointer-field on x and then read the same pointer-field from y into a variable k, we cannot really associate any term with the variable k, as the term crucially depends on whether x and y were aliases to the same location or not.

The above is a fundamental problem that wreaks havoc, making program verification essentially impossible in the presence of updatable maps. The notion of coherence in [Mathur et al. 2019a] fails, as it crucially relies on this notion of terms. And it cannot be repaired, as the semantic meaning of the coherence condition fundamentally involves not computing the same term multiple times.

The primary observation, which saves the framework, is to note that the entire problem is due to not knowing aliasing relationships. We call a program execution *alias-aware* if, at every point where it updates a function on the element pointed to by a program variable x, the precise aliasing relationship between x and all other program variables in scope is *determined*. More precisely, when the execution modifies a function/pointer-field of x, for every other program variable z, it must either be the case that x is different from z in any data-model/heap or it must be the case that x is equal to z in all data-models/heaps.

We show that alias-awareness is a panacea for our problems. For alias-aware programs (programs whose executions are all alias-aware), we show we can associate terms with variables after a computation that updates maps, and further show that the notion of coherence extends naturally to programs that update maps. We then show that for coherent alias-aware programs, the verification problem becomes decidable. These results constitute the first main contribution of the paper.

1.2 Application to Verifying Memory Safety

We then study the application of our framework to verifying memory safety. Our key observation is that for programs that manipulate *forest data-structures* (data-structures consisting of disjoint tree-like structures), programs are naturally alias-aware. Intuitively, when traversing forest data-structures, aliasing information is implicitly present. For instance, if `x` points to a location of a forest data-structure, we know that the location pointed to by `x`, the one pointed to by the left child `x·left`, and the one pointed to by the right child `x·right` are all different.

In this paper, we define memory safety as follows. A heap-manipulating program starts with a set of allocated heap locations. During its execution, it dereferences pointers on heap locations, and allocates and frees locations. A program is memory safe if it never dereferences a location that is not in the allocated set. The above definition of memory safety captures the usual categories of memory safety errors such as null-pointer dereferences, use after free, use of uninitialized memory, illegal freeing of memory, etc. [Hicks 2014]. However, in this paper, we do not consider allocation of *contiguous blocks of arbitrary size* of memory (and hence do not handle arrays and buffer overflows of arrays in languages like C, etc.). Rather, we assume that allocation is done in terms of *records* of fixed size (like `structs` in C), and we disallow pointer arithmetic in our programs.

Our second main contribution of the paper is a technical result that gives an efficient decision procedure for verifying memory safety for a subclass of imperative heap-manipulating programs, whose *initial* allocated heaps are restricted to forest data-structures.

Handling forest data-structures, i.e., disjoint lists and trees, is useful as they are ubiquitous. Note that we require only the initial heaps to be forest data-structures; the program can execute for an arbitrarily long time and create cycles/merges as it manipulates the structures. We model the primitive types and operations on them using uninterpreted functions and relations, similar in spirit to the way the work in [Mathur et al. 2019a] handles all data. The key insight here is that this is a reasonable modeling choice, since programs typically do not rely on the semantics of the primitive data domains in order to assure memory safety (we also show this empirically in experiments). The salient aspect of our work is that we model the pointers in the heap *precisely* using updatable maps, without resorting to *any* abstraction (classical automatic analysis of heap programs typically uses abstractions for heaps; for instance, shape analysis involves an abstraction of the heap locations to a finite abstract domain [Sagiv et al. 1999]).

We allow the user to specify the initial allocated set as the (unbounded) set of locations reachable from various locations (pointed to by certain program variables) using particular pointer fields, until a specific set of locations is reached. The memory safety problem is then to check whether such a program, starting from an *arbitrary* heap storing a forest data-structure, an arbitrary model for the primitive types, and with the specified allocated set, dereferences only those locations that are in the (potentially changing) allocated set, on all executions.

The above problem turns out to be undecidable (a direct consequence from [Mathur et al. 2019a], as even programs that do not manipulate heaps and have simple equality assertions yield undecidability). The main result of the paper is that for a class of programs called *streaming-coherent* *programs*, memory safety is decidable. Intuitively, these correspond to programs that traverse the forest data-structures in a *single-pass*.

Technical Challenges. The primary challenge is dealing with *updatable pointers* and *updatable sets* (the latter are needed to model the set of allocated locations, which changes during program execution). As we show in our first set of results, it is crucial for a program to be alias aware. The fact that our initial data-structures are forests implicitly causes streaming-coherent programs to be alias aware. When two variables point to locations obtained using different traversals, we know they cannot alias to each other. Also, for streaming-coherent programs, we can keep track of whether traversals for any pair of variables are the same, and track their precise aliasing relationships.

The culmination of the ideas above is our result that verification of memory safety for streaming-coherent programs over forest data-structures is decidable, and is PSPACE-complete. It is in fact decidable in time that is linear in the size of the program and exponential in the number of variables. We also show that checking whether a given program is streaming-coherent is decidable in PSPACE. Note that even checking reachability in programs with Boolean domains has this complexity, and hence our algorithms are quite efficient.

Evaluation. We implement a prototype of our automata-based decision procedure. This involves intersecting an automaton that checks whether a given streaming-coherent execution is memory-safe with an automaton that represents the given program's executions. Instead of building these automata and checking emptiness of their intersection, we use an approach that constructs the automaton and the intersection on-the-fly, hence not paying the worst case costs upfront. We evaluate our procedure on a class of standard library functions that manipulate forest data-structures, including linked lists and trees, where various other aspects of the data-structures (such as keys, height, etc.) are modeled using an uninterpreted data domain. These are typically *single-pass* algorithms on such data-structures, that take pointers to forest data-structures as input (and may create non-forest data-structures during computation).

Though we have stringent requirements that programs must meet in order to be in the decidable class, we show in our experiments that most natural single-pass programs on forest data-structures meet our requirements. We also show that our tool is able to check if the program falls in the decidable class, and both verify memory safety and find memory safety errors extremely efficiently.

We emphasize that the novelty of our approach is in building *decision procedures* for verifying memory safety without the aid of human-given loop invariants, and without abstracting the heap domain (the data domain is, however, abstracted using uninterpreted functions). In contrast, there are several existing techniques that can prove memory safety *when given manually written loop invariants* or prove memory safety by *abstracting the heap* (which can lead to false positives). Our results hence carve out new ground in memory safety verification and our experiments show that our approach holds promise for wider applicability and scalability.

In summary, this paper makes the following contributions:

- A notion of alias-aware coherent programs (Section 3), and a result that shows that the assertion-checking problem for such programs is decidable and PSPACE-complete (Section 4).
- A notion of streaming-coherence and *forest data-structures* (Section 5) with an efficient decision procedure for verifying memory safety for the class of streaming-coherent programs that dynamically manipulate forest data-structures (Section 6).
- An efficient decision procedure determining if programs are streaming-coherent (Section 6).
- An experimental evaluation (Section 7) showing (a) common library routines that manipulate forest data-structures using single-pass traversals are often streaming-coherent, and (b) that the decision procedures presented in this paper (for checking whether programs satisfy the

streaming-coherent requirement and for checking whether streaming-coherent programs are memory safe) are very efficient, both for proving programs correct and finding errors in incorrect programs.

2 PRELIMINARIES

In this section, we define the syntax and semantics of programs that manipulate heaps and other relevant concepts useful for presenting the main results of the paper. These are fairly standard and familiar readers may skip the details.

2.1 Syntax of Heap-Manipulating Programs

The programs we consider are those that manipulate heaps. It is convenient to abstract heap structures as consisting of two sorts of distinct elements — a sort Loc of memory locations in the heap, and a sort Data of data values. *Field* (or map) symbols will model pointers from memory locations to memory locations or data values. Constants and functions over the data domain will be used to construct other data values. In this paper, we will not assume any fixed interpretation for either data values or for functions on data values. In this sense, our programs work over an *uninterpreted* data domain. Predicates over data values will be modeled by functions capturing the characteristic function of the predicate.

Let Loc and Data be the sorts of locations and data respectively. Our vocabulary Σ is a tuple of the form $(\mathcal{C}_{\mathsf{Loc}}, \mathcal{F}_{\mathsf{Loc}}, \mathcal{C}_{\mathsf{Data}}, \mathcal{F}_{\mathsf{Data}}, \mathcal{F}_{\mathsf{Loc}\to\mathsf{Data}})$, where

- $\mathcal{C}_{\mathsf{Loc}}$ is a set of location constant symbols of sort Loc,
- $\mathcal{F}_{\mathsf{Loc}}$ is a set of unary location function symbols with sort 'Loc, Loc' [1], that models pointers between heap locations,
- $\mathcal{C}_{\mathsf{Data}}$ denotes the set of data constant symbols of sort Data,
- $\mathcal{F}_{\mathsf{Data}} = \bigcup_{i\geq 0} F_i$ where F_r is a set of data function symbols of arity r and sort 'Data^r, Data', and
- $\mathcal{F}_{\mathsf{Loc}\to\mathsf{Data}}$ is a set of unary location function symbols with sort 'Loc, Data', modeling pointers to data values stored in heap locations.

2.1.1 Program Syntax. Programs will use a finite set of variables to store information — heap locations and data values — during a computation. Let us fix $V_{\mathsf{Loc}} = \{u_1, \ldots, u_l\}$ as the set of location variables and $V_{\mathsf{Data}} = \{v_1, \ldots, v_m\}$ as the set of data variables and let $V = V_{\mathsf{Loc}} \uplus V_{\mathsf{Data}}$ [2] be the set of all variables. In addition, our programs manipulate fields associated with location variables. We will model these fields as second order function variables $\mathsf{Flds}_{\mathsf{Loc}} = \{p_1, \ldots, p_r\}$ (pointers from locations to locations) and $\mathsf{Flds}_{\mathsf{Data}} = \{d_1, \ldots, d_s\}$ (pointers from locations to data), and let $\mathsf{Flds} = \mathsf{Flds}_{\mathsf{Loc}} \cup \mathsf{Flds}_{\mathsf{Data}}$. Taking $x, y \in V_{\mathsf{Loc}}, p \in \mathsf{Flds}_{\mathsf{Loc}}, d \in \mathsf{Flds}_{\mathsf{Data}}, f$ in $\mathcal{F}_{\mathsf{Data}}$, $a, b \in V_{\mathsf{Data}}$, and $\mathbf{c}$ to be a tuple of variables in V_{Data}, the syntax of programs is given by the following grammar.

$$\begin{aligned}
\langle stmt\rangle ::=\ & \mathbf{skip} \mid x := y \mid x := y.p \mid y.p := x \mid a := y.d \mid y.d := a \mid \mathbf{alloc}(x) \mid \mathbf{free}(x) \\
& \mid a := b \mid a := f(\mathbf{c}) \mid \mathbf{assume}\,(\langle cond\rangle) \mid \langle stmt\rangle\,;\,\langle stmt\rangle \\
& \mid \mathbf{if}\,(\langle cond\rangle)\,\mathbf{then}\,\langle stmt\rangle\,\mathbf{else}\,\langle stmt\rangle \mid \mathbf{while}\,(\langle cond\rangle)\,\langle stmt\rangle \\
\langle cond\rangle ::=\ & x = y \mid a = b \mid \langle cond\rangle \vee \langle cond\rangle \mid \neg\langle cond\rangle
\end{aligned}$$

Our programs have well-typed assignments to variables using values stored in other variables ($x := y$ and $a := b$) or using pointer dereferences from location variables, either to the data

[1] We will use the notation σ, τ to indicate a function whose arguments are from sort σ and which returns a value in sort τ. Thus for example 'Loc, Loc' is a function with one argument of sort Loc and which returns an element of sort Loc. On the other hand, 'Data^r, Data' denotes functions with r arguments each of sort Data and which returns an element of sort Data.

[2] We use $A \uplus B$ to denote the disjoint union of sets A and B.

sort ($a := y{\cdot}d$) or to the location sort ($x := y{\cdot}p$), or using function computations in the data sort ($a := f(\mathbf{c})$). Further, programs can update fields ($y{\cdot}d := a$ or $y{\cdot}p := x$), and can dynamically allocate (**alloc**(x)) or deallocate (**free**(x)) memory. In addition, they allow the usual constructs of imperative programming — empty statements (**skip**), conditionals (**if** – **then** – **else**) and loops (**while**). Conditionals in programs can be Boolean combinations of (well-typed) equality atoms over location or data variables.

2.2 Program Executions

Executions of programs over Σ and variables V (given by the $\langle stmt\rangle$ grammar) are finite sequences over the alphabet Π given below.

$$\begin{aligned}\Pi = \{&\text{“}x := y\text{”}, \text{“}x := y{\cdot}p\text{”}, \text{“}y{\cdot}p := x\text{”}, \text{“}a := y{\cdot}d\text{”}, \text{“}y{\cdot}d := a\text{”}, \text{“}\mathbf{alloc}(x)\text{”}, \text{“}\mathbf{free}(x)\text{”}, \text{“}a := b\text{”},\\ &\text{“}a := f(\mathbf{c})\text{”}, \text{“}\mathbf{assume}(x = y)\text{”}, \text{“}\mathbf{assume}(x \neq y)\text{”}, \text{“}\mathbf{assume}(a = b)\text{”}, \text{“}\mathbf{assume}(a \neq b)\text{”}\\ &\mid x, y \in V_{\mathsf{Loc}}, p \in \mathsf{Flds}_{\mathsf{Loc}}, d \in \mathsf{Flds}_{\mathsf{Data}}, f \in \mathcal{F}_{\mathsf{Data}} a, b \in V_{\mathsf{Data}}, \text{ and } \mathbf{c} \text{ a tuple over } V_{\mathsf{Data}}\}.\end{aligned}$$

The set of executions of a program P, denoted $\mathsf{Exec}(P)$ is given by a regular expression, inductively defined below. We assume that conditionals are in negation normal form where “**assume**($\neg(r = s)$)” translates to “**assume**($r \neq s$)” and “**assume**($\neg(r \neq s)$)” translates to “**assume**($r = s$)”.

$$\begin{array}{rcll}
\mathsf{Exec}(\mathbf{skip}) & = & \epsilon & \\
\mathsf{Exec}(a) & = & a & \text{if } a \in \Pi \\
\mathsf{Exec}(\mathbf{assume}(c_1 \vee c_2)) & = & \mathsf{Exec}(\mathbf{assume}(c_1)) + \mathsf{Exec}(\mathbf{assume}(c_2)) & c_1, c_2 \in \langle cond\rangle \\
\mathsf{Exec}(\mathbf{assume}(c_1 \wedge c_2)) & = & \mathsf{Exec}(\mathbf{assume}(c_1)) \cdot \mathsf{Exec}(\mathbf{assume}(c_2)) & c_1, c_2 \in \langle cond\rangle \\
\mathsf{Exec}(\mathbf{if}\,(c)\,\mathbf{then}\,s_1\,\mathbf{else}\,s_2) & = & \begin{array}{l}\mathsf{Exec}(\mathbf{assume}(c)) \cdot \mathsf{Exec}(s_1)\\ +\ \mathsf{Exec}(\mathbf{assume}(\neg c)) \cdot \mathsf{Exec}(s_2)\end{array} & \begin{array}{l}c \in \langle cond\rangle,\\ s_1, s_2 \in \langle stmt\rangle\end{array} \\
\mathsf{Exec}(\mathbf{while}\,(c)\,s) & = & \begin{array}{l}(\mathsf{Exec}(\mathbf{assume}(c)) \cdot \mathsf{Exec}(s))^*\\ \cdot\ \mathsf{Exec}(\mathbf{assume}(\neg c))\end{array} & \begin{array}{l}c \in \langle cond\rangle,\\ s \in \langle stmt\rangle\end{array} \\
\mathsf{Exec}(s_1; s_2) & = & \mathsf{Exec}(s_1) \cdot \mathsf{Exec}(s_2) & s_1, s_2 \in \langle stmt\rangle
\end{array}$$

The set of partial executions of a program P, denoted $\mathsf{PExec}(P)$, is the set of prefixes of its executions.

2.3 Semantics of Executions

The semantics of a heap-manipulating program is given in terms of the behavior of its executions on *heap structures.*

2.3.1 *Heap Structures.*

A Σ-heap structure is a tuple $\mathcal{M} = (U_{\mathsf{Loc}}, U_{\mathsf{Data}}, \mathcal{I})$, where U_{Loc} is a universe of locations, U_{Data} is a universe of data ($U_{\mathsf{Loc}} \cap U_{\mathsf{Data}} = \emptyset$) and $\mathcal{I}$ is some interpretation of the various symbols in Σ. In order to faithfully model dynamic memory allocation, we assume that the set of locations is the disjoint union of a statically allocated set of locations and a *countably infinite* set of locations that can be allocated dynamically. That is, we have $U_{\mathsf{Loc}} = U^{\mathsf{static}}_{\mathsf{Loc}} \uplus U^{\mathsf{dynamic}}_{\mathsf{Loc}}$, where $U^{\mathsf{dynamic}}_{\mathsf{Loc}} = \{e_0, e_1, \ldots\}$ is an ordered set of distinguished locations indexed by the set of natural numbers $\mathbb{N}$. The interpretation $\mathcal{I}$ maps every constant $c \in C_{\mathsf{Loc}}$ to an element from $U^{\mathsf{static}}_{\mathsf{Loc}}$, every constant in C_{Data} to an element from U_{Data}, every function symbol $f \in \mathcal{F}_{\mathsf{Loc}}$ to an element of $[U_{\mathsf{Loc}} \to U_{\mathsf{Loc}}]$, symbol $f \in \mathcal{F}_{\mathsf{Data}}$ of arity r to an element of $[(U_{\mathsf{Data}})^r \to U_{\mathsf{Data}}]$, and, $f \in \mathcal{F}_{\mathsf{Loc}\to\mathsf{Data}}$ to an element of $[U_{\mathsf{Loc}} \to U_{\mathsf{Data}}]$. Further, we assume that the elements in $U^{\mathsf{dynamic}}_{\mathsf{Loc}}$ cannot be accessed from $U^{\mathsf{static}}_{\mathsf{Loc}}$, i.e., for every $f \in \mathcal{F}_{\mathsf{Loc}}$, we have $\forall e \in U^{\mathsf{static}}_{\mathsf{Loc}}, \mathcal{I}(f)(e) \in U^{\mathsf{static}}_{\mathsf{Loc}}$. This ensures that the set of locations reachable by any execution cannot access locations in the dynamic universe that have not been allocated by the execution. Moreover, for every $f \in \mathcal{F}_{\mathsf{Loc}}$ and every $e \in U^{\mathsf{dynamic}}_{\mathsf{Loc}}$, we have $f(e) = e$. Finally, we assume in Σ the presence of a distinguished constant symbol $\mathsf{c}_{\mathsf{dynamic}}$ and a distinguished function symbol $\mathsf{f}_{\mathsf{dynamic}}$ not used in the syntax

$\langle stmt \rangle$ of programs. We require that for every heap structure $\mathcal{M} = (U^{\mathsf{static}}_{\mathsf{Loc}} \uplus U^{\mathsf{dynamic}}_{\mathsf{Loc}}, U_{\mathsf{Data}}, \mathcal{I})$ the interpretation function $\mathcal{I}$ of $\mathcal{M}$ assigns interpretations to these symbols such that $\mathcal{I}(\mathsf{c}_{\mathsf{dynamic}}) = e_0$ and $\mathcal{I}(\mathsf{f}_{\mathsf{dynamic}}) \in [U^{\mathsf{dynamic}}_{\mathsf{Loc}} \to U^{\mathsf{dynamic}}_{\mathsf{Loc}}]$ with $\mathcal{I}(\mathsf{f}_{\mathsf{dynamic}})(e_i) = e_{i+1}$ for every $i \in \mathbb{N}$.

2.3.2 Valuation of Variables and Pointers in an Execution. We assume that corresponding to each program variable $x \in V$, there is a distinguished constant $\widehat{x} \in C_{\mathsf{Loc}} \uplus C_{\mathsf{Data}}$ of the appropriate sort denoting the initial value of x. Likewise, each field $p \in \mathsf{Flds}_{\mathsf{Loc}}$ (resp. $d \in \mathsf{Flds}_{\mathsf{Data}}$) is also associated with a unary function $\widehat{p} \in \mathcal{F}_{\mathsf{Loc}}$ (resp. $\widehat{d} \in \mathcal{F}_{\mathsf{Loc} \to \mathsf{Data}}$).

Given an execution $\sigma \in \Pi^*$ of a program $P \in \langle stmt \rangle$ and a Σ-heap structure $\mathcal{M} = (U_{\mathsf{Loc}}, U_{\mathsf{Data}}, \mathcal{I})$, the valuation of the program variables and field pointers at the end of σ are defined in terms of valuation functions $\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}} : \Pi^* \times V_{\mathsf{Loc}} \to U_{\mathsf{Loc}}$, $\mathsf{Val}^{\mathcal{M}}_{\mathsf{Data}} : \Pi^* \times V_{\mathsf{Data}} \to U_{\mathsf{Data}}$, $\mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{Loc}} : \Pi^* \times \mathsf{Flds}_{\mathsf{Loc}} \to [U_{\mathsf{Loc}} \to U_{\mathsf{Loc}}]$ and $\mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{Data}} : \Pi^* \times \mathsf{Flds}_{\mathsf{Data}} \to [U_{\mathsf{Loc}} \to U_{\mathsf{Data}}]$, which are presented next. In the following, $\mathsf{allocations}(\sigma)$ denotes the number of occurrences of statements of the form "**alloc**$(\cdot)$" in σ.

$$
\begin{array}{rcl}
\mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\varepsilon, u) & = & \mathcal{I}(\widehat{u}) \\
\mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\sigma \cdot s, u) & = & \begin{cases}
\mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, y) & \text{if } s = \text{“}x := y\text{” and } u = x \\
e_i & \text{if } s = \text{“}\mathbf{alloc}(x)\text{”}, \\
 & i = \mathsf{allocations}(\sigma) \text{ and } u = x \\
\mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, p)(\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, y)) & \text{if } s = \text{“}x := y{\cdot}p\text{” and } u = x \\
\mathcal{I}(f)(\mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, c_1), \ldots, \mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, c_r)) & \text{if } s = \text{“}a := f(c_1, \ldots, c_r)\text{”} \\
 & \text{ and } u = a \\
\mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, u) & \text{otherwise}
\end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{sort}}(\varepsilon, p) & = & \mathcal{I}(\widehat{p}) \\
\mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{sort}}(\sigma \cdot s, p) & = & \begin{cases}
\mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, q)[\mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, y) \mapsto \mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, x)] & \text{if } s = \text{“}y{\cdot}q := x\text{”} \\
 & \text{and } p = q \\
\mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{sort}}(\sigma, p) & \text{otherwise}
\end{cases}
\end{array}
$$

where by $f[a \mapsto b]$ we mean the function g defined as $g(a) = b$ and $g(x) = f(x)$ otherwise, and the sort symbols are one of $\{\mathsf{Loc}, \mathsf{Data}\}$, appropriately determined according to the program statement. The expanded definition is provided in the Technical Report [Mathur et al. 2019c].

Feasibility. An execution σ is said to be **feasible** on $\mathcal{M}$ if for every prefix of σ of the form $\rho' \cdot$ "**assume**$(x = y)$", we have $\mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\rho', x) = \mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\rho', y)$, and for every prefix of σ of the form $\rho' \cdot$ "**assume**$(x \neq y)$", we have $\mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\rho', x) \neq \mathsf{Val}^{\mathcal{M}}_{\mathsf{sort}}(\rho', y)$, where $\mathsf{sort} \in \{\mathsf{Loc}, \mathsf{Data}\}$ is the sort of both x and y.

3 DEFINING COHERENCE FOR HEAP-MANIPULATING PROGRAMS

In this section we discuss some of the challenges involved in coming up with a reasonable extension for the notion of coherence as defined in [Mathur et al. 2019a]. A key problem in defining such an extension, and indeed in generally handling programs with updatable maps, lies in keeping accurate track of *aliasing* between variables of the location sort. We will first discuss some examples that highlight these challenges and proffer a first solution to the problem. Then we will discuss our solution more formally by introducing relevant notation and define our notion of *alias-aware* executions and programs that captures the essence of the aliasing problem. Finally, we will discuss the notion of coherence adapted to the case of heap-manipulating programs.

3.1 The Importance of Being Alias Aware for Programs Updating Maps

Functions and updatable maps cannot be handled uniformly; in particular, the work of [Mathur et al. 2019a] does not immediately lend itself to handling updatable maps. Let us illustrate this using the following example.

Example 1. Consider a straight-line program that generates execution π_1, where

$$\pi_1 \overset{\Delta}{=} \mathtt{z}_1 := \mathtt{x}\cdot\mathtt{next} \cdot \ \mathbf{assume}(\mathtt{z}_1 \neq \mathtt{z}_2) \cdot \ \mathtt{y}\cdot\mathtt{next} := \mathtt{z}_2 \cdot \ \mathtt{z}_3 := \mathtt{x}\cdot\mathtt{next}$$

In the execution above we do not, and in fact cannot, know the value stored in the variable $\mathtt{z}_3$ unless we know whether $\mathtt{x} = \mathtt{y}$ or $\mathtt{x} \neq \mathtt{y}$. In the former case, we will have that $\mathtt{z}_3 = \mathtt{z}_2$, since y *aliases* x and, therefore, the update of the next pointer on y will have over-written the value of the pointer on x. Similarly, in the latter case we will have that $\mathtt{z}_3 = \mathtt{z}_1$, since the pointer update on y will have no effect on x.

This is a major difference, since in any heap structure on which π_1 is feasible, it must be that $\mathtt{z}_1 \neq \mathtt{z}_2$. Therefore, whether x and y are aliases of each other can have a drastic effect on the semantics and feasibility of the execution.

The above example illustrates that aliasing plays a crucial role in the semantics and feasibility of executions. There is, however, a second (related) issue as well.

Example 2. Consider the executions π_2 and π_3 where

$$\pi_2 \overset{\Delta}{=} \pi_1 \cdot \mathbf{assume}(\mathtt{z}_2 = \mathtt{z}_3) \qquad \pi_3 \overset{\Delta}{=} \pi_1 \cdot \mathbf{assume}(\mathtt{z}_2 \neq \mathtt{z}_3)$$

As discussed earlier, execution π_2 is only feasible in models where $\mathtt{x} = \mathtt{y}$ and π_3 only in models where $\mathtt{x} \neq \mathtt{y}$ where at the end of π_1 both kinds of models were feasible. Therefore one can have *implied* equalities and disequalities between variables that are only known much later in the execution. In general, this is hard to keep track of in a streaming setting (which we wish to do in order to use the underlying ideas in [Mathur et al. 2019a]) and can require an unbounded amount of memory, as can be seen in the following example.

Example 3. Consider the following execution

$$\begin{aligned}\pi_4' \overset{\Delta}{=} & \mathbf{assume}(\mathtt{x} \neq \mathtt{NIL}) \cdot \mathbf{assume}(\mathtt{y} \neq \mathtt{NIL}) \cdot \mathtt{z}_1 := \mathtt{x}\cdot\mathtt{next} \cdot \mathbf{assume}(\mathtt{z}_1 \neq \mathtt{z}_2) \cdot \mathtt{y}\cdot\mathtt{next} := \mathtt{z}_2 \\ & \cdot \mathtt{z}_3 := \mathtt{x}\cdot\mathtt{next} \cdot \mathtt{k}_1 := \mathtt{x}\cdot\mathtt{key} \cdot \mathtt{k}_2 := \mathtt{y}\cdot\mathtt{key} \cdot \underbrace{\mathtt{k}_1 := \mathtt{f}(\mathtt{k}_1) \cdot \mathtt{k}_2 := \mathtt{f}(\mathtt{k}_2) \cdots \mathtt{k}_1 := \mathtt{f}(\mathtt{k}_1) \cdot \mathtt{k}_2 := \mathtt{f}(\mathtt{k}_2)}_{n} \\ & \cdot \mathbf{assume}(\mathtt{z}_2 = \mathtt{z}_3) \cdot \mathbf{assume}(\mathtt{k}_1 = \mathtt{k}_2)\end{aligned}$$

The above execution is feasible, but would require an unbounded amount of memory to reason as such in a streaming fashion. Next we will see a solution to this aliasing problem that will allow us to keep track of relationships between variables and the correct semantics of an execution.

3.2 Alias-Aware Executions and Programs

In Section 3.1, we saw that unlike the work of [Mathur et al. 2019a] we cannot define what term (or value) is computed by a variable. We also observed that the main issue with programs that have updatable maps is *aliasing* — i.e., when a pointer or data field is updated on a variable, the update may also be true for a different variable that *aliases* the original variable. In this section we identify a class of executions, called alias-aware executions, which implicitly resolves any aliasing when updating the pointer fields on heap locations.

Below, we formally define alias-aware executions. We denote by $\mathsf{Heaps}(\sigma)$ the set of heap structures on which the execution σ is feasible.

Definition 1. Let $\rho \in \Pi^*$ be an execution. ρ is said to be alias-aware if for every prefix of ρ of the form $\sigma \cdot$ "$x{\cdot}h := u$" (the sorts of u and h are compatible), and for every location variable $y \in V_{\mathsf{Loc}}$, one of the following hold

- For every heap structure $\mathcal{M} = (U_{\mathsf{Loc}}, U_{\mathsf{Data}}, \mathcal{I}) \in \mathsf{Heaps}(\sigma)$, $\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, x) = \mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, y)$.
- For every heap structure $\mathcal{M} = (U_{\mathsf{Loc}}, U_{\mathsf{Data}}, \mathcal{I}) \in \mathsf{Heaps}(\sigma)$, $\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, x) \neq \mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, y)$.

Intuitively, the above definition says that for an alias-aware execution, at a map update we know all the relevant aliasing information (even in the uninterpreted sense), because the variables that alias to the variable on which the update is being performed are the same in *every* feasible model.

Definition 2 (Alias-Aware programs)**.** A program $s \in \langle stmt \rangle$ is said to be alias-aware if every execution $\rho \in \mathsf{Exec}(s)$ is alias-aware.

3.3 Term Computed by a Variable during an Execution

Before we define coherence in our setting, we require the notion of *terms* that an execution computes. The notion of terms lets us reason about infinitely many heap structures in a symbolic fashion and was crucially exploited in [Mathur et al. 2019a] to define coherence and in arguing the correctness of the decision procedure designed for coherent programs.

We would like to define the term associated with a program variable in an execution. There are two major challenges in our way. The first challenge is that, in order to accurately determine the *value* pointed to by a variable during an execution on a concrete heap structure, one needs to accurately keep track of the interpretations of pointer fields (that could have, in turn, been updated earlier in the execution). A similar problem needs to be addressed in order to successfully define the notion of terms. The second challenge is that unlike in concrete heap structures where two distinct elements in the heap are known to be unequal, terms do not behave the same way. For the case of terms, we would like to explore the possibility that two terms that might be syntactically different might still be semantically *equivalent.* While this subtlety can be dealt with easily when none of the functions are updatable, as in [Mathur et al. 2019a], greater care is required in our setting. Let us first formalize some notation and then elaborate these subtleties.

For a set C of constant symbols and a set $\mathcal{F}$ of function symbols from Σ, we use $\mathsf{Terms}(C, \mathcal{F})$ (or just Terms when the signature is clear) to denote the set of (well-sorted) ground terms constructed using the constants in C and function symbols in $\mathcal{F}$.

For a binary relation $R \subseteq \mathsf{Terms} \times \mathsf{Terms}$, the *congruence closure* of R, denoted $\cong_R$ is the smallest equivalence relation such that (i) $R \subseteq \cong_R$, and (ii) for every function f of sort $w_1 w_2 \cdots w_r, w$ ($w, w_i \in \{\mathsf{Loc}, \mathsf{Data}\}$) and terms $t_1, t_1', t_2, t_2', \ldots, t_r, t_r'$ of sorts $w_1, w_1, w_2, w_2, \ldots, w_r, w_r$, we have $\left(\bigwedge\limits_{i=1}^{r} (t_i, t_i') \in \cong_R \right) \implies (f(t_1, \ldots, t_r), f(t_1', \ldots, t_r')) \in \cong_R$. We say $t_1 \cong_R t_2$ if $(t_1, t_2) \in \cong_R$, where t_1, t_2 are terms.

We now define the terms associated with variables $\mathsf{Comp} : \Pi^* \times V \to \mathsf{Terms}$, the interpretations of updatable maps on terms corresponding to variables $\mathsf{FldsComp} : \Pi^* \times \mathsf{Flds} \times V \to \mathsf{Terms}$, and the set of equalities accumulated by the execution $\alpha : \Pi^* \to \mathcal{P}(\mathsf{Terms} \times \mathsf{Terms})$. Recall that, in the following, $\mathsf{c}_{\mathsf{dynamic}} \in C_{\mathsf{Loc}}$ and $\mathsf{f}_{\mathsf{dynamic}} \in \mathcal{F}_{\mathsf{Loc}}$ are special symbols in our vocabulary.

$$
\begin{aligned}
\mathsf{Comp}(\varepsilon, u) &= \widehat{u} \\
\mathsf{Comp}(\sigma \cdot s, u) &= \begin{cases}
\mathsf{Comp}(\sigma, y) & \text{if } s = \text{“}x := y\text{” and } u = x \\
\mathsf{f}^{i}_{\mathsf{dynamic}}(\mathsf{c}_{\mathsf{dynamic}}) & \text{if } s = \text{“}\textbf{alloc}(x)\text{”}, \\
 & i = \text{allocations}(\sigma), \text{ and } u = x \\
\mathsf{FldsComp}(\sigma, h, y) & \text{if } s = \text{“}x := y{\cdot}h\text{” and } u = x \\
f(\mathsf{Comp}(\sigma, z_1), \ldots, \mathsf{Comp}(\sigma, z_r)) & \text{if } s = \text{“}x := f(z_1, \ldots, z_r)\text{” and } u = x \\
\mathsf{Comp}(\sigma, u) & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{FldsComp}(\varepsilon, h, z) &= \widehat{h}(\widehat{z}) \\
\mathsf{FldsComp}(\sigma \cdot s, h, z) &= \begin{cases}
\mathsf{Comp}(\sigma, x) & \text{if } s = \text{“}y{\cdot}h := x\text{”} \\
 & \text{and } \mathsf{Comp}(\sigma, z) \cong_{\alpha(\sigma)} \mathsf{Comp}(\sigma, y) \\
\mathsf{FldsComp}(\sigma, h, z) & \text{otherwise}
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\alpha(\varepsilon) &= \emptyset \\
\alpha(\sigma \cdot s) &= \begin{cases}
\alpha(\sigma) \cup \{(\mathsf{Comp}(\sigma, x), \mathsf{Comp}(\sigma, y))\} & \text{if } s = \text{“}\textbf{assume}(x = y)\text{”} \\
\alpha(\sigma) & \text{otherwise}
\end{cases}
\end{aligned}
$$

The set of terms computed by an execution σ is the set $\mathsf{Terms}(\sigma) = \{\mathsf{Comp}(\rho, x) \mid x \in V, \rho \text{ is a prefix of } \sigma\}$.

Let us discuss some aspects of the above definition here. The effect of some of the commands in the executions is obvious and similar to [Mathur et al. 2019a]. At the beginning of an execution, each term $x \in V$ is associated with a unique constant term $\widehat{x}$, each pointer h assigns every term t (of the location sort) to $\widehat{h}(t)$, and the set of equalities accumulated is $\emptyset$. On an assignment "$x := y$", the term stored in x is updated to be that stored in y. On an assignment involving a (non-updatable) function ($x := f(z_1, \ldots, z_r)$), the execution computes the term $f(t_1, \ldots, t_r)$, and stores it in x, where t_i is the term associated with z_i before the assignment.

Let us now look at the other commands. Recall that we model our set of dynamically allocated locations as a disjoint set from the set of statically allocated locations. The term associated with x after an "**alloc**(x)" follows this premise. It is built using n applications of a distinct function $\mathsf{f}_{\mathsf{dynamic}} \in \mathcal{F}_{\mathsf{Loc}}$ on a distinct constant $\mathsf{c}_{\mathsf{dynamic}} \in \mathcal{C}_{\mathsf{Loc}}$, where n is the number of allocations performed before this point in the execution. Let us now turn to the most subtle aspect: *pointer updates*. When the execution encounters a command of the form "$y{\cdot}h := x$", we need to not only update the term pointed to by $t_y{\cdot}h$ (where t_y is the term associated with y), we also need to update the term pointed to by $t{\cdot}h$, when t can be inferred to be equivalent to t_y using the equalities $\alpha(\cdot)$ accumulated so far.

Example 4. Consider the following execution.

$$\pi_5 \overset{\Delta}{=} \textbf{assume}(\mathsf{x} = \mathsf{y}) \cdot \mathsf{x}{\cdot}\mathsf{p} := \mathsf{z}_1 \cdot \mathsf{z}_2 := \mathsf{y}{\cdot}\mathsf{p} \cdot \textbf{assume}(\mathsf{z}_1 \neq \mathsf{z}_2)$$

Here, the terms associated with the pointers $x{\cdot}p$ and $y{\cdot}p$ are $\widehat{p}(\widehat{x})$ and $\widehat{p}(\widehat{y})$ respectively. After the "**assume**($\mathsf{x} = \mathsf{y}$)", the locations pointed to by x and y are the same in every heap structure on which this assumption is valid, and thus the terms corresponding to x and y must be deemed *equivalent*. This means that the pointer update $\mathsf{x}{\cdot}\mathsf{p} := \mathsf{z}_1$ should in fact be reflected in the term associated with the pointer $\mathsf{y}{\cdot}\mathsf{p}$. The above definition of $\mathsf{Comp}(\cdot)$ and $\mathsf{FldsComp}(\cdot)$ will, in fact, ensure that $\mathsf{Comp}(\pi_5, \mathsf{z}_1) = \mathsf{Comp}(\pi_5, \mathsf{z}_2)$. Not doing so will, in turn, not reflect the contradiction due to the last statement of this execution, which makes it infeasible in every heap structure.

It turns out that the above definition of terms associated with variables is only a *best effort*, in that, it may not accurately summarize all heap structures in which the execution is feasible. The following example illustrates why this is the case.

Example 5. Let us consider the following, which is a permutation of the execution in Example 4.

$$\pi_6 \overset{\Delta}{=} \mathsf{x}.\mathsf{p} := \mathsf{z}_1 \cdot \mathsf{z}_2 := \mathsf{y}.\mathsf{p} \cdot \textbf{assume}(\mathsf{x} = \mathsf{y}) \cdot \textbf{assume}(\mathsf{z}_1 \neq \mathsf{z}_2)$$

Similar to the execution in Example 4, the execution π_6 is infeasible in every heap structure. However, in this case, the terms associated with z_1 and z_2 will be $\mathsf{Comp}(\pi_6, \mathsf{z}_1) = \widehat{\mathsf{z}_1}$ and $\mathsf{Comp}(\pi_6, \mathsf{z}_2) = \widehat{\mathsf{p}}(\widehat{\mathsf{y}})$, which are not equal, even when considering the equivalence induced due to $\alpha(\pi_6) = \{(\widehat{\mathsf{x}}, \widehat{\mathsf{y}})\}$. As a result, it is hard to conclude that π_6 is an infeasible execution by analyzing the terms associated with variables alone.

However, for the case of an alias-aware execution (Definition 1), the definition of terms above indeed accurately relates the terms associated with each variable to their values in every heap structure that makes the execution feasible. This brings us to the first main result of the paper that motivates the definition of alias-aware executions. It simply states that the interpretation of the terms defined above on a given model coincides with the actual values computed on the model.

Theorem 1. *Let σ be an alias-aware execution and let $\mathcal{M} = (U_{\mathsf{Loc}}, U_{\mathsf{Data}}, \mathcal{I}) \in Heaps(\sigma)$ be a heap structure on which σ is feasible. Then, $\mathcal{I}(Comp(\sigma, x)) = \mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, x)$ for every variable $x \in V_{\mathsf{Loc}}$ and $\mathcal{I}(Comp(\sigma, a)) = \mathsf{Val}^{\mathcal{M}}_{\mathsf{Data}}(\sigma, a)$ for every $a \in V_{\mathsf{Data}}$.*

Moreover, $\mathcal{I}(\mathsf{Flds}Comp(\sigma, h, x)) = \mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, h)(\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, x))$ and $\mathcal{I}(\mathsf{Flds}Comp(\sigma, d, x)) = \mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{Data}}(\sigma, d)(\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, x))$ for every $h \in \mathsf{Flds}_{\mathsf{Loc}}, d \in \mathsf{Flds}_{\mathsf{Data}}$ and $x \in V_{\mathsf{Loc}}$.

A gist of the proof of this theorem can be found in the Technical Report [Mathur et al. 2019c].

3.4 Coherent Programs

Having defined the concept of terms associated with variables in an execution, we can now extend the notion of *coherence*, adapting it from [Mathur et al. 2019a] to accommodate updatable pointers as in the heap-manipulating programs that we study.

In the following, we say that s is a superterm of t modulo some congruence relation $\cong$ if there are terms s' and t' such that $t \cong t'$, s' is a superterm of t' and $s' \cong s$.

Definition 3 (Coherence)**.** A complete or partial execution σ is said to be a coherent execution if it satisfies the following two conditions.

Memoizing. Let ρ be a prefix of σ such that it is of the form $\rho' \cdot$ "$x := y.h$" or $\rho' \cdot$ "$x := f(\mathrm{y})$" and let $t_x = \mathsf{Comp}(\rho, x)$. If there is a term $t' \in \mathsf{Terms}(\rho')$ such that $t' \cong_{\alpha(\rho')} t_x$, then it must be the case that there is some variable $z \in V$ such that $\mathsf{Comp}(\rho', z) \cong_{\alpha(\rho')} t_x$.

Early Assumes. Let ρ be a prefix of the form $\rho' \cdot$ "**assume**$(x = y)$" and let $t_x = \mathsf{Comp}(\rho', x), t_y = \mathsf{Comp}(\rho', y)$. If there is a term $t \in \mathsf{Terms}(\rho')$ such that t is a superterm of t_x or t_y modulo $\cong_{\alpha(\rho')}$, then there must be some variable $w \in V$ such that $\mathsf{Comp}(\rho', w) \cong_{\alpha(\rho')} t$.

Note that these conditions are applicable to every sort-sensible combination of symbols.

As mentioned earlier, this definition of coherence is inspired from the notion of coherence defined previously in [Mathur et al. 2019a]. The *memoizing* restriction is the heart of the notion of coherent executions. Let us illustrate with a simple example.

Example 6. Let π_7 be the following execution:

$$\pi_7 \overset{\Delta}{=} \mathtt{u := f(w)} \cdot \underbrace{\mathtt{u := f(u)} \cdots \mathtt{u := f(u)}}_{n} \cdot \mathtt{v := f(w)} \cdot \underbrace{\mathtt{v := f(v)} \cdots \mathtt{v := f(v)}}_{n} \cdot \mathbf{assume}(\mathtt{u} \neq \mathtt{v})$$

It is easy to see that in the above execution π_7, the values of the variables $\mathtt{u}$ and $\mathtt{v}$ will be equal after executing all the $2n + 2$ assignments, in any heap structure. As a result of the last assumption "**assume**($\mathtt{u} \neq \mathtt{v}$)", the execution π_7 will thus be infeasible in all heap structures.

However, in order to accurately determine the relationship "$\mathtt{u} = \mathtt{v}$" at the end of the execution, one needs to keep track of an unbounded amount of information (as n can be increased unboundedly). This is the crucial insight in the first condition (*memoizing*), which states that when a term has been computed and *dropped* (i.e., no variable points to the term), then the execution should not recompute this term. Indeed, the above execution π_7 does not meet this requirement — the execution computed the term $t = \mathtt{f}(\widehat{\mathtt{w}})$ in its first half and stored it in $\mathtt{u}$, re-assigned this variable $\mathtt{u}$ immediately after computing t, thereby losing all copies of t, and then later recomputed this term again in its second half. In fact, each of the terms $\mathtt{f}^i(\mathtt{w})$ ($1 \leq i < n$) have been recomputed without retaining their original copies. Clearly, π_7 is not a coherent execution.

A similar example highlights the importance of the second coherence restriction (*early* assumes).

Example 7. Let π_8 be the following execution:

$$\pi_8 \overset{\Delta}{=} \mathtt{u := u_0} \cdot \overbrace{\mathtt{u := f(u)} \cdots \mathtt{u := f(u)}}^{n} \cdot \mathtt{v := v_0} \cdot \overbrace{\mathtt{v := f(v)} \cdots \mathtt{v := f(v)}}^{n} \cdot \mathbf{assume}(\mathtt{u_0} = \mathtt{v_0}) \\ \cdot \mathbf{assume}(\mathtt{u} \neq \mathtt{v})$$

Observe that this execution is similar to the execution in Example 6. Also observe that, as before, this execution is infeasible in any heap structure. However, any algorithm that would accurately determine this in a *streaming* fashion would require access to unbounded memory.

Definition 4. A program is said to be coherent if all its executions are coherent.

4 ASSERTION CHECKING FOR COHERENT ALIAS-AWARE PROGRAMS

In this section we discuss our first main result — decidability of assertion checking for programs that are both alias-aware as well as coherent.

Let us first define the problem of assertion checking. For this, we augment our programs with a special statement '**assert**(**false**)', and thus our new syntax is given by the following grammar ($\langle stmt \rangle$ is as defined in Section 2.1.1).

$$\langle stmt \rangle_{\text{assert}} ::= \langle stmt \rangle \mid \mathbf{assert}(\mathbf{false}) \mid \langle stmt \rangle_{\text{assert}} \,;\, \langle stmt \rangle_{\text{assert}}$$

Observe that more complex assertions (including boolean combinations of equality/disequality assertions) can be expressed by translating them to conditional statements. For example, the assertion '**assert**($x = y$)' would translate to the program '**if** ($x \neq y$) **then assert**(**false**) **else skip**'.

The set of executions for programs with assertions consists of sequences over the alphabet $\Pi_{\text{assert}} = \Pi \cup \{$"**assert**(**false**)"$\}$ and can be defined as in Section 2.2, with the addition of the following rule: Exec(**assert**(**false**)) = **assert**(**false**)

The functions $\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}, \mathsf{Val}^{\mathcal{M}}_{\mathsf{Data}}, \mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{Loc}}$ and $\mathsf{FldsVal}^{\mathcal{M}}_{\mathsf{Data}}$ in the presence of "**assert**(**false**)" are defined as before for execution prefixes without "**assert**(**false**)", and can be assumed to map all elements in their domain to a special symbol $\perp$ for all executions containing "**assert**(**false**)". The feasibility of a partial execution on the alphabet $\Pi_{\text{assert}} \setminus \{$"**assert**(**false**)"$\}$ on a heap structure $\mathcal{M}$ is defined as before (Section 2.3) and is undefined otherwise.

Definition 5 (Assertion Checking for Heap-Manipulating Programs)**.** The problem of assertion checking for a program $s \in \langle stmt \rangle_{\text{assert}}$ is to check whether for every heap structure $\mathcal{M}$ and for every partial execution of s of the form $\sigma \cdot$ "**assert**(**false**)", we have that σ is infeasible on $\mathcal{M}$.

We first note that the above problem is undecidable in general, a direct consequence of [Mathur et al. 2019a]. Indeed, uninterpreted programs are heap-manipulating programs that do not mention any heap variables.

Theorem 2. *Assertion checking for heap-manipulating programs is undecidable.*

Finally, we state our first decidability result. In the following, an alias-aware coherent program is a program that is both coherent and alias-aware.

Theorem 3 (Decidability of Uninterpreted Assertion Checking)**.** *Assertion checking is decidable for the class of alias-aware coherent programs and is* PSPACE*-complete.*

The proof of the above result relies on the following observations. First, for alias-aware executions, the terms associated with variables reflect their relationships on all heap structures (Theorem 1). Second, in this case, the streaming congruence closure algorithm for checking feasibility of coherent executions introduced in [Mathur et al. 2019a] can be extended directly to the case of coherent executions of heap-manipulating programs. The PSPACE-hardness follows from the PSPACE-hardness result for uninterpreted programs without updates ([Mathur et al. 2019a]). As a consequence of these observations, we also obtain the following result:

Theorem 4. *The problem of checking membership in the class of coherent and alias-aware programs is decidable and is in* PSPACE.

5 MEMORY SAFETY, FOREST DATA-STRUCTURES, AND STREAMING-COHERENCE

We now tackle the problem of memory safety verification for heap-manipulating programs. The goal here is, given a program and an allocated set of locations (defined using a reachability specification), to check whether the program only dereferences (using pointer fields) locations that are in the (dynamically changing) set of allocated locations. We develop a technique for when the initial heap holds a *forest data-structure* (disjoint lists and tree-like structures). We define a subclass of programs, called streaming-coherent, for which memory safety is decidable. The key idea is to utilize the fact that the initial heap is a forest data-structure in order to make programs alias aware.

For an execution that works over a forest data-structure, one can accurately infer the aliasing relationship between two program variables by tracking if they point to locations on the heap obtained by traversing the same path (i.e., starting from the same initial location and taking the same pointers at each step). Intuitively, in a forest data-structure, two distinct traversals always lead to *different* locations. In addition, when the program execution does a single pass over the data-structure, one can keep track of the aliasing relationship between variables (or equivalently, whether the locations pointed to by two variables are same or not) using a *streaming* algorithm. The notion of streaming-coherence essentially ensures such a single pass. Finally, updatable maps can be used to keep track of the initial allocated set and the allocations/frees performed by the program during its execution. Consequently, memory safety can be reduced to assertion checking.

This section is organized as follows. Section 5.1 introduces the reachability specifications that specify the initial set of allocated locations, and defines the memory safety problem formally. We also show here that memory safety is undecidable in general, and undecidable even for coherent programs. Section 5.2 defines the class of forest data-structures, and shows that memory safety remains undecidable for programs that start with a heap that holds a forest data-structure. Section 5.3 defines the notion of streaming-coherent programs.

5.1 Reachability Specification and Memory Safety

Reachability Specification. Heap-manipulating programs can be annotated by a *reachability specification* that restricts the allowable nodes that can be accessed by a program. A reachability specification is an indexed set of triples $\varphi = \{\varphi_k\}_{k=1}^n$ where $\varphi_i = (\mathsf{Start}_i, \mathsf{Pointers}_i, \mathsf{Stop}_i)$ is such that $\mathsf{Start}_k \subseteq C_{\mathsf{Loc}}$, $\mathsf{Stop}_k \subseteq C_{\mathsf{Loc}}$ and $\mathsf{Pointers}_k \subseteq \mathcal{F}_{\mathsf{Loc}}$. Each triple φ_i denotes a set of locations Reach_i, which is the least set that contains Start_i, does not include Stop_i and is closed under repeated applications of pointer fields $\mathsf{Pointers}_i$. More formally, given a heap structure $\mathcal{M}$ with interpretation $\mathcal{I}$, Reach_i gives a set of locations, which is the smallest set such that (a) $\{\mathcal{I}(c) \mid c \in \mathsf{Start}_i\} \setminus \{\mathcal{I}(c) \mid c \in \mathsf{Stop}_i\} \subseteq \mathsf{Reach}_i$, and (b) for every $e \in \mathsf{Reach}_i$ and for every $p \in \mathsf{Pointers}_i$, if $\mathcal{I}(p)(e) \notin \{\mathcal{I}(c) \mid c \in \mathsf{Stop}_i\}$, then $\mathcal{I}(p)(e) \in \mathsf{Reach}_i$. We let $\mathsf{Reach}_\varphi = \bigcup\limits_{i=1}^{n} \mathsf{Reach}_i$. Often the heap structure $\mathcal{M}$ will be implicit and we will omit mentioning it.

Allocated Nodes. Starting with a reachability specification φ on a given heap structure $\mathcal{M}$, an execution σ defines a set of *allocated nodes*, which we denote as $\mathsf{Alloc}(\sigma)$ and define as follows.

$$\begin{array}{rcl} \mathsf{Alloc}(\varepsilon) & = & \mathsf{Reach}_\varphi \\ \mathsf{Alloc}(\sigma \cdot s) & = & \begin{cases} \mathsf{Alloc}(\sigma) \cup \{\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma \cdot s, x)\} & \text{if } s = \text{“}\mathbf{alloc}(x)\text{”} \\ \mathsf{Alloc}(\sigma) \setminus \{\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\sigma, x)\} & \text{if } s = \text{“}\mathbf{free}(x)\text{”} \\ \mathsf{Alloc}(\sigma) & \text{otherwise} \end{cases} \end{array}$$

Memory Safety. An execution σ is said to *violate memory safety* over a heap structure $\mathcal{M}$ with respect to a reachability specification φ if there is a prefix $\rho' = \rho \cdot s$ of σ such that ρ is feasible over $\mathcal{M}$ and one of the following holds.

(1) s dereferences a location that was not allocated. That is, s is of the form “$w := y{\cdot}h$” or “$y{\cdot}h := w$”, $y \in V_{\mathsf{Loc}}$ and w and h are variables and pointer fields of appropriate sorts, such that $\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\rho, y) \notin \mathsf{Alloc}(\rho)$.

(2) s frees an unallocated location. That is, s is of the form $\mathbf{free}(x)$ and $\mathsf{Val}^{\mathcal{M}}_{\mathsf{Loc}}(\rho, x) \notin \mathsf{Alloc}(\rho)$.

An execution σ is *memory safe* over $\mathcal{M}$ with respect to φ if it does not violate memory safety over $\mathcal{M}$ with respect to φ. With this, we can now define the memory safety verification problem. In the following, we fix our signature Σ.

Definition 6 (Memory Safety Verification)**.** The memory safety verification problem asks, given a program $P \in \langle stmt \rangle$ and a reachability specification φ, whether for all heap structures $\mathcal{M}$, each execution $\sigma \in \mathsf{Exec}(P)$ is memory safe over $\mathcal{M}$ with respect to φ.

We show, unsurprisingly, that checking memory safety is undecidable in general.

Theorem 5 (Undecidability of Memory Safety)**.** *Memory safety verification is undecidable.*

Proof. In [Mathur et al. 2019a], the authors consider uninterpreted programs, which are programs that have variables taking values in a data domain that is uninterpreted; programs in [Mathur et al. 2019a] don't have heap variables, and do not modify heaps. It was shown (Theorem 11 in [Mathur et al. 2019a]) that given an uninterpreted program P, the problem of determining if there is a data domain $\mathcal{M}$ and an execution ρ of P, such that ρ is feasible in $\mathcal{M}$, is undecidable. Our result here can be proved by a simple reduction from that problem. Let P be an uninterpreted program. Consider the reachability specification $\varphi = (\{\widehat{x}\}, \{p\}, \{\widehat{x}\})$ such that $x \in V_{\mathsf{Loc}}$ is a new variable. Consider program $P' = P; y = x{\cdot}p$. Observe that P' is memory safe with respect to φ if and only if P does not have a feasible execution with respect to some data model. □

Given the undecidability result in Theorem 5, we need to identify a restricted subclass of programs and initial heap structures for which the problem of verifying memory safety is decidable. This leads us to the notions of forest data-structures and streaming-coherent programs. In some sense, forest data-structures are natural classes of heap structures where the alias-aware restrictions become very minimal. We combine these remaining restrictions with our usual notion of coherence to introduce a new notion of streaming-coherent programs. We define these below.

5.2 Forest Data-Structures

Let us now define our characterization of heap structures that will be amenable to our decidability result. The main restriction is stated in bullet (3) below and intuitively disallows two different paths to the same location on the heap.

Definition 7 (Forest Data-structures)**.** A heap structure $\mathcal{M} = (U_{\mathrm{Loc}}, U_{\mathrm{Data}}, \mathcal{I})$ over a signature Σ is said to be a forest data-structure with respect to a reachability specification $\varphi = \{\varphi_k\}_{k=1}^{n}$ if

(1) for every $1 \leq i \leq n$, each set of stopping locations is a singleton set of the form $\mathsf{Stop}_i = \{\mathsf{stop}_i\}$,

(2) for every $c \in \bigcup_{k=1}^{n} \mathsf{Stop}_k$, and for every $f \in \mathcal{F}_{\mathrm{Loc}}$, we have that $\mathcal{I}(f(c)) = \mathcal{I}(c)$, and

(3) for every $t_i \in \mathsf{Terms}(\mathsf{Start}_i, \mathsf{Pointers}_i) \cup \mathsf{Stop}_i$ and $t_j \in \mathsf{Terms}(\mathsf{Start}_j, \mathsf{Pointers}_j) \cup \mathsf{Stop}_j$ we have, if $\mathcal{I}(t_i) = \mathcal{I}(t_j)$, then either $t_i = t_j \in \mathsf{Start}_i \cap \mathsf{Start}_j$, or $\mathcal{I}(t_i) = \mathcal{I}(t_j) = \mathcal{I}(\mathsf{stop}_i) = \mathcal{I}(\mathsf{stop}_j)$.

Intuitively, a heap structure is a forest data-structure with respect to φ, if the subgraph G_i induced by the set of nodes (excluding Stop_i) reachable from Start_i, using any number of pointers from $\mathsf{Pointers}_i$ forms a tree, and further, any two subgraphs G_i and G_j do not have a node in common (except possibly for the starting locations). Notice that Definition 7 do not impose any restrictions on the elements of the data sort of a heap structure.

Our notion of forest data-structures handles the aliasing problem while still being able to express many practical reachability specifications. Consider the following example similar to our pathological example from Section 3.1.

Example 8. $\pi_2 = \pi_2' \cdot \textbf{assume}(\mathtt{z}_2 = \mathtt{z}_3)$ where

$$\pi_2' \stackrel{\Delta}{=} \textbf{assume}(\mathtt{x} \neq \mathtt{NIL}) \cdot \textbf{assume}(\mathtt{y} \neq \mathtt{NIL}) \cdot \mathtt{z}_1 := \mathtt{x{\cdot}next} \cdot \textbf{assume}(\mathtt{z}_1 \neq \mathtt{z}_2) \cdot \mathtt{y{\cdot}next} := \mathtt{z}_2 \cdot \mathtt{z}_3 := \mathtt{x{\cdot}next}$$

With respect to forest data-structures the above pathological execution π_2 is, in fact, infeasible. To understand this, observe that for a forest data-structure, if two different locations ($\mathtt{x}$ and $\mathtt{y}$ in the execution π_2 for instance) are not equal to the stopping location ($\mathtt{x} \neq \mathtt{NIL}$ and $\mathtt{y} \neq \mathtt{NIL}$), then these locations are also different from each other (and thus $\mathtt{x} \neq \mathtt{y}$). This means that in the execution π_2, the update "$\mathtt{y{\cdot}next} := \mathtt{z}_2$" will not affect the value of "$\mathtt{x{\cdot}next}$". Now, when the execution reads the value of "$\mathtt{x{\cdot}next}$" in $\mathtt{z}_3$, it is expected to be the same as earlier (i.e., some location as pointed to by $\mathtt{z}_1$), but since $\mathtt{z}_1 \neq \mathtt{z}_2$, we must have $\mathtt{z}_3 \neq \mathtt{z}_2$. This means that the last assume $\textbf{assume}(\mathtt{z}_2 = \mathtt{z}_3)$ makes the execution infeasible.

However, the notion of forest data-structures, by itself, is not enough to ensure decidability, as is shown by the following result.

Theorem 6. *The memory safety verification problem for forest data-structures is undecidable.*

Proof. Follows trivially from Theorem 5 because the reachability specification φ used in the proof of Theorem 5 is such that Reach_φ is the empty set. □

Next, we introduce a class of programs called streaming-coherent programs that, in conjunction with forest data-structures results in a PSPACE decision procedure. We will discuss the decision procedure in Section 6.

5.3 Streaming-Coherent Executions and Programs

In this section, we identify the class of streaming-coherent programs and executions, for which we show decidability. It is analogous to the notion of coherence (Definition 3), except that it uses a different notion of equivalence (for terms) instead of $\equiv_{\alpha(\cdot)}$. We define this new notion of equivalence (*forest equality closure*) below. Intuitively, this notion characterizes the behavior of executions on the class of forest data-structures and allows us to adapt the algorithm used for assertion checking of alias-aware coherent programs (Theorem 3).

Definition 8 (Forest Equality Closure). Let $\varphi = \{\varphi_i\}_{i=1}^{n}$ be a reachability specification, with $\varphi_i = (\mathsf{Start}_i, \mathsf{Pointers}_i, \mathsf{Stop}_i)$. Let $\mathsf{Terms}_i = \mathsf{Terms}(\mathsf{Start}_i, \mathsf{Pointers}_i)$ $(1 \leq i \leq n)$. Let $E \subseteq \mathsf{Terms} \times \mathsf{Terms}$ be an equality relation on terms. The *forest equality closure* of E with respect to φ, denoted $\mathsf{Closure}^{=}(\varphi, E) \subseteq \mathsf{Terms} \times \mathsf{Terms}$ is the smallest congruence relation that satisfies the following.

- $E \subseteq \mathsf{Closure}^{=}(\varphi, E)$.
- For every $t_i \in \mathsf{Terms}_i$ and $t_j \in \mathsf{Terms}_j$ such that $t_i \neq t_j$, we have

$$(t_i, t_j) \in \mathsf{Closure}^{=}(\varphi, E) \implies \{(t_i, \mathsf{stop}_i), (t_j, \mathsf{stop}_j)\} \subseteq \mathsf{Closure}^{=}(\varphi, E)$$

We will now define the notion of *streaming-coherent programs* with respect to a given reach specification φ. To do this, we first need the notion of streaming-coherent executions. This notion is very similar to the notion of coherence (Definition 3) and requires that terms are not recomputed and that assumes over the data sort are early.

Definition 9 (Streaming-Coherence). Let σ be a complete or partial execution. σ is said to be streaming-coherent if it satisfies the following two conditions.

Memoizing. Let ρ be a prefix of σ of the form $\rho' \cdot$"$x := y \cdot h$" or $\rho' \cdot$"$x := f(\mathbf{y})$" and let $t = \mathsf{Comp}(\rho, x)$. If there is a term $t' \in \mathsf{Terms}(\rho')$ such that $t \cong_{\mathsf{Closure}^{=}(\varphi, \alpha(\rho'))} t'$, then it must be the case that there is some variable $z \in V$ such that $\mathsf{Comp}(\rho', z) \cong_{\mathsf{Closure}^{=}(\varphi, \alpha(\rho'))} t$. This condition is applicable to every sort-sensible combination of symbols.

Early Assumes. Let ρ be a prefix of the form $\rho' \cdot$ "**assume**$(u = v)$" where $u, v \in V_{\mathsf{Data}}$ and $t_u = \mathsf{Comp}(\rho', u), t_v = \mathsf{Comp}(\rho', v)$. If there is a term $t \in \mathsf{Terms}(\rho')$ such that t is a superterm of t_u or t_v modulo $\cong_{\mathsf{Closure}^{=}(\varphi, \alpha(\rho'))}$, then there must be some variable $w \in V_{\mathsf{Data}}$ such that $\mathsf{Comp}(\rho', w) \cong_{\mathsf{Closure}^{=}(\varphi, \alpha(\rho'))} t$.

Observe that the above definition is close the definition of coherence (Definition 3), but nevertheless there are important differences. First, the early assumes requirement does not apply to variables of the Loc sort because forest data-structures simplify that requirement. Second, the congruence with respect to which equalities are demanded in the above definition is not one of the congruence closure defined by the equalities accumulated by the execution (which is the congruence used in Definition 3), but rather the congruence of forest equality closure.

Definition 10. A program is said to be streaming-coherent with respect to φ if all its executions are streaming-coherent with respect to φ.

We note here that streaming-coherent programs on forest data-structures are essentially *one-pass* algorithms. Intuitively, forest data-structures enforce the alias-aware restriction by mandating that two locations obtained by two different traversals from the set of initial locations (therefore being represented by two different *terms*) are different (with minor exceptions). Therefore if a streaming-coherent execution computes a term twice, i.e., visits a location twice, by definition it has to store a pointer to that location in some variable. Since the number of variables is fixed *a priori*, it is simple to see that a nontrivial, multi-pass algorithm such as linked-list sorting would inherently be non-streaming-coherent (since it is meant to work on lists of arbitrary size).

6 STREAMING CONGRUENCE CLOSURE FOR FOREST DATA-STRUCTURES

We will now focus on streaming-coherent programs whose initial heap is a forest data-structure. We now present the second main result of our paper.

Theorem 7 (Decidability of Memory Safety for Streaming-Coherent Programs and Forest Data-Structures)**.** *The memory safety verification problem over forest data-structures for streaming-coherent programs is decidable and is* PSPACE*-complete.*

Given a reachability specification, we present an algorithm for checking memory safety of such programs. This will establish the decidability and the membership of the problem in PSPACE. The hardness follows from the PSPACE-hardness result in [Mathur et al. 2019a].

Our algorithm is automata theoretic— we construct a finite state automaton $\mathcal{A}_{\mathsf{MS}}$ such that its language $L(\mathcal{A}_{\mathsf{MS}})$ includes all streaming-coherent executions that are memory safe and excludes all streaming-coherent executions that are not memory safe. Finally, in order to determine if a given streaming-coherent program P is memory safe, we simply check if $\mathsf{Exec}(P) \subseteq L(\mathcal{A}_{\mathsf{MS}})$; since $\mathsf{Exec}(P)$ is also a regular language, this reduces to checking intersection of regular languages.

Let us first discuss some intuition for our construction. The state of the automaton has several components which can be categorized into two groups: (1) those that track feasibility of executions, and (2) those that are used to check for violations of memory safety. The first category comprises of three components $\equiv, d, P$ which keep track of equalities, disequalities, and functional relationships between variables at a given point in the execution. These components are inspired by the work of [Mathur et al. 2019a] in the context of *coherent uninterpreted programs*, where this information was used to answer questions of assertion checking/feasibility. Our extended and refined notion of streaming-coherent programs also has this property, and these three components serve the same roles in our setting. Similar to the work in [Mathur et al. 2019a], when an execution is streaming-coherent, these three components $\equiv, d, P$ can be accurately maintained at every step in the execution by performing *local congruence closure*.

We now describe the remaining components of the state, namely those that help check for memory safety. We know from our earlier discussion that forest data-structures are implicitly alias aware. This is because such heap structures consist of many disjoint reachability specifications each of which describes a tree. Therefore, nodes obtained by different pointer traversals from the initial locations are different, and each tree in the forest is closed under a given set of pointers. We keep track of locations (for the purpose of memory safety) primarily using three ordered collections which are disjoint: (a) a collection $Y = \{Y_i\}_i$ of 'yes' sets, (b) a collection $M = \{M_i\}_i$ of 'maybe' sets and (c) a 'no' set N. Each Y_i and M_i above correspond to the i^{th} reachability specification φ_i. In the following, we describe each of these.

N is a set of variables that point to locations which, when dereferenced, lead to a memory safety violation. These locations include the stopping locations of any of the reachability specifications, or those locations in the allocated set that have been freed during the execution. Consider the reachability specification $\psi = (\{\mathtt{x}\}, \{\mathtt{next}\}, \{\mathtt{NIL}\})$. At the beginning of any execution, the set N for this specification is $\{\mathtt{NIL}\}$. Further, when the execution encounters the statement **free**(y), we update the N component of the state by adding y.

The sets Y_i hold a subset of program variables that point to locations obtained by traversing the i^{th} reachability specification φ_i. In particular, a variable z belongs to Y_i when we know that the corresponding location can be dereferenced without causing a memory safety violation in any heap structure. This can happen when the execution establishes that the location pointed to by z is not the stopping location of φ_i (i.e., stop_i). Consider ψ and the single step execution $\sigma_1 = \textbf{assume}(\mathtt{x} \neq \mathtt{NIL})$. At the end of σ_1 we know that in all heap structures in which σ_1 is feasible, it must be that the location pointed to by x can be dereferenced, and therefore we would add x to the corresponding 'yes' set.

Lastly, the 'maybe' sets M_i hold variables that can be obtained by traversing the tree of φ_i, but are neither known to be stopping locations/freed locations, nor known to be in the allocated set in all feasible heap structures. Consider again ψ and the execution $\sigma_2 = \textbf{assume}(\mathtt{x} \neq \mathtt{NIL}) \cdot \mathtt{y} := \mathtt{x}{\cdot}\mathtt{next}$. Recall that x belongs to the 'yes' set after the first statement (i.e., the execution σ_1) and we can therefore dereference it. However, at the end of the execution we would include y in the 'maybe' set because the location x·next is not in the allocated set in all feasible heap structures. In particular, a heap structure defining a list of length one is a feasible structure for this execution but x·next would be NIL, which is not in the allocated set. Now consider the execution $\sigma_3 = \sigma_2 \cdot \textbf{assume}(\mathtt{y} \neq \mathtt{NIL})$. After the last statement in σ_3 we would now shift y from the 'maybe' set to the 'yes' set.

This is precisely what our automaton does, shifting variables between components and flagging memory safety violations when the execution dereferences (the location pointed to by) a variable that is not in any Y_i. However, we do this modulo equalities between variables and these sets described above are in fact sets of equivalence classes. Lastly, we also keep track of variables that point to locations allocated by "$\textbf{alloc}(\cdot)$" using a set A of equivalence classes of variables. Of course, we would transfer an equivalence class from A to N if the execution frees this memory, similar to what happens to members of the 'yes' sets.

6.1 Formal Description of Construction

We shall now proceed with the formal construction of the automaton $\mathcal{A}_{\mathsf{MS}}$, which accepts a streaming-coherent execution iff it is memory safe. Recall that our reachability specification is an indexed set of tuples $\varphi = \{\varphi_k\}_{k=1}^n$, where $\varphi_k = (\mathsf{Start}_k, \mathsf{Pointers}_k, \mathsf{Stop}_k)$. To simplify presentation, we assume that the set of variables V in our programs is such that for every constant c appearing in the reachability specification φ, there is a variable v_c corresponding to c. Further, we assume that these variables are never overwritten. We will, therefore, often interchangeably refer to these constants in φ by their corresponding variables, and vice versa. These assumptions can be relaxed with a more involved construction.

The automaton is a tuple $\mathcal{A}_{\mathsf{MS}} = (Q, q_0, \delta)$, where Q is the set of states, q_0 is the initial state and $\delta : Q \times \Pi \to Q$ is the transition function. Recall that executions are strings over Π, which is also the alphabet of the automaton $\mathcal{A}_{\mathsf{MS}}$. We describe each of these components below.

States. The automaton has two distinguished states $q_{\mathsf{infeasible}}$ and q_{unsafe}. All other states are tuples of the form $(\equiv, d, P, Y, M, N, A, X)$, where each component is as follows.

- $\equiv$ is an equivalence relation over V that respects sorts. We will use $[x]_\equiv$ to denote the set $\{y \mid (x, y) \in \equiv\}$
- d is a symmetric set of pairs of the form (c_1, c_2), where $c_1, c_2 \in V/\equiv$ are equivalence classes.
- P associates partial mappings to function symbols in $\mathcal{F}_{\mathsf{Data}}$ and pointer field in $\mathsf{Flds} = \mathsf{Flds}_{\mathsf{Loc}} \cup \mathsf{Flds}_{\mathsf{Data}}$. More formally, for every $f \in \mathcal{F}_{\mathsf{Data}}$ of arity r, and classes $c_1, c_2, \ldots, c_r \in V_{\mathsf{Data}}/\equiv$, we have $P(f)(c_1, \ldots, c_r) \in V_{\mathsf{Data}}/\equiv$ (if defined). Similarly, for every $p \in \mathsf{Flds}_{\mathsf{Loc}}$, and every $c \in V_{\mathsf{Loc}}/\equiv$, $P(p)(c) \in V_{\mathsf{Loc}}/\equiv$, and for every $d \in \mathsf{Flds}_{\mathsf{Data}}$, and every $c \in V_{\mathsf{Loc}}/\equiv$, $P(d)(c) \in V_{\mathsf{Data}}/\equiv$. We will say $P(h)(c_1, \ldots, c_k) = \mathsf{undef}$ when the function/pointer symbol h is not defined on the arguments $c_1, \ldots, c_k$.
- $Y = \{Y_k\}_{k=1}^n$ and $M = \{M_k\}_{k=1}^n$ are indexed collections of sets such that for every $1 \leq i \leq n$, we have $Y_i, M_i \subseteq V_{\mathsf{Loc}}/\equiv$.

- The sets N, A and X are sets of equivalence classes of location variables, i.e., $N, A, X \subseteq V_{\mathsf{Loc}}/\equiv$.

Initial State. The initial state q_0 is the tuple $(\equiv_0, d_0, P_0, Y_0, M_0, N_0, A_0, X_0)$ such that

- $\equiv_0$ is the identity relation on the set V of variables,
- P_0 is such that for all functions and pointer fields f, the range of $P(f)$ is empty,
- each of d_0, A_0 and $(Y_0)_1, \ldots, (Y_0)_n$ are $\varnothing$ (empty set),
- for each $1 \leq i \leq n$, $(M_0)_i = \{[v]_{\equiv_0} \mid v \in \bigcup_{i=1}^{n} \mathsf{Start}_i\}$,
- $N_0 = \{[v]_{\equiv_0} \mid v \in \bigcup_{i=1}^{n} \mathsf{Stop}_i\}$.
- $X_0 = \{[v]_{\equiv_0} \mid v \in V_{\mathsf{Loc}}\} \setminus \big(N_0 \cup \bigcup_{i=1}^{n}(M_0)_i\big)$.

Transitions. The states $q_{\mathsf{infeasible}}$ and q_{unsafe} are absorbing states. That is, for every $s \in \Pi$, $\delta(q_{\mathsf{infeasible}}, s) = q_{\mathsf{infeasible}}$ and $\delta(q_{\mathsf{unsafe}}, s) = q_{\mathsf{unsafe}}$. In the following, we describe the transition function for every other state in Q. Let $q \in Q \setminus \{q_{\mathsf{infeasible}}, q_{\mathsf{unsafe}}\}$, $s \in \Pi$ and let $q' = \delta(q, s)$. Then, q' is $q_{\mathsf{infeasible}}$ or q_{unsafe}, or is of the form $q' = (\equiv', d', P', Y', M', N', A', X')$.

(1) **Case** $s =$ **"**$u := v$**"**, $u, v \in V$.
In this case, we add the variable u into the class of v and appropriately update each of the components. That is, $\equiv' = \big(\equiv \setminus \{(u, u'), (u', u) \mid u \neq u', u' \in [u]_\equiv\}\big) \cup \{(u, v'), (v', u) \mid v' \in [v]_\equiv\}$. The other components of the state are the same as in q.

(2) **Case** $s =$ **"**$x := y{\cdot}p$**"**, $x, y \in V_{\mathsf{Loc}}$ **and** $p \in \mathsf{Flds}_{\mathsf{Loc}}$.
In this case, we need to check if the variable y corresponds to a location that can be dereferenced. If not, we have a memory safety violation; otherwise, we establish the relationship $x = p(y)$ in the next state. Formally, if there is no i such that $[y]_\equiv \in Y_i$ and if $[y]_\equiv \notin A$, then $q' = q_{\mathsf{unsafe}}$. Otherwise we have that $[y]_\equiv \in A$ or there is a k such that $[y]_\equiv \in Y_k$. In this case we define the tuple $q' = (\equiv', d', P', Y', M', N', A', X')$ below. Here, we need to consider the following cases.
   - Case $P(p)([y]_\equiv)$ is defined and equals $[z]_\equiv$. In this case, q' is defined in the same manner as though $s =$ "$x := z$".
   - Case $P(p)([y]_\equiv) = \mathsf{undef}$. Here, for streaming-coherent programs it must be the case that $[y]_\equiv \in Y_k$ for some k. Here, we create a new singleton equivalence class containing x and set the value of the p map, on y to be this new class. We also assert that $[x]_{\equiv'}$ is not equal to any class in any of the Y_is or in A. That is,
     – $\equiv' = \equiv \setminus \{(x, x') \mid x' \neq x\} \cup \{(x, x)\}$.
     – $d' {=} \{([u]_{\equiv'}, [v]_{\equiv'}) \mid u \neq x, v \neq x, ([u]_\equiv, [v]_\equiv) \in d\} \cup \{([x]_{\equiv'}, c), (c, [x]_{\equiv'}) \mid c \in A \cup \bigcup\limits_{i=1}^{n} Y_i\}$
     – $P'(p)([y]_{\equiv'}) = [x]_{\equiv'}$. For all other combinations of functions/pointers and arguments, P' behaves same as P.
     – One of the sets M_k and X are updated depending upon the pointer p. If $p \in \mathsf{Pointers}_k$, then $M'_k = \{[z]_{\equiv'} \mid [z]_\equiv \in M_k\} \cup \{[x]_{\equiv'}\}$. Otherwise, $X = \{[z]_{\equiv'} \mid [z]_\equiv \in X\} \cup \{[x]_{\equiv'}\}$.
     – All other components are the same as in q.

(3) **Case** $s =$ **"**$a := y{\cdot}d$**"**, $a \in V_{\mathsf{Data}}, y \in V_{\mathsf{Loc}}$ **and** $d \in \mathsf{Flds}_{\mathsf{Data}}$.
As in the previous case, $q' = q_{\mathsf{unsafe}}$ if $[y]_\equiv \notin \cup_{i=1}^{n} Y_i \cup A$. Otherwise, similar to the previous case, we have two cases to consider. As before, if there is a variable $b \in V_{\mathsf{Data}}$ such that $P(d)([y]_\equiv) = [b]_\equiv$, then we treat this case as that of "$a := b$". Otherwise, the new equivalence relation $\equiv'$ is such that $\equiv' = \equiv \setminus \{(a, u) \mid u \neq a\} \cup \{(a, a)\}$, while the other components are the same as in q.

(4) **Case** $s =$ **"**$y{\cdot}h := u$**"**, $y \in V_{\mathsf{Loc}}$.
We uniformly handle the case of h being either a pointer field ($\mathsf{Flds}_{\mathsf{Loc}}$) or a data field ($\mathsf{Flds}_{\mathsf{Data}}$). Here we have $q' = q_{\mathsf{unsafe}}$ if $[y]_\equiv \notin \cup_{i=1}^{n} Y_i \cup A$. Otherwise, we simply change P as follows (while keeping other components the same as in q):

- $P'(f) = P(f)$ for $f \neq h$
- $P'(h)([y]_{\equiv'}) = [u]_{\equiv'}$
- For $z \in V_{\mathsf{Loc}}$ such that $z \notin [y]_{\equiv}$, $P'(h)([z]_{\equiv'}) = [w]_{\equiv'}$ if $P(h)([z]_{\equiv}) = [w]_{\equiv}$ for some variable w, and $P'(h)([z]_{\equiv'}) = \mathsf{undef}$ if no such variable w exists.

(5) **Case** $s =$ **"**$a := f(c_1, \ldots, c_r)$**"**, $a \in V_{\mathsf{Data}}$.
Here, we have two cases to consider again. If there is a variable $b \in V_{\mathsf{Data}}$ such that $P(f)([c_1]_{\equiv}, \ldots, [c_r]_{\equiv}) = [b]_{\equiv}$, then we treat this case as that of "$a := b$". Otherwise, we add a singleton equivalence class containing a and update $P(f)$, while keeping all other components the same (modulo the new equivalence relation). Formally,

- $\equiv' = \equiv \setminus \{(a,u),(u,a) \mid u \in V_{\mathsf{Data}}\} \cup \{(a,a)\}$.
- $P'(h)$ is same as in q if $h \neq f$. The evaluation of P' on f is described as follows.

$$P'(f)([u_1]_{\equiv'}, \ldots, [u_r]_{\equiv'}) = \begin{cases} [a]_{\equiv'} & \text{if for every } 1 \leq i \leq r, [u_i]_{\equiv} = [c_i]_{\equiv} \text{ and } a \neq u_i \\ [u]_{\equiv'} & \begin{array}{l}\text{otherwise if } a \notin \{u, u_1, \ldots, u_r\} \\ \text{and } [u]_{\equiv} = P(f)([u_1]_{\equiv}, \ldots, [u_r]_{\equiv})\end{array} \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

- All other components are the same as in q.

(6) **Case** $s =$ **"alloc**(x)**"**.
In this case, we create a new singleton class containing x, and add this class to the component A. We also assert that this new class is not equal to any other class. Formally,

- $\equiv' = \equiv \setminus \{(x,u) \mid u \in V_{\mathsf{Loc}}\} \cup \{(x,x)\}$.
- $d' = \{([u_1]_{\equiv'}, [u_2]_{\equiv'}) \mid u_1 \neq u_2, ([u_1]_{\equiv}, [u_2]_{\equiv}) \in d \text{ or } u_1 = x \vee u_2 = x\}$
- $A' = \{[u]_{\equiv'} \mid [u]_{\equiv} \in A\} \cup \{[x]_{\equiv'}\}$.
- The component P is updated as follows. For every function symbol $f \in \mathcal{F}_{\mathsf{Data}}$, $P'(f)$ is the same as $P(f)$. Further, for every pointer field $h \in \mathsf{Flds}_{\mathsf{Data}} \cup \mathsf{Flds}_{\mathsf{Loc}}$ and for every variable $y \in V_{\mathsf{Loc}} \setminus \{x\}$, we have $P'(h)([y]_{\equiv'}) = P(h)([y]_{\equiv})$. Finally, the mapping on x is defined as $P(p)([x]_{\equiv'}) = [x]_{\equiv'}$ for every location pointer $p \in \mathsf{Flds}_{\mathsf{Loc}}$ and $P(d)([x]_{\equiv'}) = \mathsf{undef}$ for every data pointer $d \in \mathsf{Flds}_{\mathsf{Data}}$.
- All other components are updated as usual.

(7) **Case** $s =$ **"free**(x)**"**.
In this case, if $[x]_{\equiv} \notin A \cup \bigcup_{i=1}^{n} Y_i$, then $q' = q_{\mathsf{unsafe}}$. Otherwise, we remove the class $[x]_{\equiv}$ from the sets A or $Y_1, \ldots, Y_n$ and add it to the set N. That is,

- $\equiv' = \equiv$
- $N' = \{[z]_{\equiv'} \mid [z]_{\equiv} \in N\} \cup \{[x]_{\equiv'}\}$
- for every $1 \leq i \leq n$, $Y_i' = \{[z]_{\equiv'} \mid [z]_{\equiv} \neq [x]_{\equiv}, [z]_{\equiv} \in Y_i\}$
- $A = \{[z]_{\equiv'} \mid [z]_{\equiv} \neq [x]_{\equiv}, [z]_{\equiv} \in A\}$
- Other components remain the same.

(8) **Case** $s =$ **"assume**$(x = y)$**"**, $x, y \in V_{\mathsf{Loc}}$. In this case, if $[x]_{\equiv} = [y]_{\equiv}$ then $q' = q$. Otherwise we have several cases to consider.
In each of these cases, we construct a new tuple $q'' = (\equiv'', d'', P'', Y'', N'', M'', A'', X'')$. Finally, we set $q' = q''$ if $d'' \cap \equiv'' = \varnothing$; otherwise we have $q' = q_{\mathsf{infeasible}}$.

- The first case to consider is when $\{[x]_{\equiv}, [y]_{\equiv}\} \cap \bigcup_{i=1}^{n} M_i = \varnothing$. In this case, we merge $[x]_{\equiv}$ and $[y]_{\equiv}$. More formally, $\equiv''$ is the smallest equivalence relation such that $\big(\equiv \cup \{(x,y)\}\big) \subseteq \equiv''$. Further, for every component $C \in \{X, A, N\} \cup \bigcup_{i=1}^{n} \{Y_i, M_i\}$ with the corresponding component in q'' being C'', and for every $z \in V_{\mathsf{Loc}}$ such that $z \notin [x]_{\equiv} \cup [y]_{\equiv}$, we have $[z]_{\equiv''} \in$

  C'' iff $[z]_\equiv \in C$. For a variable $z \in [x]_\equiv \cup [y]_\equiv$, we have $[z]_{\equiv''} \in C''$ iff $\{[x]_\equiv, [y]_\equiv\} \cap C \neq \emptyset$. The other components of q'' are the same as in q (modulo the new equivalence classes).
  - Otherwise, consider the case when $[x]_\equiv \in M_i$ for some i. In this case, in addition to adding $\{(x, y)\}$ we also add the pair $\{(x, \mathrm{stop}_i)\}$. Similarly if $[y]_\equiv \in M_j$ for some j we add $\{(y, \mathrm{stop}_j)\}$. Construct the state q'' with $\equiv''$ being the smallest equivalence relation including these new pairs (and other components remaining the same).

(9) **Case** $s =$ **"assume**$(x \neq y)$**"**, $x, y \in V_{\mathrm{Loc}}$. Similarly as above, in this case when $([x]_\equiv, [y]_\equiv) \in d$ we have $q' = q$. If $[x]_\equiv = [y]_\equiv$ then $q' = q_{\mathrm{infeasible}}$. Otherwise, we have the following cases:
  - $[x]_\equiv = [\mathrm{stop}_i]_\equiv$ and $[y]_\equiv \in M_i$ for some i (or vice versa). In this case, we simply put the equivalence class of y into Y_i and assert that $[y]_\equiv$ is unequal to all other classes. More formally, $\equiv' = \equiv$, $Y_i' = Y_i \cup \{[y]_\equiv\}$ $d' = d \cup \{([y]_\equiv, [z]_\equiv), ([z]_\equiv, [y]_\equiv) \mid z \notin [y]_\equiv\}$. The other components remain the same.
  - Otherwise, we simply update $d' = d \cup \{([x]_\equiv, [y]_\equiv)\}$; all other components are the same.

(10) **Case** $s =$ **"assume**$(a = b)$**"**, $a, b \in V_{\mathrm{Data}}$. In this case, we merge equivalence classes repeatedly and perform a 'local congruence closure'. We construct a state q'' to determine if the transition must be to $q_{\mathrm{infeasible}}$. More formally, we define the state q'' with the $\equiv''$ component as the smallest equivalence relation such that: (a) $\equiv \cup (a, b) \subseteq \equiv''$ (b) If $(u_i, v_i) \in \equiv$ for $1 \leq i \leq r$ and $[w]_{\equiv''} = f([u_1]_{\equiv''} \ldots, [u_r]_{\equiv''}), [w']_{\equiv''} = f([v_1]_{\equiv''} \ldots, [v_r]_{\equiv''})$ then $(w, w') \in \equiv''$.
The other components remain the same. In particular, it is correct to retain the P component since the above construction is a congruence relation. Finally, if there exist $u, v \in V_{\mathrm{Data}}$ such that $(u, v) \in \equiv''$ and $([u]_{\equiv''}, [v]_{\equiv''}) \in d''$ then $q' = q_{\mathrm{infeasible}}$. Otherwise, $q' = q''$.

(11) **Case** $s =$ **"assume**$(a \neq b)$**"**, $a, b \in V_{\mathrm{Data}}$. Similarly as above, if $[a]_\equiv = [b]_\equiv$ then $q' = q_{\mathrm{infeasible}}$. Otherwise, we update $d' = d \cup \{([a]_\equiv, [b]_\equiv)\}$ and all other components remain the same.

The following theorem states the correctness of the automaton $\mathcal{A}_{\mathrm{MS}}$.

Lemma 8. Let σ be a streaming-coherent execution and let φ be a reachability specification and let q be the state of the automaton $\mathcal{A}_{\mathrm{MS}}$ after reading σ. Then, $q = q_{\mathrm{unsafe}}$ iff there is a forest data-structure $\mathcal{M}$ (with respect to φ) such that σ violates memory safety on $\mathcal{M}$.

The problem of checking if a streaming-coherent program is memory safe against a given specification is decided as follows. Recall that the set of executions of a given program P constitutes a regular language $\mathsf{Exec}(P)$. Let $L(\mathcal{A}_{\mathrm{MS}})$ be the set of executions $\sigma \in \Pi^*$ that do not go to the state q_{unsafe}. Then, the problem of checking if P is memory safe reduces to checking if $\mathsf{Exec}(P) \subseteq L(\mathcal{A}_{\mathrm{MS}})$.

Next, we show that the problem of checking streaming-coherence is also decidable. To address the problem of checking streaming-coherence, we construct an automaton $\mathcal{A}_{\mathrm{checkSC}}$ similar to $\mathcal{A}_{\mathrm{MS}}$ (similar to the automaton for checking coherence in [Mathur et al. 2019a]) that keeps track of the following information. For every function/pointer f of arity r, and for every tuple $(x_1, \ldots, x_r)$ of variables (of appropriate sorts), each state of the automaton $\mathcal{A}_{\mathrm{checkSC}}$ maintains a boolean predicate denoting whether or not $f(x_1, \ldots, x_r)$ has been computed in any execution that reaches the state. This gives us our next result.

Theorem 9. *The problem of checking whether a given program is streaming-coherent with respect to a given reach specification defining a forest data-structure is decidable in* PSPACE.

7 IMPLEMENTATION AND EVALUATION

We implemented a tool [Mathur et al. 2019d] for deciding memory safety of forest data-structures based on the construction from Section 6. The tool is ~2000 lines of Ocaml 4.07.0 code. It takes as input a program from the grammar presented in Section 2.1.1 annotated with a reachability specification, as in Section 5.1. The tool does not explicitly construct the automaton from Section 6.

Instead, it implements a fixpoint algorithm, described below, which determines the set of states associated with every program point, and uses this set to check for memory safety.

7.1 Fixpoint Algorithm

Here we give a high-level description of the implementation. First, observe that $\mathcal{A}_{\mathsf{MS}}$ has exponentially many states in the number of program variables.We manage this by implementing the transition δ from Section 6 and only building automaton states as they are encountered. For straight-line programs, each transition results in a single new state. For complex programs that use **if**-**then**-**else** and **while** statements, we need to keep track of a bag of states. To see this, suppose we are checking the program $\textbf{if}\,(c)\,\textbf{then}\,s_1\,\textbf{else}\,s_2$. To begin our bag of states contains only the initial state q_0. In order to process the **if**-**then**-**else**, the initial state needs to make two separate transitions, one for each of the two executions generated by the two branches. The bag of states thus grows to include $s_{\mathsf{then}} = \delta(q_0, \text{“}\textbf{assume}(c)\text{”})$ and $s_{\mathsf{else}} = \delta(q_0, \text{“}\textbf{assume}(\neg c)\text{”})$. The branches can then take transitions starting from s_{then} and s_{else}. The union of the reachable states from each branch gives us a new bag of states from which to process the remaining program. The intuition for **if**-**then**-**else** carries over to **while**. From any state in our bag at the beginning of a **while**, we collect the bag of states that results from any number of executions of the loop guard and body, starting from that state. The number of states $|\mathcal{A}_{\mathsf{MS}}|$ is finite, and thus a fixed point is guaranteed. For the benchmarks considered here, the number of states explored by the tool is significantly smaller than the worst case.

If at any point the tool detects a memory safety violation it halts and reports the error. In addition to memory safety, it also monitors the streaming-coherence property as it processes the input program. To do so, it keeps track of terms that were computed using existing equivalence classes, but which were subsequently dropped. If the program attempts to compute the same term using the same classes, the implementation flags a failure of streaming-coherence and halts. For example, to process “$a := f(c)$”, the tool checks that f was not previously applied to $[c]_{\equiv}$ and later dropped. In general, this information can be maintained by remembering the equivalence classes $[c_1]_{\equiv}, \ldots, [c_r]_{\equiv}$ on which any function f (of arity r) has been computed.

The algorithm implemented by the tool is more general than we have described thus far. It can output the set of states reached at the head of a loop. By interpreting individual states as conjunctions of equality and disequality, and the set of states as a disjunction of such conjuncts, we can obtain an inductive invariant that proves memory safety (when the program is memory safe). Any assertion in the form of a Boolean combination of equality statements on program variables can also be checked. This can be accomplished by appending the negated assertion to the end of the program and checking that all reachable states are infeasible.

7.2 Benchmarks

We seek to answer the following basic questions about streaming-coherent programs and our algorithm. First, is it the case that the most natural way to write heap-manipulating single-pass programs on lists and trees results in streaming-coherence? Second, for streaming-coherent programs with and without memory safety violations, is the algorithm able to verify memory safety or find violations of it in the corresponding uninterpreted program? Third, how fast is the algorithm? Note that since we do abstract the primitive types and functions/relations on real programs, it is not clear that the tool will be able to prove memory safe programs as so.

To answer the first and second questions, we wrote natural heap-manipulating programs over singly-linked lists (sorted and unsorted) and tree data-structures (binary search trees, AVL trees, rotations of trees) in our input language, and evaluated the tool on them to determine if they were streaming-coherentand to test for memory safety.

Table 1. Evaluation of tool for proving memory safety and finding memory safety errors

Program	LOC	Streaming-coherent?	Found Safe	# States	Time (ms)
Verification of Memory Safe Programs					
sll-append-safe	19	yes	✓	4	4
sll-copy-all-safe	27	yes	✓	6	3
sll-delete-all-safe	56	yes	✓	58	9
sll-deletebetween-safe	42	yes	✓	53	8
sll-find-safe	16	yes	✓	4	3
sll-insert-back-safe	20	yes	✓	3	3
sll-insert-front-safe	8	yes	✓	1	3
sll-insert-safe	50	yes	✓	12	4
sll-reverse-safe	12	yes	✓	3	3
sll-sorted-concat-safe	17	yes	✓	4	3
sll-sorted-insert-safe	50	yes	✓	12	4
sll-sorted-merge-safe	74	yes	✓	62	8
bst-find-safe	23	yes	✓	21	4
bst-insert-safe	45	yes	✓	29	6
bst-remove-root-safe	52	yes	✓	12	4
avl-balance-safe	190	yes	✓	48	12
tree-rotate-left-safe	25	yes	✓	3	3
Finding Errors in Memory-unsafe Programs					
sll-append-unsafe	20	yes	✗	—	3
sll-copy-all-unsafe	29	yes	✗	—	4
sll-delete-all-unsafe	58	yes	✗	—	5
sll-deletebetween-unsafe	44	yes	✗	—	4
sll-find-unsafe	18	yes	✗	—	3
sll-insert-back-unsafe	20	yes	✗	—	3
sll-insert-front-unsafe	9	yes	✗	—	3
sll-insert-unsafe	50	yes	✗	—	3
sll-reverse-unsafe	12	yes	✗	—	3
sll-sorted-concat-unsafe	17	yes	✗	—	3
sll-sorted-insert-unsafe	50	yes	✗	—	3
sll-sorted-merge-unsafe-1	69	yes	✗	—	4
sll-sorted-merge-unsafe-2	63	yes	✗	—	3
bst-find-unsafe	25	yes	✗	—	4
bst-insert-unsafe	49	yes	✗	—	6
bst-remove-root-unsafe	54	yes	✗	—	3
avl-balance-unsafe	111	yes	✗	—	3
tree-rotate-left-unsafe	20	yes	✗	—	3
Detecting Programs are *not* streaming-coherent					
sll-sorted-merge-non-streaming-coherent	75	no	—	—	4
bst-remove-root-non-streaming-coherent	55	no	—	—	3

The first column of Table 1 gives the set of programs in our benchmark. These are typically *single-pass* algorithms over an input data-structure. For example, finding a key in a binary search tree or in-place reversal of a linked list are single pass algorithms.

The names of the programs indicate whether or not the program truly contains an unsafe memory access (i.e., the ground truth). Programs whose names end in 'unsafe' were obtained by introducing one of two possible memory safety errors into their 'safe' counterparts: (i) attempts to read or write to a location that is unallocated, and (ii) freeing unallocated memory locations.

One example of the first kind is illustrated in sll-copy-all, which copies the contents of a linked list into a freshly allocated list. In this example, the program steps through the input list in a loop until it reaches `NIL`. In each iteration, a new node is allocated, initialized with the contents of the current node, and connected to the end of the new list. The program relies on the invariant that the new list has a next node to step to whenever the old list does. Thus, it does not perform a `NIL` check when advancing along the next pointer for the new list. The sll-copy-all-*unsafe* fails to maintain the invariant by incorrectly adding the freshly allocated node to the new list. An example for errors of the second kind (freeing memory locations that may not be allocated) can be found in sll-deletebetween-unsafe. In this example, the task is to delete all nodes in a linked list that have key values in a certain range. The mistake in this example happens when the program has found a node to delete, but, instead of saving the next node and deleting the current node, it instead frees the next node, which may be unallocated.

7.3 Discussion of Results

Table 1 shows the results for the evaluation, which was performed on a machine running Ubuntu 18.04 with an Intel i7 processor at 2.6 GHz. Columns 3-6 pertain to the operation of the algorithm on the benchmarks. Column 3 indicates whether or not the benchmark fails the streaming-coherence condition. Our tool terminates and identifies memory safety and violation of memory safety on all streaming-coherent programs. Column 4 depicts whether or not an unsafe memory access was detected. Column 5 gives the total number of states that are reachable at the end of the program. Note that non-streaming-coherence and violation of memory safety preclude each other in the table. Upon detecting either, the algorithm halts (and we do not report the number of reachable states). Column 6 gives the total running time of the tool on each benchmark, which is negligible in all cases. Note that the number of reachable states for each example is also quite small relative to the total number of possible states, which grows faster than the Bell numbers. That our algorithm only examines a small fraction of the total state space is encouraging, and suggests that it may scale well for much larger and more complex programs.

8 RELATED WORK

Memory safety errors have attracted a lot of attention because they are serious security vulnerabilities that have been exploited to attack systems [Nagarakatte et al. 2015; Szekeres et al. 2013]; they are one of the most common causes of security bugs [Microsoft 2019]. Memory safety concerns have even led to new programming languages, such as Rust [The Rust Team 2019], that statically assure memory safety (while being efficient). Memory safety vulnerabilities of programs written in C/C++ are still of great concern, and, consequently, identifying fundamental techniques that establish decidability of the problem even for restricted classes of programs is interesting.

There is a rich literature of preventing memory safety errors at runtime by instrumenting code with runtime checks, or at compile time (see [Nagarakatte et al. 2015] and references therein, SafeC [Safe-C 2019], CCured [Austin et al. 1994; Condit et al. 2003; Necula et al. 2005, 2002], Cyclone [Jim et al. 2002], SVA [Criswell et al. 2007], etc.). Static checking for memory safety is certainly possible when it is part of language design (for instance, using type systems as in Rust [Matsakis and Klock 2014]). Dynamic analysis techniques as in [Nethercote and Seward 2007; Rosu et al. 2009; Serebryany et al. 2012] instrument program executables and report errors as they occur during program execution. Recently, there has also been emerging interest in enforcing runtime memory safety using hardware and software support for tagged memory [Joannou et al. 2017; lowRISC 2019; Oleksenko et al. 2018; Serebryany et al. 2018; Watson et al. 2015].

This work stems from the recent decidability result on uninterpreted coherent programs [Mathur et al. 2019a], which has also been extended to incorporate reasoning modulo theories including associativity and commutativity of functions over the data domain and ordering relations on the data domain [Mathur et al. 2019b]. In our work we use automata-theoretic techniques that, over the data domain, reason about equality and function computation over the data elements. Further, for checking memory safety, our procedure tracks a subset of the allocated nodes, namely, the *frontier* nodes. While a considerable portion of the literature on assertion checking and memory safety for heap-manipulating programs is devoted to techniques that compromise on either soundness, completeness, or decidability, there has been some work that aims at decidable reasoning, while still preserving some form of soundness and completeness.

The work in [Alur and Černý 2011] reduces assertion checking of single-pass list-processing programs to questions on data string transducers that work over sequences of tagged data values. The decidablity is mainly a consequence of the fact that there is a *single* variable that advances in a single-pass fashion. In contrast, our work defines a more general notion of single-pass programs using the *streaming-coherence* restriction that still allows for multiple variables to support pointer updates. Further, the work in [Alur and Černý 2011] does not directly apply to verifying memory safety of programs as it does not explicitly handle freeing of memory locations, and the operations of the transducer cannot effect changes to the shape of the heap. Another key difference is that streaming data string transducers only allow reasoning about ordering and equality over data but cannot support more complex reasoning such as the congruence arising from function computation.

The work in [Bouajjani et al. 2006] proposes a class of list programs for which verification is decidable, and crucially relies on the idea of representing fragments of the allocated heap by a bounded number of segments and summaries about them, which is one among many other works [Balaban et al. 2005; Bardin et al. 2004; Berdine et al. 2004; Dor et al. 2000; Manevich et al. 2005; Sagiv et al. 1999] that employ a similar approach. These works can handle limited reasoning with data such as total orders on the data domain, but again, do not support predicates like equality on data or function congruences resulting from equalities, and further, often address questions specific to lists. Our work, on the other hand, tackles the more general problem of the verification of uninterpreted heap programs and instantiates the alias-aware condition to the class of forest data-structures. The key idea of tracking the *frontier* heap locations, which we use for obtaining decidability of streaming-coherent programs, appears orthogonal to this line of work.

The work in [Bozga and Iosif 2007] differs fundamentally from ours in that pointer updates are forbidden, which is a salient feature of our work. Pointer updates are at the heart of the difficulty in building a theory of uninterpreted programs working over heaps. Additionally, that work forbids nesting of loops as well as conditionals within loops, a restriction also used to obtain decidability in earlier work [Godoy and Tiwari 2009] on uninterpreted programs; there is no such syntactic restriction on the class of programs we introduce in this paper.

The work in [Bouajjani et al. 2005] over-approximates the set of heap configurations associated with a given program location as a regular language over a finite alphabet. The program transformations are represented by finite state transducers, and the work employs an abstraction refinement approach for verifying heap-manipulating programs. Since such abstraction refinement loops may not always terminate for arbitrary programs, the proposed approach is only a semi-decision procedure. Further, this work does not support reasoning over the data sort. Other notable static analyses that employ abstraction refinement for verifying heap programs include the notable work on shape analysis [Lev-Ami and Sagiv 2000; Sagiv et al. 1999; Yahav 2001] and automatic predicate abstraction [Ball et al. 2001]. Statically verifying memory safety using such incomplete techniques can of course, and commonly does, result in false positives.

There is a rich literature on program logics for heap verification; in particular separation logics [O'Hearn et al. 2001; Reynolds 2002] and FO logics based on the principles of separation logic [Løding et al. 2019]. Decidable fragments of such logics have also been studied [Berdine et al. 2004, 2006; Cook et al. 2011; Møller and Schwartzbach 2001; Navarro Pérez and Rybalchenko 2011, 2013; Piskac et al. 2013, 2014a,b]. However, typically, these decision procedures are for checking validity of Hoare triples, and the problem of generating loop invariants is often undecidable, as is the problem of completely automatic verification of programs against specifications expressed in these logics. Some invariant generation techniques have been discovered for problems in this domain [Calcagno et al. 2011; Neider et al. 2018], but are, of course, inherently incomplete.

9 CONCLUSIONS AND FUTURE WORK

This paper establishes a foundational result for decidably verifying assertions in programs that update maps for a subclass that is alias-aware and coherent. We have used this general result to develop decidable verification of memory safety for a class of programs, called streaming-coherent programs, working on forest data-structures. We also proved membership of programs in this class is decidable. We showed through a prototype implementation of our decision procedure, and its evaluation on a set of single-pass algorithms, that forest data-structures typically fall in our decidable class, and that we can verify memory safety accurately for them.

The most compelling future direction is to adapt the technique in this paper to provide a memory safety analysis tool for a standard programming language (such as C/C++), handling the rest of the programming language using abstractions (e.g., arrays, allocation of varying blocks of memory, etc.). We believe that our automata-based algorithm will scale well. Realizing the techniques presented herein in a full-fledged memory safety analysis tool would be interesting.

On the theoretical front, there are several interesting directions. First, we could ask how to generalize our results to go beyond streaming-coherent programs on forest data-structures. Although forest data-structures are fairly common as initial heaps for many programs, finding a natural class of heap structures beyond forest data-structures where alias-awareness can be easily established seems an interesting, challenging problem. We believe that data-structures such as doubly-linked lists and trees with parent pointers, and more generally, data-structures that have bounded tree-width with uniform tree decompositions may be amenable to our technique. Tackling multi-pass algorithms on forest data-structures is also an interesting open direction. We believe the best way to look at our work in a larger verification context is that single-pass streaming-coherent programs are the new *basic blocks* that can be completely automatically analyzed, despite the fact that they contain loops. Putting these blocks together to handle programs with multiple passes over data-structures, even in an incomplete fashion, is an interesting future direction.

Another dimension for exploration is to consider more general post-conditions that can be proved automatically and that go beyond simple assertions. One of the limitations of our work is that, though we have an implicit precondition that demands that data-structures are forests, we do not establish that the data-structure in the post state is also a forest. The ability to establish such a property will allow us to maintain the forest property of data-structures as an invariant across multiple calls from clients that manipulate a data-structure using a library of methods, by showing that each of the methods provably maintains this invariant.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of POPL for several comments that helped improve the paper. Umang Mathur is partially supported by a Google PhD Fellowship. This material is based upon work supported by the National Science Foundation under Grants NSF CCF 1901069 and NSF CCF 1527395.

REFERENCES

Rajeev Alur and Pavol Černý. 2011. Streaming Transducers for Algorithmic Verification of Single-pass List-processing Programs. *SIGPLAN Not.* 46, 1 (Jan. 2011), 599–610. https://doi.org/10.1145/1925844.1926454

Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/178243.178446

Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. 2005. Shape Analysis by Predicate Abstraction. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. Springer-Verlag, Berlin, Heidelberg, 164–180. https://doi.org/10.1007/978-3-540-30579-8_12

Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/378795.378846

Sébastien Bardin, Alain Finkel, and David Nowak. 2004. Toward symbolic verification of programs handling pointers. (2004).

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*. Springer-Verlag, Berlin, Heidelberg, 97–109. https://doi.org/10.1007/978-3-540-30538-5_9

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–137.

Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomáš Vojnar. 2006. Programs with Lists Are Counter Automata. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 517–531.

Ahmed Bouajjani, Peter Habermehl, Pierre Moro, and Tomáš Vojnar. 2005. Verifying Programs with Dynamic 1-selector-linked Structures in Regular Model Checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*. Springer-Verlag, Berlin, Heidelberg, 13–29. https://doi.org/10.1007/978-3-540-31980-1_2

Marius Bozga and Radu Iosif. 2007. On Flat Programs with Lists. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'07)*. Springer-Verlag, Berlin, Heidelberg, 122–136. http://dl.acm.org/citation.cfm?id=1763048.1763061

Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. 2003. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. 232–244. https://doi.org/10.1145/781131.781157

Byron Cook, Christoph Haase, Joël Ouaknine, Matthew Parkinson, and James Worrell. 2011. Tractable Reasoning in a Fragment of Separation Logic. In *Proceedings of the 22Nd International Conference on Concurrency Theory (CONCUR'11)*. Springer-Verlag, Berlin, Heidelberg, 235–249. http://dl.acm.org/citation.cfm?id=2040235.2040256

John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram S. Adve. 2007. Secure virtual architecture: a safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 351–366. https://doi.org/10.1145/1294261.1294295

Nurit Dor, Michael Rodeh, and Shmuel Sagiv. 2000. Checking Cleanness in Linked Lists. In *SAS*.

Robert W. Floyd. 1967. Assigning meanings to programs. *Mathematical aspects of computer science* 19, 19-32 (1967), 1. http://www.cs.ucdavis.edu/~su/teaching/ecs240-s09/readings/FloydMeaning.pdf

Guillem Godoy and Ashish Tiwari. 2009. Invariant Checking for Programs with Procedure Calls. In *Proceedings of the 16th International Symposium on Static Analysis (SAS '09)*. Springer-Verlag, Berlin, Heidelberg, 326–342.

Michael Hicks. 2014. What is memory safety?, The Programming Languages Enthusiast. http://www.pl-enthusiast.net/2014/07/21/memory-safety/. Accessed: 2019-04-05.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Trevor Jim, J Greg Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 275–288.

A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos. 2017. Efficient

Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD)*. 641–648. https://doi.org/10.1109/ICCD.2017.112

Tal Lev-Ami and Mooly Sagiv. 2000. TVLA: A System for Implementing Static Analyses. In *Static Analysis*, Jens Palsberg (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 280–301.

Christof Løding, P. Madhusudan, Adithya Murali, and Lucas Pe. 2019. A First Order Logic with Frames. *CoRR* abs/1910.09089 (2019). arXiv:1910.09089 https://arxiv.org/abs/1901.09089

lowRISC. 2019. lowRISC: A fully open-sourced, linux-capable, system-on-a-chip. https://www.lowrisc.org/.. Accessed: 2019-10-29.

Roman Manevich, E. Yahav, G. Ramalingam, and Mooly Sagiv. 2005. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Verification, Model Checking, and Abstract Interpretation*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 181–198.

Umang Mathur, P. Madhusudan, and Mahesh Viswanathan. 2019a. Decidable Verification of Uninterpreted Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 46 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290359

Umang Mathur, P. Madhusudan, and Mahesh Viswanathan. 2019b. What's Decidable About Program Verification Modulo Axioms? *CoRR* abs/1910.10889 (2019). arXiv:1910.10889 https://arxiv.org/abs/1910.10889

Umang Mathur, Adithya Murali, Paul Krogmeier, P. Madhusudan, and Mahesh Viswanathan. 2019c. Deciding Memory Safety for Single-Pass Heap-Manipulating Programs. *CoRR* abs/1910.00298 (2019). arXiv:1910.00298 http://arxiv.org/abs/1910.00298

Umang Mathur, Adithya Murali, Paul Krogmeier, P. Madhusudan, and Mahesh Viswanathan. 2019d. StreamVerif : Automata Based Verification of Uninterpreted Programs. https://github.com/umangm/streamverif.

Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188

Microsoft. 2019. 70 Percent of All Security Bugs Are Memory Safety Issues. https://it.slashdot.org/story/19/02/11/2019247/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues. Accessed: 2019-04-05.

Anders Møller and Michael I. Schwartzbach. 2001. The Pointer Assertion Logic Engine. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 221–231. https://doi.org/10.1145/378795.378851

Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 190–208. https://doi.org/10.4230/LIPIcs.SNAPL.2015.190

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2011. Separation Logic + Superposition Calculus = Heap Theorem Prover. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 556–566. https://doi.org/10.1145/1993498.1993563

Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2013. Separation Logic Modulo Theories. In *Programming Languages and Systems*, Chung-chieh Shan (Ed.). Springer International Publishing, Cham, 90–106.

George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526. https://doi.org/10.1145/1065887.1065892

George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 128–139.

Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. 2018. Invariant Synthesis for Incomplete Verification Engines. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 232–250.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning About Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. Springer-Verlag, London, UK, UK, 1–19. http://dl.acm.org/citation.cfm?id=647851.737404

Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2, Article 28 (June 2018), 30 pages. https://doi.org/10.1145/3224423

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044 (CAV 2013)*. Springer-Verlag New York, Inc., New York, NY, USA, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014a. Automating Separation Logic with Trees and Data. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 711–728.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014b. GRASShopper. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–139.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, Washington, DC, USA, 55–74. http://dl.acm.org/citation.cfm?id=645683.664578

Grigore Rosu, Wolfram Schulte, and Traian Florin Serbanuta. 2009. Runtime Verification of C Memory Safety. In *Runtime Verification (RV'09) (Lecture Notes in Computer Science)*, Saddek Bensalem and Doron A. Peled (Eds.), Vol. 5779. 132–152.

Safe-C. 2019. Safe C Library. https://rurban.github.io/safeclib/doc/safec-3.4/index.html. Accessed: 2019-04-05.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1999. Parametric Shape Analysis via 3-valued Logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/292540.292552

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *CoRR* abs/1802.09517 (2018). arXiv:1802.09517 http://arxiv.org/abs/1802.09517

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

The Rust Team. 2019. The Rust programming language. https://www.rust-lang.org/

Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 20–37. https://doi.org/10.1109/SP.2015.9

Eran Yahav. 2001. Verifying Safety Properties of Concurrent Java Programs Using 3-valued Logic. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/360204.360206