An Abstract Domain for Analyzing Heap-Manipulating Low-Level Software*

Sumit Gulwani[1] and Ashish Tiwari[2]

[1] Microsoft Research, Redmond, WA 98052, sumitg@microsoft.com
[2] SRI International, Menlo Park, CA 94025, tiwari@csl.sri.com

Abstract. We describe an abstract domain for representing useful invariants of heap-manipulating programs (in presence of recursive data structures and pointer arithmetic) written in languages like C or low-level code. This abstract domain allows representation of must and may equalities among pointer expressions. Pointer expressions contain existentially or universally quantified integer variables guarded by some base domain constraint. We allow quantification of a special form, namely $\exists\forall$ quantification, to balance expressiveness with efficient automated deduction. The existential quantification is over some *dummy* non-program variables, 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. Our abstract interpreter automatically computes invariants about programs over this abstract domain. We present initial experimental results demonstrating the effectiveness of this abstract domain on some common coding patterns.

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 the number of expressions, or aliasing relationships, becomes potentially infinite. The presence of pointer arithmetic makes this even harder.

We describe an abstract domain that can represent precise must and may-equalities among pointer expressions that are needed to prove correctness of several common code patterns in low-level software. It is motivated by the early work on representing aliasing directly using must-alias and may-alias pairs of pointer expressions [2, 15, 4, 5]. 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 dereferencing at arbitrary integer (expression) offsets. This gives our abstract domain the ability to handle arrays, pointer arithmetic,

* The second author was supported in part by the National Science Foundation under grant CCR-0326540.

```
struct List {int Len, *Data; List* Next;}
ListOfPtrArray(struct List* $x$)
  for ($y := x$; $y \neq$ null; $y := y\to$next)
      $t :=?$; $y\to$len $:= t$; $y\to$data $:=$ malloc($4t$);
  for ($y := x$; $y \neq$ null; $y := y\to$next)
      for ($z := 0$; $z < y\to$len; $z := z + 1$) $y\to$data$\to(4z) := ...$;
```

Fig. 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.

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 nontrivial properties of data-structures in programs. [3]

We allow only a special form of quantification in our abstract domain, namely ∃∀ quantification, to balance expressiveness with potential for automated deduction. The quantification is over integer variables that are *not* program variables. The existentially quantified variables can be seen as *dummy* program variables that are explicitly needed to express common program invariants. The universally quantified variables describe properties of (potentially unbounded) collections of memory locations.

Our abstract domain uses only two base predicates, must and may equality, unlike the common approach of using a pre-defined set of richer predicates [13, 20, 19, 14]. As a result, reasoning in our abstract domain does not require any special deduction rules, thereby yielding comparative simplicity and easier automation.

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 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 `Data`. This *quantified* invariant is expressed in our abstract domain as

$$\exists i.\texttt{List}(x, i, \texttt{next}) \wedge \forall j[0 \leq j < i \Rightarrow \texttt{Array}(x{\to}\texttt{next}^j{\to}\texttt{data}, 4 \times x{\to}\texttt{next}^j{\to}\texttt{len})] \quad (1)$$

where $x{\to}\texttt{next}^j$ is an (pointer) expression in our language that denotes the memory location obtained by performing j dereferences at offset `next` starting from x. The predicates `List` and `Array` are abbreviations for the following definitions.

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

[3] A limited form of quantification over integer variables was implicitly hidden in the set representation used for representing may-aliases in the work by Deutsch [5].

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)[4], and can be encoded as the following must-equality:

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

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.We automatically generate invariants, like the one described in Equation 1, by performing abstract interpretation (whose transfer functions are described in a full version of this paper [11]) over our abstract domain.

This paper is organized as follows. Section 2 describes our program model, which closely reflects the memory model of C modulo some simple assumptions. We then formally describe our abstract domain and present its semantics in relation to our program model (Section 3). We then describe the procedure to check implication in this abstract domain (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 a pair of a region identifier and a positive offset.

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

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 (which denotes size of that 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). The above expressions have the usual expected semantics with the usual restrictions that it is not proper to add or subtract two

[4] 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.

$$\frac{}{\llbracket\texttt{null}\rrbracket\rho = \texttt{null}}\ \textsf{null} \qquad \frac{\llbracket e_1\rrbracket\rho = \langle r,d\rangle \quad \llbracket e_2\rrbracket\rho = c \quad r \in D \quad 0 \le d+c < R(r)}{\llbracket e_1 {\rightarrow} e_2\rrbracket\rho = P(\langle r, d+c\rangle)}\ \textsf{deref}$$

$$\frac{\llbracket e_1\rrbracket\rho \text{ and } \llbracket e_2\rrbracket\rho \text{ are ints}}{\llbracket e_1 \pm e_2\rrbracket\rho \ = \ \llbracket e_1\rrbracket\rho + \llbracket e_2\rrbracket\rho}\ \textsf{intArith} \qquad \frac{\llbracket x_1\rrbracket\rho \text{ and } \llbracket x_2\rrbracket\rho \text{ are ints}}{\llbracket x_1 \operatorname{rel} x_2\rrbracket\rho \ = \ \llbracket x_1\rrbracket\rho \ \operatorname{rel} \ \llbracket x_2\rrbracket\rho}\ \textsf{IntCompare}$$

$$\frac{}{\llbracket c\rrbracket\rho = c}\ \textsf{cons} \qquad \frac{\operatorname{rel} \in \{\neq, =\} \quad \llbracket x_1\rrbracket\rho,\ \llbracket x_2\rrbracket\rho \text{ are } \texttt{null} \text{ or valid pointers}}{\llbracket x_1 \operatorname{rel} x_2\rrbracket\rho \ = \ \llbracket x_1\rrbracket\rho \ \operatorname{rel} \ \llbracket x_2\rrbracket\rho}\ \textsf{ptrCompare}$$

$$\frac{\llbracket e_1\rrbracket\rho = \langle r,d\rangle \quad \llbracket e_2\rrbracket\rho = \text{int } c}{\llbracket e_1 \pm e_2\rrbracket\rho = \langle r, d \pm c\rangle}\ \textsf{ptrArith} \qquad \frac{V(y) = \langle r,i\rangle \quad r \in D \quad 0 \le i < R(r)}{\llbracket \text{free}(y)\rrbracket\rho = \langle D - \{r\}, R, V, P\rangle}\ \textsf{Free}$$

$$\frac{\text{Let } c \text{ be a non-det int}}{\llbracket ?\rrbracket\rho = c}\ \textsf{nonDet} \qquad \frac{}{\llbracket x := e\rrbracket\rho = \langle D, R, V[x \mapsto \llbracket e\rrbracket\rho],\ P\rangle}\ \textsf{varUpdate}$$

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

$$\frac{\llbracket e\rrbracket\rho \ge 0 \quad \text{Let } r \text{ be some fresh region identifier}}{\llbracket x := \text{malloc}(e)\rrbracket\rho = \langle D \cup \{r\},\ R[r \mapsto \llbracket e\rrbracket\rho],\ V[x \mapsto \langle r,0\rangle],\ P\rangle}\ \textsf{Malloc}$$

Fig. 2. Semantics of Expressions, Predicates, and Statements in our language. ρ denotes the state $\langle D, R, V, P\rangle$. In a program state, an expression is evaluated to a value, a predicate is evaluated to a boolean value, and a statement is evaluated to a program state. Evaluation of an expression, or statement in a state s.t. none of the above rules apply yields a $\perp$ value or $\perp$ state respectively.

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 (e.g., those that involve bitwise arithmetic). Given a program state ρ, an expression e evaluates to some value, denoted by $\llbracket e\rrbracket\rho$, according to the formal semantics given in Figure 2.

Statements The assignment statements, $x := e$ and $*x := e$, have standard semantics. The memory allocation assignment, $x := malloc(e)$, assigns a pointer value with a fresh region identifier to x. The statement $free(e)$ frees the region pointed to by e. The formal semantics of these statements is given in Figure 2.

Predicates The predicates that occur in conditionals are of the form $x_1 \operatorname{rel} x_2$, where $\operatorname{rel} \in \{<, \le, \neq, =\}$. 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 inequality too; see Figure 2.

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 ρ' that have the following properties respectively: (a) $\rho' \neq \perp$, (b) if $\rho' = \langle D, R, V, P \rangle$, then for all region identifiers $r \in D$, there exists an expression e s.t. $[\![e]\!]\rho' = \langle r, d \rangle$.

Intuitively, a procedure is memory-safe if all memory dereferences and free operations are performed on valid pointer values. 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 reflects 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 (some implementations of C insist that x point to the beginning of a region returned by malloc). Our program model can be easily adapted to capture other possible semantics of C while not depending on the above assumptions. The current choice has been made to simplify presentation. 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 as follows:

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

`valid` is a special constant in our domain that satisfies $\texttt{valid} \neq \texttt{null}$. 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 3 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 following semantics (If e_3 is 1, we write $e_1 {\rightarrow} e_2^{e_3}$ as $e_1 {\rightarrow} e_2$).

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

Must-equality is a binary predicate over *pointer* expressions denoted using "=" and is used in an infix notation. This predicate 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 *pointer* expressions. It is denoted using "$\sim$" and is used in an infix notation. This predicate describes 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 disequalities 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. The constraints C and $C_{\mathtt{f}}$ are arithmetic constraints on expressions in a *base constraint domain* that is a parameter to our algorithm.

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

The existentially quantified variables, U, can be seen as *dummy* program variables that are needed 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 substitution σ for variables in U such that the following holds: If $\rho_{\mathtt{e}} = \langle D, R, V\sigma, P\rangle$, (where $V\sigma$ denotes the result of mapping v to $\sigma(v)$, for all v, in V) then,

- $\rho_{\mathtt{e}} \models C$, i.e., for each predicate $e_1 \,\mathrm{rel}\, e_2 \in C$, $[\![e_1 \,\mathrm{rel}\, e_2]\!]\rho_{\mathtt{e}}$ evaluates to $\mathtt{true}$.
- $\rho_{\mathtt{e}} \models M$, i.e., for all facts $(\forall U_{\mathtt{f}}(C_{\mathtt{f}} \Rightarrow (e_1 = e_2))) \in M$, for every 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 $\rho_{\mathtt{f}} = \langle D, R, V\sigma\sigma_f, P\rangle$. In the special case when $e_1 = e_2$ is of the form $e \to \beta = \mathtt{valid}$, then $[\![e]\!]\rho_{\mathtt{f}} = \langle r, c\rangle$, $r \in D$, and $0 \le c + [\![\beta]\!]\rho_{\mathtt{f}} < R(r)$.
- For all expressions e_1 and e_2, if there is a state ρ' s.t. $\rho' \models C$, $\rho' \models Y$ (treating may-equality as must-equality and using the above definition of $\models$), $[\![e_1]\!]\rho' \neq \bot$, $[\![e_2]\!]\rho' \neq \bot$, and $[\![e_1]\!]\rho' \neq [\![e_2]\!]\rho'$, then $[\![e_1]\!]\rho_{\mathtt{e}} \neq [\![e_2]\!]\rho_{\mathtt{e}}$. Informally, if $e_1 \sim e_2$ is not implied by Y, then $[\![e_1]\!]\rho_{\mathtt{e}} \neq [\![e_2]\!]\rho_{\mathtt{e}}$.

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

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

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 any variable x may be aliased to anything reachable from any variable 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. Instead the validity of an expression needs to be explicitly stated using `Valid` predicates (defined on Page 3 in Section 1).

Observe that there cannot be any program state that satisfies a formula whose must-equalities are not a subset of (implied by the) may-equalities. Hence, any useful formula will have every must-equality also as a may-equality. Therefore, we assume that in our formulas all must-equalities are also may-equalities, and avoid duplicating them in our examples.

3.2 Expressiveness

In this section, we discuss examples of program properties that our abstract elements can express.
(a) x points to an (possibly `null`) acyclic list: $\exists i : \texttt{List}(x, i, \texttt{next})$. The predicate `List` is as defined on Page 2.
(b) x points to a region (array) of t bytes: $\texttt{Array}(x, t)$. The predicate `Array` is as defined on Page 2.
(c) x points to a cyclic list: $\exists i : i \geq 1 \wedge x = x{\rightarrow}\texttt{next}^i \wedge \forall k(0 \leq k < i \Rightarrow \texttt{Valid}(x{\rightarrow}\texttt{next}^k))$
(d) Lists x and y share a common tail: $\exists i, j : i \geq 0 \wedge j \geq 0 \wedge x{\rightarrow}\texttt{next}^i = y{\rightarrow}\texttt{next}^j$
(e) y *may* point to some node in the list pointed to by x.

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

Observe that existential quantification and forall quantification over may-equalities has the same semantics.
(f) The (reachable) heap is completely disjoint, i.e., no two distinct reachable memory locations point to the same location: `true`. Observe that disjointedness comes for free in our representation, i.e., we do not need to say anything if we want to represent disjointedness.
(g) 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 described above is insufficient to represent this invariant precisely. However, a simple extension in which disjunctions of offsets (as opposed to a single offset) are allowed can represent this invariant precisely as follows: $\forall i \geq 0 : x{\rightarrow}(\texttt{left}||\texttt{right})^i \sim y$. The

semantics of the abstract domain can be easily extended to accomadate disjunctive offsets as above. A formal treatment of disjunctive offsets was avoided in this paper for the purpose of simplified presentation.

Regarding limitations of the abstract domain, we can not express arbitrary disjunctive facts and invariants that requires $\forall\exists$ quantification (such as the invariants required to analyze the Schorr-Waite algorithm [12]). We plan to enrich our abstract domain in the future.

4 Automated Deduction over the Abstract Domain

In this section, we briefly describe the key ideas behind our sound procedure for checking implication in our abstract domain.[5] For lack of space, the remaining transfer functions (namely, `Join`, `Meet`, `Widen`, and `Strongest Postcondition` operations) needed for performing abstract interpretation over our abstract domain are described in a full version of this paper [11].

The first step in deciding if F implies F', where F, F' are abstract elements, is to instantiate the existentially quantified variables in F' in terms of existentially quantified variables in F. We do this by means of a heuristic that we have found to be effective for our purpose. After this step, we can treat the existential variables as constants. Now consider the simpler problem of checking whether F implies $e_1 = e_2$ or whether F implies $e_1 \neq e_2$. 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.

The function `MustAliases`(e, F) returns an under-approximation A of all must-aliases of expression e such that for every $e' \in A$, we can deduce that $F \Rightarrow e = e'$. Similarly, the function `MayAliases`(e, F) returns an over-approximation A of all may-aliases of expression e such that if $F \Rightarrow e \sim e'$, then $e' \in A$. Since these alias sets may have an infinite number of expressions, we represent the alias sets of an expression e using a finite set of pairs (C, e'), where (C, e') denotes all expressions e' that satisfy the constraint C. [6]

The pseudo-code for `MustAliases` and `MayAliases` is described in Figure 3. The key idea in our algorithm for `MustAliases` is to do a bounded number of transitive inferences on the existing must-equalities. The key idea in `MayAliases`

[5] We have not investigated decidability of the entailment relation in our abstract domain. Results about $\exists\forall$ fragment of first-order logic are not directly applicable because of integer variables in our terms. In this work, the focus was on obtaining an abstract domain for building a sound abstract interpreter that can generate useful invariants. Theoretical issues, such as decidability, are left for future work.

[6] This representation is motivated by the one used by Deutsch [5] 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.

```
MustAliases$(e, F)$
  $A := \{\langle \texttt{true}, e\rangle\}$
  While change in $A$ and not tired
    Forall $(\forall V(C \Rightarrow e_1 = e_2)) \in F$ and
           $\langle C', e'\rangle \in A$
      If $((\sigma, \gamma) := \texttt{MatchExpr}(e', e_1) \neq \bot)$
        $A := A \cup \{\langle C' \wedge C\sigma, (e_2\sigma)\rightarrow\gamma\rangle\}$
  return $A$

MayAliases$(e, F)$
  $A := \{\langle \texttt{true}, e\rangle\}$
  While change in $A$
    Forall $(\forall V(C \Rightarrow e_1 \sim e_2)) \in F$ and
           $\langle C', e'\rangle \in A$
      If $((\sigma, \gamma) := \texttt{MatchExpr}(e', e_1) \neq \bot)$
        $A := A \cup \{\langle C' \wedge C\sigma, (e_2\sigma)\rightarrow\gamma\rangle\}$
        $A := \texttt{OverApprox}(A)$
  return $A$
```

(a) Algorithm

Inputs:

$e = x$

$F_1 = \{x = x\rightarrow\texttt{n}^j\}$

$F_2 = \{\forall i((0 \le i < j) \Rightarrow x\rightarrow\texttt{n}^i = x\rightarrow\texttt{n}^{i+1}\rightarrow\texttt{p})\}$

Outputs:

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

$\texttt{MustAliases}(e, F_2) = \{x\rightarrow\texttt{n}\rightarrow\texttt{p}, x\rightarrow\texttt{n}\rightarrow\texttt{p}\rightarrow\texttt{n}\rightarrow\texttt{p}\}$

$\texttt{MayAliases}(e, F_1) = \{x\rightarrow\texttt{n}^t \mid t \ge j\}$

$\texttt{MayAliases}(e, F_2) = \{x\rightarrow(\texttt{n}||\texttt{p})^t \mid 0 \le t\}$ or
$\{x\rightarrow(t_1)^{t_2} \mid 0 \le t_2 \wedge \ell \le t_1 \le u\}$
where $\ell = \min(\texttt{n}, \texttt{p}), u = \max(\texttt{n}, \texttt{p})$

(b) Examples

Fig. 3. The functions MustAliases and MayAliases. In (b), the first choice for $\texttt{MayAliases}(e, F_2)$ is better than the second choice (if the n and p fields are not laid out successively), but will be generated only if we allow disjunctive offsets, as addressed in Section 3.2. Even though MayAliases is a conservative overapproximation it helps us prove that x does not alias with, for example, $x\rightarrow\texttt{data}$.

is to do transitive inferences on may-equalities until fixed-point is reached. A function, OverApprox, for over-approximating the elements in the set is used to guarantee termination in a bounded number of steps. (Similar widening techniques have been used for over-approximating regular languages [21].) Due to the presence of universal variables, the application of transitive inference requires matching and substitution, as in the theory of rewriting. The function $\texttt{MatchExpr}(e', e_1)$ returns either $\bot$ or a substitution σ (for the universally quantified variables in e_1) and a subterm γ s.t. e' and $e_1\sigma\rightarrow\gamma$ are syntactically equal.

Observe that the above algorithm for MustAliases lacks the capability for inductive reasoning. For example, even if the transitive inference goes on forever, it cannot deduce, for example, that $x\rightarrow\texttt{n}^i\rightarrow\texttt{p}^i$ is a must-alias of x, for any i, given F_2 of Figure 3. However, such inferences are not usually required.

5 Experiments

We have implemented a tool that performs an abstract interpretation of programs over the abstract domain described in this paper. Our tool is implemented in 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.

Program	Property Discovered (apart from memory safety)	Precondition Used
`ListOfPtrArray`		Input is a list
`ListReverse`	Reversed list has length n	Input is list of size n
`List2Array`	Corresponding array and list elmts are same	Input is a list

Fig. 4. Examples on which we performed our experiments. 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.

Our experimental results are encouraging. We chose the base constraint domain to be the conjunctive domain over *combination* of linear arithmetic and uninterpreted function terms [10]. We were successfully able to run our tool on the example programs shown in the table in Figure 4. These examples have been chosen for the following reasons: (i) These examples represent very common coding patterns. (ii) We do not know of any automatic tool that can verify memory safety of these programs automatically in low-level form, where pointer arithmetic is used to compute array offsets and even field dereferences.

ListOfPtrArray This is the same example as described in Figure 1. Our tool generates the following non-trivial loop invariant required to establish the property in Equation 1, which is required to prove memory safety in the second loop.

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

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 $F^{\texttt{l}}$ and $F^{\texttt{r}}$ (among other must-equalities) before the loop header and after one loop iteration respectively.

$$F^{\texttt{l}} \;=\; (y = x\rightarrow\texttt{next} \wedge S(0)) \qquad F^{\texttt{r}} \;=\; (y = x\rightarrow\texttt{next}^2 \wedge S(0) \wedge S(1))$$

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 [11] to generate quantified facts from quantifier-free inputs.

ListReverse This 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 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 n of the original input list.

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

```
  List2Array($x$)
  struct {int Data, *Next}*$x$;
1 $\ell := 0$;
2 for($y := x; y \neq \mathtt{null}; y := y \rightarrow \mathtt{n}$)
3    $\ell := \ell + 1$;
4 $A :=$ malloc($4\ell$); $y := x$;
5 for($k := 0;\ k < \ell;\ k := k + 1$)
6    $A \rightarrow (4k) := y \rightarrow \mathtt{d}$; $y := y \rightarrow \mathtt{n}$;
7 return $A$
```

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

Fig. 5. List2Array example. We assume that the structure fields Data and Next are at offsets $\mathtt{d} = 0$ and $\mathtt{n} = 4$ respectively. The table on the right lists *selected* invariants at the corresponding program points that were discovered by our implementation. The List and Array predicates are as defined on Page 2.

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 5 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 compute the invariant that ℓ stores the length of the input list.

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., [15]), or a combination of the two (e.g., [4]). However, such techniques had limited expressiveness and precision. 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 [5]. The basic idea was to use new variables to represent the number of field dereferences and then describe arithmetic constraints on those variables. Deutsch analysis did not have any *must* information.

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 [13, 20, 19, 14]. There is a lot of recent activity on building abstract interpreters using these specialized logics [6, 17, 9]. 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 [20] 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 [8]. 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.

Data-structure Specifications McPeak and Necula have suggested specifying and verifying properties of data-structures using local equality axioms [18]. For example, the invariant associated with the program `List2Array` (after execution of the first loop) in Figure 5 might be specified at the data-structure level as saying that the field `Len` is the length of the array field `Data`. Similar approaches have been suggested to specify and verify properties of object-oriented programs [16], or locking annotations associated with fields of concurrent objects [7].

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 have to provide 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 [16] or the temporary invariant breakage approach [18] may be used for a well-defined semantics, but this requires additional annotations for updates to additional (non-program) variables.

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. These quantified invariants can be automatically discovered by performing an abstract interpretation of programs over this domain - without using any support in the form of user-specified list of predicates. Future work includes performing more experiments and extending these techniques to an interprocedural analysis.

References

1. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, 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. In *TACAS*, 2007.
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*, 1993.
5. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k -limiting. In *PLDI*, pages 230–241, 1994.
6. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
7. C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI*, pages 219–232, 2000.
8. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
9. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
10. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, June 2006.
11. S. Gulwani and A. Tiwari. Static analysis for heap-maniuplating low level software. Technical Report MSR-TR-2006-160, Microsoft Research, Nov. 2006.
12. 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.
13. J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI*, 1997.
14. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.
15. W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *PLDI*, June 1992.
16. K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP*, pages 115–130, 2006.
17. S. Magill, A. Nanevsky, E. Clarke, and P. Lee. Inferring invariants in separation logic for list-processing programs. In *SPACE*, 2006.
18. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.
19. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
20. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
21. T. Touili. Regular model checking using widening techniques. *Electr. Notes Theor. Comput. Sci.*, 50(4), 2001.