Static Analysis of Heap-Manipulating Low-Level Software

Sumit Gulwani
Microsoft Research, Redmond, WA
sumitg@microsoft.com

Ashish Tiwari
SRI International, Menlo Park, CA
tiwari@csl.sri.com

Abstract

This paper describes a static (intraprocedural) analysis for analyzing heap-manipulating programs (in presence of recursive data structures and pointer arithmetic) in languages like C or low-level code. This analysis can be used for checking memory-safety, memory leaks, and user specified assertions.

We first propose a rich abstract domain for representing useful invariants about such programs. This abstract domain allows representation of must and may equalities among pointer expressions. The integer variables used in the pointer expressions can be existentially as well as universally quantified and can have constraints over some base domain. We allow quantification of a special form, namely $\exists\forall$ quantification. This choice has been made to balance expressiveness with efficient automated deduction. The existential quantification is over some *ghost* variables of programs, which are automatically made explicit by our analysis to express useful program invariants. The universal quantifier is used to express properties of collections of memory locations.

We then show how to perform sound abstract interpretation over this abstract domain. We give transfer functions for performing join, meet, and postcondition operations over this abstract domain. The basis of all these operations is an abstract interpreter for the quantifier-free base constraint domain (eg., the conjunctive domain of linear arithmetic combined with uninterpreted functions). To our knowledge, this is the first abstract interpreter that can automatically deduce first-order logic invariants in programs (without requiring any explicit predicates).

We also present initial experimental results demonstrating the effectiveness of our ideas on some common coding patterns.

Keywords Pointer Analysis, Memory Safety, Abstract Interpretation, First-order logic invariants, Pointer Arithmetic, Recursive data-structure, Arrays

1. Introduction

Alias analysis attempts to answer, for a given program point, whether two pointer expressions e_1 and e_2 are always equal (must-alias) or may be equal (may-alias). Keeping precise track of this information in the presence of recursive data-structures is hard because number of expressions or aliasing relationships become potentially infinite. Presence of pointer arithmetic makes this even harder.

In this paper, we propose a new technique for representing must and may-equalities between pointer expressions with the goal of automatically verifying properties of low-level software that manipulates heap using recursive data structures and pointer arithmetic. This paper has two main contributions. We describe an abstract domain that is expressive enough to capture invariants required to prove correctness of several common code patterns in low-level software Secondly, we describe how to perform automated deduction (in the style of an abstract interpretation) on this abstract domain thereby automatically inferring the invariants required to prove the desired properties.

Our abstract domain essentially represents must-equalities and may-equalities among pointer expressions. It is motivated by the early work on representing aliasing directly using must-alias and may-alias pairs of pointer expressions [2, 18, 4, 7]. However, there are two main differences. (a) The language of our pointer expressions is richer: The earlier work built on constructing pointer expressions from (pre-defined) field dereferences; however our expressions are built from derefencing at arbitrary integer (expression) offsets. This gives our abstract domain the ability to handle arrays, pointer arithmetic, and recursive structures in one unified framework. (b) Apart from the integer program variables, we also allow integer variables (in our expressions) that are existentially or universally quantified. This allows our abstract domain to represent useful properties of data-structures in programs. Our abstract domain is thus significantly richer than those used in earlier work on representing aliasing pairs, which did not allow an explicit quantification. [1]

The appeal of our choice of having only two base predicates of must and may equality (as opposed to a pre-defined set of richer predicates [15, 23, 22, 17]) is comparative simplicity and easier automation. However, in some sense, the power of quantification over the base predicates of must-equality and may-equality makes our domain expressive enough to define a potentially infinite family of base-predicates (for eg., see definitions of useful predicates like `List`, `Array` and `Valid` below), thereby attempting to match the advantages offered by the approach of defining new logics with pre-defined predicates.

We allow only a special form of quantification in our abstract domain, namely $\exists\forall$ quantification - this choice has been made to balance expressiveness with potential for automated deduction. The existential quantification is over some *ghost* variables of the program that are required to express useful program invariants. Such ghost variables are automatically identified by our abstract interpreter. The universal quantifier allows us to describe properties of collections of memory locations.

Consider, for example, the program shown in Figure 1. The input variable x points to a list (unless qualified, list refers to an acyclic singly-linked list in this paper), where each list element contains two fields `Data` and `Len` apart from the `Next` field. `Data` is supposed to be a pointer to some array, and `Len` is intended to be the length of that array. In the first while loop, the iterator y iterates over each list element, initializing `Data` to point to a newly created array and `Len` to the length of that array. In the second while loop, the iterator y iterates over each list element accessing the array pointed to by `Data`. The proof of memory safety of this commonly used code pattern requires establishing the invariant that for all list elements in the list pointed to by x, `Len` is the length of the array

[1] A limited form of quantification over integer variables was implicitly hidden though in the set representation used for representing may-aliases in the work by Deutsch [7].

MSR Technical Report MSR-TR-2006-160, November 2006

```
struct List {int Len, *Data; List* Next;}
ListOfPtrArray(struct List* x)
1  for (y := x; y ≠ null; y := y→next)
2      t :=?; y→len := t; y→data := malloc(4t);
3  for (y := x; y ≠ null; y := y→next)
4      for (z := 0; z < y→len; z := z + 1)
5          y→data→(4z) := ...;
```

Figure 1. An example of a pattern of initializing the pairs of dynamic arrays and their lengths inside each list element and later accessing the array elements.

Data. This invariant is expressed in our abstract domain as

$$\exists i : \texttt{List}(x, i, \texttt{next}) \ \wedge$$
$$\forall j[(0 \le j < i) \Rightarrow \texttt{Array}(x{\rightarrow}\texttt{next}^j{\rightarrow}\texttt{data}, 4 \times (x{\rightarrow}\texttt{next}^j{\rightarrow}\texttt{len}))]$$

where $x{\rightarrow}\texttt{next}^j$ is an (pointer) expression in our language that semantically means using j dereferences at offset next starting from x, and the predicates List and Array are abbreviations for the following definitions.

$$\begin{aligned}\texttt{List}(x, i, \texttt{next}) \quad &\equiv \quad i \ge 0 \ \wedge \ x{\rightarrow}\texttt{next}^i = \texttt{null} \\ &\qquad \wedge \ \forall j[(0 \le j < i) \Rightarrow \texttt{Valid}(x{\rightarrow}\texttt{next}^j)] \\ \texttt{Array}(x, t) \quad &\equiv \quad \forall j[(0 \le j < t) \Rightarrow \texttt{Valid}(x + j)]\end{aligned}$$

Intuitively, $\texttt{List}(x, i, \texttt{next})$ denotes that x points to a list of length i (with next as the next field) and $\texttt{Array}(x, t)$ denotes that x points to a region of memory of length t. The predicate $\texttt{Valid}(e)$ is intended to denote that e is a valid pointer value, which is safe to dereference (provided the subexpressions of e are safe to dereference) [2], and can be encoded as the following must-equality.

$$\texttt{Valid}(e) \quad \equiv \quad e{\rightarrow}\beta = \texttt{valid}$$

where β is a special symbolic integer offset that is known to not alias with any other integer expression, and valid is a special constant in our expression language.[3]

In fact, the loop invariant required to establish this property is even more sophisticated (but still expressible in our abstract domain):

$$\exists i, j' : \texttt{List}(x, i, \texttt{next}) \ \wedge \ 0 \le j' \le i \ \wedge \ y = x{\rightarrow}\texttt{next}^{j'} \ \wedge$$
$$\forall j[(0 \le j < j') \Rightarrow \texttt{Array}(x{\rightarrow}\texttt{next}^j{\rightarrow}\texttt{data}, 4 \times (x{\rightarrow}\texttt{next}^j{\rightarrow}\texttt{len}))]$$

A key contribution of this paper is to describe how to automatically generate such quantified invariants (in order to prove desired program properties) by means of abstract interpretation. Automatic discovery of such quantified invariants is known to be a challenging problem and has only been attempted in domain-specific settings where the predicates that occur in these quantified formulas are explicitly provided [16, 10].

Our approach for discovering such invariants is by means of describing transfer functions to perform a forward abstract interpretation of programs over our quantified abstract domain. We now briefly describe how the above invariant is automatically generated. We denote $\texttt{Array}(x{\rightarrow}\texttt{next}^i{\rightarrow}\texttt{data}, 4 \times (x{\rightarrow}\texttt{next}^i{\rightarrow}\texttt{len}))$ by the notation $S(i)$. For simplicity, assume that the length of the list x is at least 1 and the body of the loop has been unfolded once. The postcondition operator generates the following must-equalities F^1 and F^r (among other must-equalities) before the loop header and after one loop iteration respectively.

$$\begin{aligned}F^1 \quad &= \quad (y = x{\rightarrow}\texttt{next} \wedge S(0)) \\ F^r \quad &= \quad (y = x{\rightarrow}\texttt{next}^2 \wedge S(0) \wedge S(1))\end{aligned}$$

Our join algorithm computes the join of these must-equalities as

$$\exists j' : 1 \le j' \le 2 \wedge y = x{\rightarrow}\texttt{next}^{j'} \wedge \forall j(0 \le j < j' \Rightarrow S(j))$$

which later gets widened to the desired invariant. Note the power of our join algorithm to generate quantified facts from quantifier-free inputs. (See Section 4.2 for more details.)

The algorithm that we describe for the join operation (as well as other transfer functions) reasons about must and may-equalities between pointer expressions and quantifiers directly, while using the transfer functions for the underlying *base constraint domain* to reason about arithmetic constraints between integer expressions. In this architecture, different base constraint domains can be used to achieve differing capabilities. Our preliminary experience suggests a base constraint domain over combination of linear arithmetic (to model pointer arithmetic) and uninterpreted functions (to model dereferences in expressions) to be a good choice (e.g., it can represent the following constraint between integer expressions: $0 \le j < x{\rightarrow}\texttt{next}^i{\rightarrow}\texttt{len}$; this constraint is actually required to represent the above loop invariant). The transfer functions for exactly this domain were recently described by Gulwani and Tiwari [13].

This paper is organized as follows. We first start with a description of our program model, which closely reflects the memory model of C modulo some simple assumptions (Section 2). We then formally describe our abstract domain and present its semantics in relation to our program model (Section 3). We describe the transfer functions for performing an abstract interpretation over this abstract domain in Section 4. Section 5 discusses preliminary experimental results, while Section 6 describes some related work.

2. Program Model

Values A value v is either an integer, or a pointer value, or is undefined. A pointer value is either null or is a pair of a region identifier and a positive offset.

$$v \ ::= \ c \ \mid \ \langle r, d\rangle \ \mid \ \texttt{null} \ \mid \ \bot$$

Program State A *program state* ρ is either undefined, or is a tuple $\langle D, R, V, P\rangle$, where D represents the set of valid region identifiers, R is a *region map* that maps a region identifier in D to a positive integer (denoting size of the region), V is a *variable map* that maps program variables to values, and P is a *memory* that maps non-null pointer values to values.

We say that a pointer value $\langle r, d\rangle$ is *valid* in a program state $\langle D, R, V, P\rangle$ if $r \in D$ and $0 \le d < R(r)$. We say that a pointer value is *invalid* if it is neither valid nor null.

Expressions The program expressions e that occur on the right side of an assignment statement are described by the following language.

$$e \ ::= \ c \ \mid \ x \ \mid \ e_1 \pm e_2 \ \mid \ c \times e \ \mid \ e_1{\rightarrow}e_2 \ \mid \ \texttt{null} \ \mid \ ?$$

$e_1{\rightarrow}e_2$ represents dereference of the region pointed to by e_1 at offset e_2 (i.e., $*(e_1 + e_2)$ in C language syntax). In fact, we sometimes use the notation $*e$ to denote $e{\rightarrow}0$. The above expressions have the usual expected semantics with the usual restrictions that it is not proper to add or subtract two pointer values, and that only a valid pointer value can be dereferenced. ? denotes a non-deterministic integer and is used to conservatively model other program expressions whose semantics we do not precisely capture (eg., those that

[2] This assumption is important because we want to treat Valid as an uninterpreted unary predicate, which allows us to encode it as a simple must-equality. However this necessitates that validity of all valid subexpressions be described explicitly

[3] The variable β denotes a symbolic offset. Without β, the encoding will be unsound as $\texttt{Valid}(e)$ and $\texttt{Valid}(e')$ will then imply $e = e'$ by transitivity of $=$.

$$\frac{}{[\![x]\!]\rho \;=\; V(x)}\ \textsf{var} \qquad \frac{[\![e_1]\!]\rho = \langle r, d\rangle \quad [\![e_2]\!]\rho \text{ is some integer } c}{[\![e_1 \pm e_2]\!]\rho \;=\; \langle r, d \pm c\rangle}\ \textsf{ptrArith} \qquad \frac{[\![e_1]\!]\rho = (r, d) \quad [\![e_2]\!]\rho = c \quad r \in D \quad 0 \le d + c < R(r)}{[\![e_1 \rightarrow e_2]\!]\rho \;=\; P(\langle r, d + c\rangle)}\ \textsf{deref}$$

$$\frac{}{[\![\texttt{null}]\!]\rho \;=\; \texttt{null}}\ \textsf{null} \qquad \frac{}{[\![c]\!]\rho \;=\; c}\ \textsf{cons} \qquad \frac{[\![e_1]\!]\rho \text{ and } [\![e_2]\!]\rho \text{ are both integers}}{[\![e_1 \pm e_2]\!]\rho \;=\; [\![e_1]\!]\rho + [\![e_2]\!]\rho}\ \textsf{intArith} \qquad \frac{\text{Let } c \text{ be a non-det integer}}{[\![?]\!]\rho \;=\; c}\ \textsf{nonDet}$$

$$\frac{\text{rel} \in \{\neq, =\} \quad [\![x_1]\!]\rho, [\![x_2]\!]\rho \text{ are } \texttt{null} \text{ or valid pointer values}}{[\![x_1 \text{ rel } x_2]\!]\rho \;=\; [\![x_1]\!]\rho \text{ rel } [\![x_2]\!]\rho}\ \textsf{ptrComparison} \qquad \frac{[\![x_1]\!]\rho \text{ and } [\![x_2]\!]\rho \text{ are integer values}}{[\![x_1 \text{ rel } x_2]\!]\rho \;=\; [\![x_1]\!]\rho \text{ rel } [\![x_2]\!]\rho}\ \textsf{IntComparison}$$

$$\frac{[\![e]\!]\rho \ge 0 \quad \text{Let } r \text{ be some fresh region identifier}}{[\![x := \text{malloc}(e)]\!]\rho \;=\; \langle D \cup \{r\},\ R[r \mapsto [\![e]\!]\rho],\ V[x \mapsto \langle r, 0\rangle],\ P\rangle}\ \textsf{Malloc} \qquad \frac{V(y) = \langle r, i\rangle \quad r \in D \quad 0 \le i < R(r)}{[\![\text{free}(y)]\!]\rho \;=\; \langle D - \{r\}, R, V, P\rangle}\ \textsf{Free}$$

$$\frac{}{[\![x := e]\!]\rho \;=\; \langle D, R, V[x \mapsto [\![e]\!]\langle D, R, V, P\rangle],\ P\rangle}\ \textsf{varUpdate} \qquad \frac{V(x) = \langle r, i\rangle \quad [\![e_1]\!]\rho = j \quad r \in D \quad 0 \le i + j < R(r)}{[\![x \rightarrow e_1 := e_2]\!]\ \rho \;=\; \langle D, R, V, P[\langle r, i + j\rangle \mapsto [\![e_2]\!]\rho]\rangle}\ \textsf{MemUpdate}$$

Figure 2. Semantics of Expressions, Predicates, and Statements in our language. ρ denotes the state $\langle D, R, V, P\rangle$. An expression takes a program state and returns a value. A predicate takes a program state and returns a boolean value. A statement takes a program state and returns another program state. Evaluation of an expression, or statement in a $\perp$ program state or in any state such that none of the above rules apply yields a $\perp$ value or $\perp$ state respectively.

involve bitwise arithmetic). Given a program state ρ, an expression e evaluates to some value, denoted by $[\![e]\!]\rho$. The formal semantics of these expressions is defined in Figure 2. The expressions which do not match any antecedent of the rules evaluate to $\perp$.

Statements There are two kinds of assignment statements $x := e$ and $*x := e$. Memory is allocated using the malloc statement $x := malloc(e)$ and freed using the free statement $free(e)$. The malloc statement returns a pointer value with a fresh region identifier. The free statement frees the region pointed to by e. The formal semantics of these statements is described in Figure 2. Every statement takes a program state and returns another program state. The statements that do not match any antecedent of the rules yield an undefined program state.

Predicates The predicates that occur in conditionals are of the form x_1 rel x_2, where rel $\in \{<, \le, \neq, =\}$. Without loss of any generality, we assume that x_1 and x_2 are either program variables or constants. These predicates have the usual semantics: Given a program state ρ, a predicate evaluates to either `true` or `false`. Pointer-values can be compared for equality or disequality, while integer values can be compared for any inequality also. Any other comparison results in the predicate evaluating to `true` or `false` in a non-deterministic manner. The formal semantics of these predicates is described in Figure2.

Memory Safety and Leaks We say that a procedure is *memory-safe* and *leak-free* under some precondition, if for any program state ρ satisfying the precondition, the execution of the procedure yields program states $\rho' = \langle D, R, V, P\rangle$ that have the following properties respectively.

- $\rho' \neq \perp$
- For all region identifiers $r \in D$, there exists an expression e such that $[\![e]\!]\rho' = \langle r, d\rangle$.

Intuitively, a procedure is memory-safe if all memory dereferences and free operations are performed on a valid pointer value. Observe that our definition of memory safety precludes dangling pointer dereferences also. Similarly, a procedure is leak-free if all allocated regions can be traced by means of some expression.

Relation with C programs The semantics of our program model closely reflect the C language semantics under the following assumptions: (a) all memory accesses are at word-boundaries and the size of each object read or written is at most a word. (b) The free(x) call frees a valid region returned by malloc even if x points somewhere in middle of that region (Note that, some implementations of C might insist that x point to the beginning of a region returned by malloc.) Our program model may be changed easily to capture other possible semantics of C while not depending on the above assumptions. However, the current choice has been made for simplicity of describing the analysis in our program model. We can thus test if a C program is memory-safe and leak-free by checking for the respective properties in our model.

3. Abstract Domain

The elements of our abstract domain describe must and may equalities between expressions. However, we need a richer language of expressions (as compared to the language of program expressions described in Section 2) to describe useful program properties. Hence, we extend the expression language of our program model to the following:

$$e \;::=\; c \;\mid\; x \;\mid\; e_1 \pm e_2 \;\mid\; c * e \;\mid\; e_1 \rightarrow e_2^{e_3} \;\mid\; \texttt{valid} \;\mid\; \texttt{null}$$

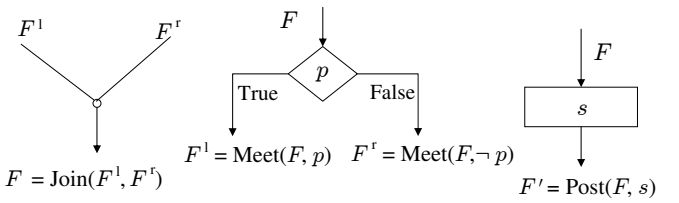
`valid` is a special constant in our domain that satisfies `valid` $\neq$ `null`. The constant `valid` is used to represent that certain expressions contain a valid pointer value (as opposed to null or uninitialized or dangling etc) in the `Valid` predicate defined on Page 2 in Section 1.

The new construct $e_1 \rightarrow e_2^{e_3}$ denotes e_3 de-references of expression e_1 at offset e_2, as is formalized by the following semantics.

$$[\![e_1 \rightarrow e_2^{e_3}]\!]\rho \;=\; \begin{cases} [\![e_1]\!]\rho & \text{if } [\![e_3]\!]\rho = 0 \\ [\![(e_1 \rightarrow e_2) \rightarrow e_2^{e_3 - 1}]\!]\rho & \text{if } [\![e_3]\!]\rho > 0 \end{cases}$$
$$\;=\; \perp, \text{ otherwise}$$

If e_3 is 1, we simply write $e_1 \rightarrow e_2^{e_3}$ as $e_1 \rightarrow e_2$.

Must-equality is a binary predicate over *pointer* expressions denoted using "=" and is used in an infix notation. This predicate



(a) Join Node (b) Conditional Node (c) Statement Node

Figure 3. Abstract interpretation on flowchart nodes

describes equalities between expressions that have the same value at a given program point (in all runs of the program). May-equality is also a binary predicate over sets of *pointer* expressions. It is denoted using "$\sim$" and is used in an infix notation. This predicate is supposed to describe an over-approximation of all possible expression equalities at a given program point (in any run of the program). Disequalities are deduced from absence of (transitive closure of) may-equalities. The reason for keeping may-equalities instead of dis-equalities is that the former representation is often more succinct in the common case when most memory locations are not aliased (i.e., have only one incoming pointer).

3.1 Abstract Elements

An abstract element F in our domain is a collection of must-equalities M, and may-equalities Y, together with some arithmetic constraints C on *integer* expressions. Apart from the program variables, the expressions in M, Y, and C may contain extra integer variables that are existentially or universally quantified. Each must-equality and may-equality is universally quantified over integer variables $U_{\mathtt{f}}$ that satisfy some constraints $C_{\mathtt{f}}$. The collection of these must-equalities M, may-equalities Y and constraints C may further be existentially quantified over some variables. Thus, the abstract element is a $\exists\forall$ formula. This choice balances expressiveness with automation as will be explained later. The constraints C and $C_{\mathtt{f}}$ are some arithmetic constraints over expressions, which are represented in some *base constraint domain* that is a parameter to our algorithm.

$$\begin{array}{lcl} F & ::= & \exists U : C, M, Y \\ M & ::= & \mathtt{true} \mid M \ \wedge \ \forall U_{\mathtt{f}}(C_{\mathtt{f}} \Rightarrow (e_1 = e_2)) \\ Y & ::= & \mathtt{true} \mid Y \ \wedge \ \forall U_{\mathtt{f}}(C_{\mathtt{f}} \Rightarrow (e_1 \sim e_2)) \end{array}$$

The existentially quantified variables, U, can be seen as *ghost* variables of the program that need to be made explicit to express the particular program invariant. The universal quantification allows us to express properties of collections of entities (expressions in our case).

Formal Semantics of Abstract Elements An abstract element F represents a collection of program states ρ, namely those states ρ that *satisfy* F (as defined below). A program state $\rho = \langle D, R, V, P\rangle$ satisfies the formula $F = \exists U : C, M, Y$ (denoted as $\rho \models F$) if there exists an integer assignment σ to variables of U such that the following holds: If $V\sigma$ denotes the result of replacing v by $\sigma(v)$ in V and $\rho_{\mathtt{e}} = \langle D, R, V_{\mathtt{e}}, P\rangle$, then,

- $\rho_{\mathtt{e}} \models consf$, i.e., for each each predicate $e_1 \,\mathrm{rel}\, e_2 \in C$, $[\![e_1 \,\mathrm{rel}\, e_2]\!]\rho_{\mathtt{e}}$ evaluates to `true` whenever it is a valid comparison.
- $\rho_{\mathtt{e}} \models M$, i.e., for all facts $(\forall U_{\mathtt{f}}(C_{\mathtt{f}} \Rightarrow (e_1 = e_2))) \in M$, for all integer assignment $\sigma_{\mathtt{f}}$ to variables in $U_{\mathtt{f}}$, if $\rho_{\mathtt{f}} \models C_{\mathtt{f}}$ then $[\![e_1]\!]\rho_{\mathtt{f}} = [\![e_2]\!]\rho_{\mathtt{f}}$, where $\rho_{\mathtt{f}} = \langle D, R, V_{\mathtt{e}}\sigma_{\mathtt{f}}, P\rangle$. In the special case when $e_1 = e_2$ is of the form $e{\rightarrow}\beta = \mathtt{valid}$, then $[\![e]\!]\rho_{\mathtt{f}} = \langle r, c\rangle$, $r \in D$, and $0 \le c < R(r)$.
- For all expressions e_1 and e_2 such that $[\![e_1]\!]\rho_{\mathtt{e}} = [\![e_2]\!]\rho_{\mathtt{e}} = \langle r, c\rangle$ for some $r \in D$ (i.e., e_1 and e_2 are valid pointer values in the state $\rho_{\mathtt{e}}$), it is the case that for all states ρ' such that $\rho' \models C$ and $\rho' \models M'$, we have $[\![e_1]\!]\rho' = [\![e_2]\!]\rho'$, where M' is the collection of equalities obtained from M and by replacing all may-equalities in Y by must-equalities. In other words, if $e_1 = e_2$ cannot be proved using M', then $\rho_{\mathtt{e}} \models e_1 \neq e_2$.

The top element $\top$ in our abstract domain is represented as:

$$\bigwedge_{x,y} \forall i, j[(x{\rightarrow}?^j) \sim (y{\rightarrow}?^j)]$$

or, equivalently,

$$\bigwedge_{x,y} \forall i_1, i_2, j_1, j_2[(x{\rightarrow}i_1^{j_1}) \sim (y{\rightarrow}i_2^{j_2})]$$

In standard logic with equality and disequality predicates, this would be represented as `true`. However, since we represent the disequality relation by representing its dual, we have to explicitly say that anything reachable from x may be same as anything reachable from y for all pairs of variables x and y.

Observe that the semantics of must-equalities and may-equalities is *liberal* in the sense that a must-equality $e_1 = e_2$ or may-equality $e_1 \sim e_2$ does not automatically imply that e_1 or e_2 are valid pointer expressions. In fact, $e_1 = e_2$ means that either e_1 or e_2 is an invalid pointer-value, or that they have same values. Instead the validity of an expression needs to be explicitly stated using `Valid` predicates (defined on Page 2 in Section 1).

Observe that there cannot be any program state that satisfies a formula whose must-equalities are not a subset of may-equalities. Hence, any useful formula should have any must-equality also as a may-equality. Hence, without loss of generality, we assume that in our formulas all must-equalities are also may-equalities, and avoid duplicate representations in our examples.

3.2 Expressiveness

In this section, we discuss examples of program properties that our abstract elements can express.

- x points to an (possibly `null`) acyclic list.

  $$\exists i : \mathtt{List}(x, i, \mathtt{next})$$

  The predicate `List` is as defined on Page 2.
- x points to a region (array) of t bytes.

  $$\mathtt{Array}(x, t)$$

  The predicate `Array` is as defined on Page 2.
- x points to a cyclic list.

  $$\exists i, j : i \ge 0, j \ge 1, x{\rightarrow}\mathtt{next}^i = x{\rightarrow}\mathtt{next}^{i+j} \ \wedge \ \forall k(0 \le k < j \ \Rightarrow \ \mathtt{Valid}(x{\rightarrow}\mathtt{next}^k))$$
- The lists pointed to by x and y *are* shared.

  $$\exists i, j : i \ge 0, j \ge 0, x{\rightarrow}\mathtt{next}^i = y{\rightarrow}\mathtt{next}^j$$
- y *may* point to some node in the list pointed to by x.

  $$\exists i : x{\rightarrow}\mathtt{next}^i \sim y \quad \text{or, equivalently,} \quad \forall i(x{\rightarrow}\mathtt{next}^i \sim y)$$

  Observe that existential quantification and forall quantification over may-equalities has the same semantics.
- The (reachable) heap is completely disjoint, i.e., no two distinct reachable memory locations point to the same location.

  $$\mathtt{true}$$

```
MustAliases(e, F)
  A := {⟨true, e⟩}
  While change in A and not tired
     Forall (∀V(C ⇒ e1 = e2)) ∈ F and ⟨C', e'⟩ ∈ A
        If ((σ, γ) := MatchExpr(e', e1) ≠ ⊥
           A := A ∪ {⟨C' ∧ Cσ, (e2σ)→γ⟩}
  return A

MayAliases(e, F)
  A := {⟨true, e⟩}
  While change in A
     Forall (∀V(C ⇒ e1 ∼ e2)) ∈ F and ⟨C', e'⟩ ∈ A
        If ((σ, γ) := MatchExpr(e', e1) ≠ ⊥
           A := OverApprox(A ∪ {⟨C' ∧ Cσ, (e2σ)→γ⟩})
  return A
```

(a) Algorithm

Inputs:
$e = x$
$M_1 = (x = x{\rightarrow}\texttt{next}^j)$
$M_2 = (\forall j((0 \le i \le j) \Rightarrow x{\rightarrow}\texttt{next}^i = x{\rightarrow}\texttt{next}^{i+1}{\rightarrow}\texttt{prev}))$

$\texttt{MustAliases}(e, F_1) = \{x{\rightarrow}\texttt{next}^j, x{\rightarrow}\texttt{next}^{2j}\}$
$\texttt{MustAliases}(e, F_2) = \{x{\rightarrow}\texttt{next}{\rightarrow}\texttt{prev},\ x{\rightarrow}\texttt{next}{\rightarrow}\texttt{prev}{\rightarrow}\texttt{next}{\rightarrow}\texttt{prev}\}$

$\texttt{MayAliases}(e, F_1) = \{x{\rightarrow}\texttt{next}^t \mid t \ge j\}$
$\texttt{MayAliases}(e, F_2) = \{x{\rightarrow}(\texttt{next}\|\texttt{prev})^t \mid 0 \le t\}$ or
$\{x{\rightarrow}(t_1)^{t_2} \mid 0 \le t_2 \wedge \ell \le t_1 \le u\}$
where $\ell = \min(\texttt{next}, \texttt{prev})$ and $u = \max(\texttt{next}, \texttt{prev})$.

(b) Examples

Figure 4. The two important functions `MustAliases` and `MayAliases` on which the precision of our transfer functions depend. In (b), the first choice for $\texttt{MayAliases}(e, F_2)$ is better than the second choice (if the `next` and `prev` fields are not laid out successively), but will be generated only if we allow disjunctive offsets, as addressed in Section 3.2. Note that even though `MayAliases` is a conservative overapproximation it helps us prove that x does not alias with for example $x{\rightarrow}\texttt{data}$.

Observe that disjointedness comes for free in our representation, i.e., we do not need to say anything if we want to represent disjointedness.

- y may be reachable from x, but only by following `left` or `right` pointers. [Such invariants are useful to prove that certain iterators over data-structures do not update certain kinds of fields.] The expression language as described earlier, is currently insufficient to represent this invariant precisely. However, a simple extension to our expression language in which we allow disjunctions of offsets (as opposed to a single offset) can represent this invariant precisely as follows.

$$\forall i \ge 0 : x{\rightarrow}(\texttt{left}\|\texttt{right})^i \sim y$$

The semantics of the abstract domain and the analyses described in this paper can be easily extended to accomodate disjunctive offsets as above. However, we avoid a formal treatment of disjunctive offsets in this paper for purpose of simplified notation.

3.3 Limitations

Following are some examples of program properties that we cannot express in our abstract domain.

Flattening of a list of buffers of varying sizes: We came across some networking code that tries to flatten out a list of buffers of varying size into a single buffer. The invariant required to prove memory safety of such a code need to relate the output buffer size with the buffers in the list requires using a summation notation, which we cannot represent. However, we feel that not many real applications use such patterns.

Disjunctive Properties: We cannot represent arbitrary disjunctive pointer equalities like $x = z{\rightarrow}\texttt{next} \vee x = y$ since our abstract domain does not support explicit disjunction (for efficiency reasons). We plan to add disjunctive support on top of our abstract domain in the future. However, our domain can express certain kinds of disjunctive properties that can be implicitly specified using existential quantification. For example, $x = z{\rightarrow}\texttt{next} \vee x = z$ can be represented as $\exists i : 0 \le i \le 1 \ \wedge \ x = z{\rightarrow}\texttt{next}^i$.

We also cannot express invariants that require $\forall\exists$ quantification, such as the invariants required to analyze the Schorr-Waite algorithm [14]. We plan to enrich our abstract domain in future.

4. Abstract Interpretation

In this section, we describe how to automatically infer some program properties that are expressible in our abstract domain (described in Section 3) by performing a forward abstract interpretation over the program [5]. This involves computing abstract elements at each program point from the abstract elements at the preceding program points by means of different transfer functions as shown in Figure 3. The transfer function for the postcondition operator also checks for any memory safety errors or memory leaks. We assume that the base domain for representing constraints comes equipped with the standard transfer functions $\texttt{Meet}_{\texttt{base}}$, $\texttt{Join}_{\texttt{base}}$ and $\texttt{Post}_{\texttt{base}}$. We use these transfer functions to construct the transfer functions for our quantified abstract domain.

The abstract element at the entry point is initialized to the given pre-condition. The transfer functions used for computing the abstract elements at other program points are described in the subsections below. In presence of loops, the abstract interpreter goes around loops until fixed-point is reached. Since the transfer functions that we describe are not the most-precise abstract transformers (which is not a surprise since our abstract domain does not form a lattice), there are possibilities of unwanted precision loss especially at join points of loop headers. We use the heuristic of unfolding one iteration of all loops; this introduces interesting data-structure access patterns in our abstract elements and helps to avoid unwanted precision loss.

We begin by describing some functions that do transitive reasoning of must and may equalities, which is an integral part of all our transfer functions.

4.1 Transitive Inference of Must and May Aliases

One common operation required in our transfer functions is an explicit representation of (underapproximation of) must-aliases and (over-approximation of) may-aliases of a given expression that are implied by a given abstract element. For this purpose, we define the following functions.

The function $\texttt{MustAliases}(e, F)$ returns an under-approximation of all must-aliases of expression e such that for every $e' \in \texttt{MustAliases}(e, F)$, we can deduce that $F \Rightarrow e = e'$. Similarly, the function $\texttt{MayAliases}(e, F)$ returns an over-approximation of all may-aliases of expression e such that if $F \Rightarrow e \sim e'$, then $e' \in \texttt{MayAliases}(e, F)$. Since these alias sets may have an infinite number of expressions, we represent the alias sets of an ex-

```
Join(F^l, F^r)
1  F^l := Normalize(F^l)
2  F^r := Normalize(F^r)
3  Let F^l = ∃U^l : C^l, M^l, Y^l
4  Let F^r = ∃U^r : C^r, M^r, Y^r
5  M' := true;  C := true
6  Forall quantifier-free (e1 = e2) ∈ M^r
7      C_t := MustMatch(F^l, e1 = e2)
8      If M^l ∧ C_t is satisfiable
9          C := C ∧ C_t
10         M' := M' ∧ (e1 = e2)
11 Repeat Lines 6-10 with F^l and F^r swapped
12 U' := U^l ∪ U^r
13 C' := Join_base(C^l ∧ C, C^r ∧ C)
14 Forall (∀U1(C1 ⇒ e1 = e2)) ∈ M^r
15     C2 := MustMatch(F^l, e1 = e2)
16     C3 := ModuloMeet_base((C_t, C^l ∧ C), (C1, C^r ∧ C))
17     M' := M' ∧ (∀U1(C3 ⇒ e1 = e2))
18 Repeat Lines 14-17 with F^l and F^r swapped
19 Y' := Y^l ∧ Y^r
20 return ∃U' : C', M', Y'
```

(a) Algorithm

Inputs:

$F^{\mathtt{l}} \;=\; (y = x{\rightarrow}\mathtt{next}) \wedge \forall h[(0 \le h < x{\rightarrow}\mathtt{len}) \Rightarrow \mathtt{Valid}(x{\rightarrow}\mathtt{data} + h)]$

$F^{\mathtt{r}} \;=\; (y = x{\rightarrow}\mathtt{next}^2) \wedge \forall h[(0 \le h < x{\rightarrow}\mathtt{len}) \Rightarrow \mathtt{Valid}(x{\rightarrow}\mathtt{data} + h)] \wedge$
$\forall h[(0 \le h < x{\rightarrow}\mathtt{next}{\rightarrow}\mathtt{len})] \Rightarrow \mathtt{Valid}(x{\rightarrow}\mathtt{next}{\rightarrow}\mathtt{data} + h)$

Trace of $\mathtt{Join}(F^{\mathtt{l}}, F^{\mathtt{r}})$:

$\mathtt{Normalize}(F^{\mathtt{l}}) \;=\; \exists i_1 : i_1 = 1 \;\wedge\; y = x{\rightarrow}\mathtt{next}^{i_1} \;\wedge$
$\forall h_1 : (0 \le h_1 \le x{\rightarrow}\mathtt{len}) \Rightarrow \mathtt{Valid}(x{\rightarrow}\mathtt{data} + h_1)$

$\mathtt{Normalize}(F^{\mathtt{r}}) \;=\; \exists i_2 : i_2 = 2 \;\wedge\; y = x{\rightarrow}\mathtt{next}^{i_2} \;\wedge$
$\forall j, h_2 : (0 \le j \le 1, 0 \le h_2 \le x{\rightarrow}\mathtt{next}^j{\rightarrow}\mathtt{len}) \Rightarrow$
$\mathtt{Valid}(x{\rightarrow}\mathtt{next}^j{\rightarrow}\mathtt{data} + h_2)$

First loop corresponding to $y = x{\rightarrow}\mathtt{next}^{i_2}$ from $M^{\mathtt{r}}$
$C_t = (i_2 = i_1)$

$C = (i_2 = i_1)$ (after line 11)

Second loop corresponding to $\mathtt{Valid}(x{\rightarrow}\mathtt{next}^j{\rightarrow}\mathtt{data} + h_2)$ from $M^{\mathtt{r}}$
$C_2 = (j = 0 \;\wedge\; 0 \le h_2 \le x{\rightarrow}\mathtt{len})$
$C_3 = (0 \le j < i \;\wedge\; 0 \le h' \le x{\rightarrow}\mathtt{next}^j)$

$C' = (1 \le i_1 = i_2 \le 2)$

Output:

$\exists i_1, i_2 : 1 \le i_1 = i_2 \le 2,\; y = x{\rightarrow}\mathtt{next}^{i_2} \;\wedge$
$\forall j, h' : (0 \le j \le 1, 0 \le h_2 \le x{\rightarrow}\mathtt{next}^j{\rightarrow}\mathtt{len}) \Rightarrow \mathtt{Valid}(x{\rightarrow}\mathtt{next}^j{\rightarrow}\mathtt{data} + h_2),$
$Y^{\mathtt{l}} \wedge Y^{\mathtt{r}}$

(b) Example

Figure 5. The Join Algorithm uses $\mathtt{Join}_{\mathtt{base}}$ and $\mathtt{ModuloMeet}_{\mathtt{base}}$ operations from the base domain. The example instance shown in (b) arises in verification of the procedure `ListOfPtrArray` shown in Figure 1. The result of the join operation introduces new quantifiers in the output, which later yields the desired quantified invariant required to prove the memory safety of the second loop in procedure `ListOfPtrArray`.

pression e using a finite set of pairs (C, e'), where (C, e') denotes all expressions e' that satisfy the constraints C. [4]

The pseudo-code for `MustAliases` and `MayAliases` is described in Figure 4. The precision of our analysis depends on the precision of `MustAliases` and `MayAliases`. The key idea in our algorithm for `MustAliases` is to do transitive inference from must-equalities for some constant number of times. The key idea in `MayAliases` is to do transitive inference from may-equalities until fixed-point is reached, and use some function `OverApprox` for over-approximating the elements in the set that guarantees termination in a bounded number of steps. One such heuristic for function `OverApprox` may be to bound the size of the expressions, and the constants that occur in the constraints. (Similar widening techniques have been used for over-approximating regular languages [24].) The function $\mathtt{MatchExpr}(e', e_1)$ returns a substitution σ (for the universally quantified variables in e_1) and a subterm γ such that e' and $e_1\sigma{\rightarrow}\gamma$ are syntactically equal, whenever such a substitution exists; otherwise, it returns $\perp$.

Observe that the above algorithm for `MustAliases` lacks the capability for inductive reasoning. For example, even if the transitive inference goes on for ever, it cannot deduce, for example, that $x{\rightarrow}\mathtt{next}^i{\rightarrow}\mathtt{prev}^i$ is a must-alias of x (where i is some program variable that does not have a constant value). However, we feel that such an inference is not usually required.

Based on the functions `MustAliases` and `MayAliases`, we can also easily define two other functions that are used by our transfer functions: $\mathtt{MustMatch}(F, e_1 = e_2)$ returns an underapproximation of the constraints required to ensure $e_1 = e_2$, i.e., if $C = \mathtt{MustMatch}(F, e_1 = e_2)$, then $C \wedge F \Rightarrow e_1 = e_2$. Similarly, the function $\mathtt{MayMatch}(Y, e_1 \sim e_2)$ returns an over-approximation of the constraints that ensure $e_1 \sim e_2$, i.e., if $C = \mathtt{MayMatch}(Y, e_1 \sim e_2)$, then $\neg C \wedge Y \Rightarrow e_1 \neq e_2$.

The function `MustMatch` can be implemented by computing $A_1 = \mathtt{MustAliases}(M, e_1)$ and $A_2 = \mathtt{MustAliases}(M, e_2)$ and computing the disjunction of the constraints $C_1 \wedge C_2 \wedge \sigma$ for all $(C_1, e_1') \in A_1$ and $(C_2, e_2') \in A_2$ where e_1' and e_2' have some common intersection under some substitution σ that relates the free variables (i.e., the variables that do not occur in e_1,e_2 and M) in e_1' and e_2'. The functions `MayMatch` can be implemented in similar manner by using the `MayAliases` information instead of the `MustAliases` information.

4.2 Join Algorithm

The join algorithm takes as input two abstract elements $F^{\mathtt{l}}$ and $F^{\mathtt{r}}$ and outputs an abstract element F with the following property.

PROPERTY 1 (Soundness of Join Algorithm).
Let $F = \mathtt{Join}(F^{\mathtt{l}}, F^{\mathtt{r}})$ and let ρ be any program state. Then,

$$(\rho \models F^{\mathtt{l}}) \vee (\rho \models F^{\mathtt{r}}) \quad \Longrightarrow \quad \rho \models F$$

The join algorithm is used to merge the dataflow facts at join points. Note that Property 1 does not specify the behavior of the join algorithm completely (since the join algorithm can trivially return $\top$). However, a stronger F that satisfies the above property is a better solution in the sense that it makes our analysis more precise. The pseudo-code for the `Join` algorithm is described in Figure 5.

We have designed our join algorithm to (possibly) introduce new quantifiers in the output so that useful quantified invariants can be generated. We explain the key ideas in the `Join` algorithm by means of an example.

[4] This representation is motivated by the one used by Deutsch [7] except that the constraints in his formalism were pure linear arithmetic facts with no support for uninterpreted function subterms, and the expressions did not have support for pointer arithmetic. Moreover Deutsch used this representation only for computing may-aliases, and there was no support for must-aliases in his framework.

```
Meet(F, x1 rel x2)
1   Let F = (∃U : C, M, Y)
2   C' := x1 rel x2;
3   If rel = ``=''
4     Ct := MayMatch(F, x1 ∼ x2)
5     C' := C' ∧ Ct
6     M' := M ∧ x1 = x2
7     Y' := Y ∧ x1 ∼ x2
8   Else if rel = ``≠''
9     Ct := MustMatch(F, x1 = x2)
10    C' := C' ∧ ¬Ct
11    M' := M
12    Y' := Y − {x1 ∼ x2}
13  Return ∃U : C', M', Y'
```

(a) Algorithm

Inputs:	$F = \exists i : \texttt{len} \leq i \wedge \texttt{List}(x, i, \texttt{next}) \wedge y = x{\rightarrow}\texttt{next}^{\texttt{len}}$ $(x_1 \text{ rel } x_2) = (y = \texttt{null})$
After Line 4:	$C_t = (\texttt{len} = i)$
Output:	$\exists i : \texttt{len} = i \wedge \texttt{List}(x, i, \texttt{next}) \wedge y = x{\rightarrow}\texttt{next}^{\texttt{len}}$
Inputs:	$F = \exists i : \texttt{len} \leq i \wedge \texttt{List}(x, i, \texttt{next}) \wedge y = x{\rightarrow}\texttt{next}^{\texttt{len}}$ $(x_1 \text{ rel } x_2) = (y \neq \texttt{null})$
After Line 9:	$C_t = (len = i)$
Output:	$\exists i : \texttt{len} < i \wedge \texttt{List}(x, i, \texttt{next}) \wedge y = x{\rightarrow}\texttt{next}^{\texttt{len}}$

(b) Examples

Figure 6. The `Meet` algorithm involves reasoning about the interaction between equalities and dis-equalities. The example instances shown in (b) arise in verification of the program `List2Array` shown in Figure 10. The result of the first example above is critical in establishing that the list pointed to by x has length `len`, which in turn is used in proving the safety of memory operations in the second loop in procedure `List2Array`. The result of the first example above is used in checking the safety of deference of pointer y in the first loop in the procedure.

Consider the program shown in Figure 1. Performing abstract interpretation over this program in our setting will give rise to the facts F^{l} and F^{r} (shown in Figure 5(b)) on the two predecessors of the loop header (after unfolding one loop iteration).

In the first step of join computation, the two input facts are rewritten into an equivalent standardized form using the `Normalize` operator described later. This representation is obtained by introducing dummy variables. These variables are suitably quantified and matched up in the two inputs. (A more detailed description of this step is provided later in this sub-section.)

$$\begin{aligned} F^{\text{l}} \;=\; & \exists i : i = 1 \;\wedge\; y = x{\rightarrow}\texttt{next}^{i} \;\wedge \\ & \forall j, h : (j = 0 \wedge 0 \leq h \leq x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{len}) \Rightarrow \\ & \qquad \texttt{Valid}(x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{data} + h) \\ F^{\text{r}} \;=\; & \exists i : i = 2 \;\wedge\; y = x{\rightarrow}\texttt{next}^{i} \;\wedge \\ & \forall j, h : (0 \leq j \leq 1 \wedge 0 \leq h \leq x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{len}) \Rightarrow \\ & \qquad \texttt{Valid}(x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{data} + h) \end{aligned}$$

Then the join of F^{l} and F^{r} can be obtained by using the observation that the disjunction of the following two logical formulas

$$C_1 \wedge \forall U(C_1' \Rightarrow E) \qquad C_2 \wedge \forall U(C_2' \Rightarrow E)$$

can be overapproximated as

$$\texttt{Join}_{\texttt{base}}(C_1, C_2) \wedge \forall U((C_1' \wedge C_2') \Rightarrow E)$$

where $\texttt{Join}_{\texttt{base}}(C_1, C_2)$ overapproximates $C_1 \vee C_2$. However, this formula is very weak: a naive conjunction of C_1' and C_2' sometimes generates too strong constraints. An interesting thing to note is that we can use any underapproximation of

$$(C_1 \Rightarrow C_1') \wedge (C_2 \Rightarrow C_2')$$

in place of $C_1' \wedge C_2'$ without losing soundness. We implement this operation over the constraints domain by means of an operator that we call $\texttt{ModuloMeet}_{\texttt{base}}$ (This operator can be implemented using $\texttt{Join}_{\texttt{base}}$, and is described in Figure 7). For the above example, we have:

$$\begin{aligned} \texttt{Join}_{\texttt{base}}(C_1, C_2) \;&=\; (1 \leq i \leq 2) \\ \texttt{Meet}_{\texttt{base}}(C_1', C_2') \;&=\; j = 0 \wedge 0 \leq h \leq x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{len} \end{aligned}$$

$$\begin{aligned} \texttt{ModuloMeet}_{\texttt{base}}((C_1', C_1), (C_2', C_2)) \;=\; & \\ & 0 \leq j \leq i \wedge 0 \leq h \leq x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{len} \end{aligned}$$

Hence, by using the results of the join and modulo-meet operator, we obtain the desired result as:

$$\begin{aligned} \texttt{Join}(F^{\text{l}}, F^{\text{r}}) \;=\; & \exists i : 1 \leq i \leq 2 \;\wedge\; y = x{\rightarrow}\texttt{next}^{i} \;\wedge \\ & \forall j, h[(0 \leq j < i \wedge 0 \leq h \leq x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{len}) \Rightarrow \\ & \qquad \texttt{Valid}(x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{data} + h)] \end{aligned}$$

The psuedo-code for the join algorithm is described in Figure 5. The `Normalize` operator introduces fresh variables at the exponents and offset positions in the expressions and existentially or universally quantifies them as per the strategy described below. We make two copies of each (universal) quantifier-free must-equality. In one of them, all fresh variables are existentially quantified and in the other all fresh variables are universally quantified. The fresh variables in already quantified facts are all universally quantified. The rationale behind this choice is as follows. The existential variables are supposed to represent some ghost variables that are not made explicit in the program, and we target those ghost variables that can be defined by a simple (i.e., quantifier-free) must-equality. Hence we existentially quantify the fresh variables in only the quantifier-free facts. Once the existentially quantified variables have been discovered, the problem reduces to finding all universally quantified facts and hence we universally quantify all the fresh variables in all the must-equalities. The second step in the `Normalize` operator is to group together the constraints of matching must-equalities by using the rule that $\forall U(C_1 \Rightarrow E)$ and $\forall U(C_2 \Rightarrow E)$ can be written as $\forall U(C_3 \Rightarrow E)$, where $C_3 = C_1 \vee C_2$ provided C_3 is expressible in the constraint domain. Figure 5(b) gives an example of the normalization step. However, we introduce only some fresh variables (to avoid cluttering in the example) that are enough to obtain the desired result.

Once the inputs have been normalized, we match the must-equalities in the two inputs. Lines 6-11 in the pseudo-code for `Join` algorithm shown in Figure 5 relate the existentially quantified variables in the inputs and output the result in constraint C. This is done by means of matching the non-universally quantified must-facts. Essentially such facts define the witnesses for the existentially quantified variables. The constraints C' on the existentially quantified variables are a join of the constraints on the existentially quantified variables C^{l} and C^{r} in presence of C.

Lines 15-17 relate the variables in some matching universally quantified must-equality in the inputs to obtain the constraints C_2 under which the other input also satisfies the same must-equality. The constraints C_3 of that must-equality in the output are then ob-

```
ModuloMeet_base((C1, C1), (C2, Cr))
1  C := Join_base(C1 ∧ C1, C2 ∧ Cr)
2  Forall Ce ∈ C1:
3     If C ∧ C1 ⇏ Ce, C := C ∧ Ce
4  Forall Ce ∈ C2:
5     If C ∧ Cr ⇏ Ce, C := C ∧ Ce
6  Return C
```

(a) Algorithm

Inputs:

$(C_1, C^{\mathtt{l}}) \quad = \quad (j_1 = 0 \ \wedge \ j_2 \leq 5, \ i = 1)$

$(C_2, C^{\mathtt{r}}) \quad = \quad (0 \leq j_1 \leq 1 \ \wedge \ j_2 \geq 5, \ i = 2)$

Trace of $\mathtt{ModuloMeet}_{\mathtt{base}}((C_1, C^{\mathtt{l}}), (C^{\mathtt{r}}, C^{\mathtt{r}}))$:

After line 1: $C \quad = \quad 0 \leq j_1 < i \ \wedge \ 1 \leq i \leq 2$

After first loop: $C \quad = \quad 0 \leq j_1 < i \ \wedge \ 1 \leq i \leq 2 \ \wedge \ j_2 \leq 5$

After second loop: $C \quad = \quad 0 \leq j_1 < i \ \wedge \ 1 \leq i \leq 2 \ \wedge \ j_2 = 5$

(b) Example

Figure 7. The $\mathtt{ModuloMeet}_{\mathtt{base}}$ operator is an important component of the `Join` algorithm. It returns an underapproximation of $(C_1 \Rightarrow C^{\mathtt{l}}) \wedge (C_2 \Rightarrow C^{\mathtt{r}})$ (as opposed to the standard $\mathtt{Meet}_{\mathtt{base}}$ operator that returns $C_1 \wedge C_2$). The algorithm described in (a) gives an implementation of the $\mathtt{ModuloMeet}_{\mathtt{base}}$ operator in terms of $\mathtt{Join}_{\mathtt{base}}$.

tained by taking meet of the constraints C_1 and C_2 but in presence of environments $C_l \wedge C$ and $C_r \wedge C$ respectively, which is achieved by means of the $\mathtt{ModuloMeet}_{\mathtt{base}}$ operator (described in Figure 7). Line 19 defines Y' as simply the union of the may-equalities in the two inputs.

4.3 Meet Algorithm

The meet algorithm takes as input an abstract element F and a predicate x_1 rel x_2 and returns an abstract element F' that has the following property.

PROPERTY 2 (Soundness of Meet Algorithm). *Let F' be $\mathtt{Meet}(F, x_1 \ rel \ x_2)$, and let ρ be any program state. Then,*

$$\rho \models F \ \wedge \ (\llbracket x_1 \ rel \ x_2 \rrbracket \rho = \mathtt{true}) \ \Rightarrow \ \rho \models F'$$

The meet algorithm is used to collect information from the predicates in the conditionals. We describe an implementation of the Meet Algorithm in Figure 6 and also give some examples. The key idea in the `Meet` algorithm is to reason about the interaction between equalities and disequalities. When the input predicate is an equality, we assert that the corresponding disequality cannot be true, i.e., the equality should be a may-equality in the constraints. The call to procedure `MayMatch` in Line 4 generates these constraints.

Similarly, when the input predicate is a disequality, we assert that the corresponding equality cannot be true. The call to procedure `MustMatch` in Line 9 generates these constraints and we assert their negation. Since $\neg C_t$ may not be representable in the base constraint domain, by the operation $C \wedge \neg C_t$ in Line 10, we simply mean any overapproximation of it that is representable in the constraint domain. Line 12 removes the may-equality $e_1 \sim e_2$ from Y (if it occurs in Y). If $e_1 \sim e_2$ occurs as a part of a quantified equality in Y, then it is also sound to strengthen the corresponding constraints in the quantified may-equality just enough so they no longer imply $e_1 \sim e_2$.

Note that in absence of typing information, we do not know whether an expression is an integer or pointer expression, so all predicates are also added to the base constraint, which maintains relationships between integer expressions.

4.4 PostCondition Algorithm

The postcondition algorithm takes as input an abstract element F and an assignment statement s and returns another abstract element F', along with a possible error message, with the following property.

PROPERTY 3 (Soundness of PostCondition Algorithm). *Let F be an abstract element and s be an assignment statement such that $\mathtt{Post}(F, s)$ does not output any memory safety violation. Let $F' = \mathtt{Post}(F, s)$ and let ρ be any program state. Then,*

$$\rho \models F \ \Rightarrow \ \llbracket s \rrbracket \rho \neq \bot \ and \ \llbracket s \rrbracket \rho \models F'$$

Postcondition computation across assignments of the form $*x := e$ involves invalidating must- and may-equalities, and adding the new equality established by the assignment. Any must-equality and base constraint that involves expressions whose prefix $*e'$ is such that e' is a may-alias of x are invalidated. In the example shown in Figure 8, the must fact $\mathtt{List}(y, i, \mathtt{next})$ is invalidated since we have $y + 4 \sim x$. We are then left with only the may analogues, $y \rightarrow \mathtt{next}^i \sim \mathtt{null}$ and $\forall j[(0 \leq j \leq i) \Rightarrow y \rightarrow \mathtt{next}^i \rightarrow \beta \sim \mathtt{valid}$ (not shown in figure), of $\mathtt{List}(y, i, \mathtt{next})$ after this step. Next, any may-equality that involves expressions whose prefix $*e'$ is such that e' is a must-alias of x is also eliminated. Since $y + 4 = x$, it follows that these two may facts are removed. This step is the equivalent of performing a strong update.

However, before invalidating any must-equality, the procedure checks to see if there is any must-alias of $*x$ that can be used as a placeholder for $*x$. If such an expression e' exists, then the procedure replaces $*x$ by e' in the facts. In Figure 8, this first step leads to the addition of the fact $\mathtt{List}(\text{tmp}, i - 1, \mathtt{next})$ since $*x = \text{tmp}$ is a must-alias equation. In the absence of the existence of such a must-alias e', the procedure warns of a "potential memory leak". Moreover, if the procedure fails to find a may-alias e' for $*x$, then it signals a "definite memory leak". In the example, the removal of the equation $*x = \text{tmp}$ from the must and/or may equation set leads to these situations. As a first step, the procedure also checks if the input abstract element F implies $\mathtt{Valid}(x)$ and $\mathtt{Valid}(e')$ for every e' that occurs as $*e'$ in e. Failure of this check indicates "potential memory safety violation". The failure of the dual check of may-valid equations will indicate "definite memory safety violation". (The "definite" messages assume that the program reaches that program point.)

The procedures for computing postconditions across assignments $*x = \text{malloc}(e)$ is similar to the procedure above, except that the new equation added in the final stage is $\forall j(0 \leq j < e \ \Rightarrow \ \mathtt{Valid}(x + j))$. The post for $free(x)$ invalidates all expressions that contain subexpressions that are may aliased to $x + i$, where i is an integer offset.

4.5 Correctness and Complexity

THEOREM 1. *Let F_1 and F_2 be the precondition and postcondition respectively of some procedure in our program model. Then, if our abstract interpreter does not output any memory safety violation and verifies the postcondition, then for all program states ρ such that $\rho \models F_1$ the following holds: Execution of the procedure in state ρ yields a not-$\bot$ state ρ' (i.e., there are no memory safety violations) such that $\rho' \models F_2$.*

The soundness of Theorem 1 follows easily from Properties 3, 2, and 1, which state the soundness of the individual transfer-functions of the abstract interpreter. Also, it is not difficult to verify that the pseudo-code described for the transfer functions satisfies the respective properties.

Stages in computing post across $*x := \texttt{result}$				
Initial fact	Replace $*x$ by tmp	Invalidate must	Invalidate may	Add new fact
$\texttt{List}(y, i, \texttt{next})$	$\texttt{List}(y, i, \texttt{next})$	$y{\rightarrow}\texttt{next}^i \sim \texttt{null}$		
$\texttt{List}(\texttt{result}, j, \texttt{next})$	$\texttt{List}(\texttt{result}, j, \texttt{next})$	$\texttt{List}(\texttt{result}, j, \texttt{next})$	$\texttt{List}(\texttt{result}, j, \texttt{next})$	$\texttt{List}(\texttt{result}, j, \texttt{next})$
$y + 4 = x$	$y + 4 = x$	$y + 4 = x$	$y + 4 = x$	$y + 4 = x$
$*x = \text{tmp}$	$*x = \text{tmp}$	$*x \sim \text{tmp}$		$*x = \texttt{result}$
	$\texttt{List}(\text{tmp}, i\text{-}1, \texttt{next})$	$\texttt{List}(\text{tmp}, i\text{-}1, \texttt{next})$	$\texttt{List}(\text{tmp}, i\text{-}1, \texttt{next})$	$\texttt{List}(\text{tmp}, i\text{-}1, \texttt{next})$

Figure 8. An example from the standard in-place list reversal program illustrating postcondition computation across assignment $*x := \texttt{result}$ on facts at the beginning of the loop shown in the first column. The set of may equalities is equal to the must set initially.

To ensure termination across loops, we do the following: (a) Use a widening operation on the base constraint domain, (b) bound size of pointer expressions and the constants that occur in those pointer expressions by some fixed small constant. It is easy to see that these two heuristics guarantee termination across loops. It can also be verified that the computational complexity of each transfer function is at most polynomial in the number of program variables.

Fixed-point computation requires an implies check on elements of our abstract domain. The key idea is to match the existentially quantified variables on the right side with those on the left side. Then we check if every must-equality on the right side is also present on the left side, and if every may-equality on the left side is also present on the right side. For the former, we compute an under-approximation of must-aliases of e_1 from the must-equalities of F and then check whether e_2 belongs to that set. For that latter, we compute an over-approximation of may-aliases of e_1 from the may-equalities of F and then check whether e_2 does not belong to that set. Section 4.1 describes how to compute an under-approximation of must-aliases and an over-approximation of may-aliases.

5. Experiments

We have built a prototype implementation of the algorithms described in this paper. Our tool is implemented in C/C++ and takes two inputs: (i) some procedure in a low-level three-address code format (without any typing information) (ii) precondition for the inputs of that procedure expressed in the language of our abstract domain.

5.1 Choice of Base Constraint Domain

We chose the base constraint domain to be the conjunctive domain over *combination* of linear arithmetic and uninterpreted function terms.

For the linear arithmetic part, we used constraints of the form $x \leq ay + c$ (slight generalization of difference constraints). We observed that difference constraints were not sufficient to represent the invariants primarily because of the use of multiplication by constants to compute the offsets in array dereferences (for eg, the access $A[i].\texttt{data}$ gets translated into $A{\rightarrow}(\texttt{tmp})$ in the low-level code, where $\texttt{tmp} = 8i + \texttt{data}$ and we need to represent the latter invariant.) For `List2Array` and `Array2PtrArray` this choice turns out to be good enough.

There are three known ways of defining combination of abstract domains: direct product, reduced product [6], and logical product [13]. We chose the logical product combination, since the other two were not precise enough to represent the invariants required, for example, in the `ListPtrArray` or `ArrayPtrArray` examples. This choice is good enough to represent all base constraints that arise in our examples.

We implemented the algorithms for the logical combination of linear arithmetic (in particular, generalized difference constraints) and uninterpreted function terms based on the combination methodology described by Gulwani and Tiwari [13].

5.2 Examples

Our experimental results are encouraging. We were successfully able to run our tool on the example programs shown in the table in Figure 5. These examples have been chosen for the following reasons: (i) Each of these illustrates the interesting aspects of a different transfer function described in Section 4. (ii) Two of these examples `List2Array` and `ArrayPtrArray` also represent very common coding patterns. (iii) We do not know of any automatic tool that can verify the correctness of these programs automatically in low-level form, where pointer arithmetic is used to compute array offsets and even field dereferences.

List2Array This example flattens a list into an array by using two *congruent* loops - one to compute the length of the input list to determine the size of the array, and the second to copy each list elements in the allocated array. Figure 10 describes this example and the useful invariants generated by our tool.

This example reflects a common coding practice in which memory safety relies on inter-dependence between different loop iterations. In this example, it is crucial to (represent and) compute the invariant that ℓ stores the length of the input list. Generation of this invariant depends on the ability of our meet algorithm to reason about the interaction between must-equalities and a disequality predicate, as well as disequalities (represented by absence of may-equalities) and an equality predicate. See Figure 6(a) for details.

ListReverse The procedure performs an in-place list reversal. The interesting loop invariant that arises in this example is that the sum of the lengths of the list pointed to by the iterator current y (i.e., the part of the list that is yet to be reversed) and the list pointed to by the current result `result` (i.e., the part of the list that has been reversed) is equal to the length of the original input list (which in our example is 100).

$$\exists i_1, i_2 : i_1 + i_2 = 100 \quad \wedge \quad \texttt{List}(y, i_1, \texttt{next}) \quad \wedge \quad \texttt{List}(\texttt{result}, i_2, \texttt{next})$$

Note that since the input list is being updated in-place, we cannot refer to the length of the original input list except if its length was a constant or if it is stored in some live program variable. The discovery of the above invariant relies on the ability of our postcondition operator to do strong updates (i.e., remove may-aliase facts), and generate relevant equalities by transitive reasoning before destroying any must-equality. See Figure 8 for details.

ArrayPtrArray This example is similar to the one described in Figure 1 in which the input list x is a list of Arrays (and their length fields). The only difference is that (for our experiments) we instead chose the data structure for our input A to be an array of arrays (and their length fields) since the respective invariants required for verifying the memory safety offer a more involved reasoning of pointer arithmetic. The loop invariant that we discover for the first loop (which initializes $A[j].\texttt{data}$ to some array of length $A[j].\texttt{data}$) is the following variant of the one described on Page 2 in Section 1. Here ℓ denotes the length of the array A and is

Program	Base Constraint Domain Required	Property Discovered (apart from memory safety)	Precondition Provided
`List2Array`	Generalized Difference Constraints	The corresponding array and list elements are same	Input is a list
`ListReverse`	Generalized Difference Constraints	Reversed list has length 100	Input is a list of size 100
`ArrayPtrArray`	Generalized Difference Constraints + Uninterpreted Functions		Input array has length `Len` (where `Len` is also an input)

Figure 9. Examples on which we performed our experiments. Column 2 lists the base constraint domain that is required to reason about memory safety as well as the property listed in column 3 of the program in column 1. Our prototype implementation took less than 0.5 seconds for automatic generation of invariants on these examples. We also ran our tool in a verification setting in which we provided the loop invariants and the tool took less than 0.1 seconds to verify the invariants.

```
  List2Array(struct {int Data,*Next;}* x)
1   ℓ := 0;
2   for (y := x; y ≠ Null; y := y→next)
3         ℓ := ℓ + 1;
4   A := malloc(4ℓ); y := x;
5   for (k := 0; k < ℓ; k := k + 1)
6         A→(4k) := y→data; y := y→next;
7   return A
```

π	Invariant at π
1	$\exists i : \texttt{List}(x, i, \texttt{next})$
2	$\exists i : \ell = 0, \texttt{List}(x, i, \texttt{next})$
3	$\exists i : 0 \leq \ell < i, \texttt{List}(x, i, \texttt{next}), y = x{\rightarrow}\texttt{next}^{\ell}$
4	$\texttt{List}(x, \ell, \texttt{next})$
5	$\texttt{List}(x, \ell, \texttt{next}), \texttt{Array}(A, 4\ell)$
6	$\texttt{List}(x, \ell, \texttt{next}), \texttt{Array}(A, 4\ell), 0 \leq k < \ell, y = x{\rightarrow}\texttt{next}^{k}$ $\forall j((0 \leq j < k) \Rightarrow x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{data} = A{\rightarrow}(4j + \texttt{data})$
7	$\texttt{List}(x, \ell, \texttt{next}), \texttt{Array}(A, 4\ell), y = \texttt{null}$ $\forall j((0 \leq j < \ell) \Rightarrow x{\rightarrow}\texttt{next}^{j}{\rightarrow}\texttt{data} = A{\rightarrow}(4j + \texttt{data}))$

Figure 10. An example program that flattens a list of integers into an array of integers. We assume that the structure fields Data and Next are at offsets `data` $= 0$ and `next` $= 4$ respectively. The table on the right lists *selected* invariants at the corresponding prog ram points that were discovered by our implementation. The `List` and `Array` predicates are as defined on Page 2.

also an input to the procedure, and k is the loop iterator.

$$0 \leq k < \ell \ \wedge \ \texttt{Array}(A, 8\ell) \ \wedge$$
$$\forall j : (0 \leq j < k) \Rightarrow \texttt{Array}(A{\rightarrow}(8j + \texttt{data}), 4 \times (A{\rightarrow}(8j + \texttt{len})))$$

The discovery of the above quantified invariant relies crucially on the ability of our join algorithm to generalize its inputs into quantified invariants. See Figure 5(b) for details.

6. Related Work

Alias/Pointer analysis Early work on alias analysis used two main kinds of approximations to deal with recursive data-structures: *summary nodes* that group together several concrete nodes based on some criteria such as same allocation site (e.g., [2]), or *k-limiting* which does not distinguish between locations obtained after k dereferences (e.g., [18]), or a combination of the two (e.g., [4]). Deutsch proposed reducing the imprecision that arises as a result of k-limiting by using suitable representations to describe pointer expressions (and hence alias pairs) with potentially unbounded number of field dereferences [7]. The basic idea was to use new variables to represent the number of field dereferences and then describe arithmetic constraints on those variables. However, such techniques were not deemed sufficient to express rich pointer properties.

Most of the new techniques that followed focused on defining logics with different kinds of predicates (other than simple must-equality and may-equality predicates, which were used by earlier techniques) to keep track of shape of heap-structures [15, 23, 22, 17]. There is a lot of recent activity on building abstract interpreters using these specialized logics [8, 20, 12]. In this general approach, the identification of the "right" abstract predicates and automation of the analysis are challenging tasks. In some cases, the analysis developer has to provide the transfer functions for each of these predicates across different flowchart nodes.

Additionally, the focus of the above mentioned techniques has been on recursive data structures, and they do not provide good support for handling arrays and pointer arithmetic. Recently though, there has been some work in this area. Gopan, Reps, and Sagiv have suggested using canonical abstraction [23] to create a finite partition of (potentially unbounded number of) array elements and using summarizing numeric domains to keep track of the values and indices of array elements [11]. However, the description of their technique has been limited to reasoning about arrays of integers. Calcagno et al. have used separation logic to reason about memory safety in presence of pointer arithmetic, albeit with use of a special predicate tailored for a specific kind of data-structure (multi-word lists) [1]. Chatterjee et al. have given a formalization of the reachability predicate in presence of pointer arithmetic in first-order logic for use in a modular verification environment where the programmer provides the loop invariants [3].

The work presented in this paper tries to address some of the above-mentioned limitations. Our use of quantification over two simple (must and may-equality) predicates offers the benefits of richer specification as well as the possibility of automated deduction. Additionally, our abstract domain has good support for pointer arithmetic in presence of recursive data structures.

Quantified invariants There has been some earlier work on automatically discovering quantified invariants [16, 10]. However these approaches require that the predicates that occur in the quantified invariants be provided explicitly, either by the programmer or by an automatic abstraction-refinement technique. Moreover, these approaches have specifically focussed on discovering universally quantified invariants over arrays. On the other hand, our approach does not require these predicates to be provided - relevant predicates are automatically discovered by the base constraint domain. Also, we discover invariants of the form $\exists\forall$ (most of the examples presented in this paper cannot be verified with simply $\forall$ quantification), and we have first-class support for reasoning about pointer arithmetic.

Data-structure Specifications McPeak and Necula have suggested specifying and verifying properties of data-structures using local equality axioms [21]. For example, the invariant associated with the program `List2Array` (after execution of the first loop) in Figure 10 might be specified at the data-structure level as saying that the field `Len` is the length of the array field `Data`. Similar ap-

proaches have been suggested to specify and verify properties of object-oriented programs [19], or locking annotations associated with fields of concurrent objects [9].

These approaches might result in simpler specifications that avoid universal quantification (which has been made implicit), but they also have some disadvantages: (a) They require source code with data-structure declarations, while our approach also works on low-level code without any data-structure declarations. (b) Sometimes it may not be feasible to provide specifications at the data-structure level since the related fields may not be local (i.e., not present in the same data-structure). (c) Programmers need to write down the intended specifications for the data-structures which can be a daunting task for large legacy code-bases, (d) It is not clear what such a specification would mean when these fields are set only after some computation has been performed. Perhaps something like pack/unpack of Boogie methodology [19] or the temporary invariant breakage approach suggested in [21] may be used to give it a well-defined semantics, but in such a setting establishing validity of these specifications may itself require an inductive reasoning (like the one required in our `List2Array` example for establishing the invariant after the first loop).

7. Conclusion and Future Work

This paper describes an abstract domain that gives first-class treatment to pointer arithmetic and recursive data-structures. The proposed abstract domain can be used to represent useful quantified invariants for several common code patterns (even in low-level software). These invariants can be automatically discovered by performing an abstract interpretation of programs over this abstract domain. The transfer functions required for this purpose are built using the transfer functions for the base constraint domain.

We feel that the techniques described in this paper can be used for automatically discovering quantified invariants in real software– without using any support in the form of user-specified list of predicates. We are currently in the process of performing more experiments with our tool. Future work includes extending these techniques to an interprocedural analysis, wherein preconditions of most procedures are automatically discovered in a whole-program setting.

References

[1] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.

[2] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.

[3] S. Chatterjee, S. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyszing low-level software. Technical Report MSR-TR-2006-154, Microsoft Research, 2006.

[4] J.-D. Choi, M. G. Burke, and P. R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, 1993.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 234–252, 1977.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Symp. on POPL*, pages 269–282, 1979.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k -limiting. In *PLDI*, pages 230–241, 1994.

[8] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.

[9] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI*, pages 219–232, 2000.

[10] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[11] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.

[12] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.

[13] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, June 2006.

[14] T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *3rd IEEE Intl. Conf. SEFM'05*, 2005.

[15] J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI*, pages 226–236, 1997.

[16] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.

[17] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.

[18] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *PLDI*, June 1992.

[19] K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP*, pages 115–130, 2006.

[20] S. Magill, A. Nanevsky, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *SPACE*, 2006.

[21] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.

[22] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[23] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[24] T. Touili. Regular model checking using widening techniques. *Electr. Notes Theor. Comput. Sci.*, 50(4), 2001.