Beyond Reachability: Shape Abstraction in the Presence of Pointer Arithmetic

Cristiano Calcagno[1], Dino Distefano[2], Peter W. O'Hearn[2,3], and Hongseok Yang[4]

[1] Imperial College, London
[2] Queen Mary, University of London
[3] Microsoft Research, Cambridge
[4] Seoul National University

Abstract. Previous shape analysis algorithms use a memory model where the heap is composed of discrete nodes that can be accessed only via access paths built from variables and field names, an assumption that is violated by pointer arithmetic. In this paper we show how this assumption can be removed, and pointer arithmetic embraced, by using an analysis based on separation logic. We describe an abstract domain whose elements are certain separation logic formulae, and an abstraction mechanism that automatically transits between a low-level RAM view of memory and a higher, fictional, view that abstracts from the representation of nodes and multiword linked-lists as certain configurations of the RAM. A widening operator is used to accelerate the analysis. We report experimental results obtained from running our analysis on a number of classic algorithms for dynamic memory management.

1 Introduction

Shape analysis algorithms statically infer deep properties of the runtime heap, such as whether a variable points to a cyclic or acyclic linked list. Previous shape analyses (e.g., [7, 30, 31, 14, 21, 23, 24, 6, 1]) assume a high-level storage model based on records and field access rather than a RAM with arithmetic. This is a significant limitation. They can deliver reasonable results for the usage of pointers or references in high-level languages such as Java or ML, or for programs written in a low-level language that happen to satisfy assumptions not dissimilar to those required by conservative garbage collectors. But, for many important low-level programs they would deliver imprecise results[1].

The crux of the problem is that the assumption of memory as composed of discrete nodes with pointers to one another – essentially as a form of graph – is a fiction that is exposed as such by pointer arithmetic. It is difficult to use the notion of *reachability* to characterize how memory may be accessed, because a memory cell can be accessed by an arithmetic calculation; in a sense, *any* cell is reachable. And yet, most shape analyses rely strongly on reachability

[1] Correspondingly, even for high-level languages current analyses are limited in the structures they infer *within* an array.

K. Yi (Ed.): SAS 2006, LNCS 4134, pp. 182–203, 2006.

or, more to the point, what can be inferred from non-reachability (from chosen roots). Several analyses use explicit reachability predicates in their formulation of abstract states [31, 24, 1], where others use graph models [30, 14, 21].

This paper has two main contributions. First, we show that it is possible to define a shape analysis for programs that mutate linked data structures using pointer arithmetic. Our abstract domain uses separation logic [27] formulae to represent abstract states, following on from the work on Space Invader [15] and Smallfoot [3, 4]. We use separation logic because it deals smoothly with pointer arithmetic; crucially, it does not depend in any way on anything about reachability for its soundness. We focus on a particular kind of linked structure that uses arithmetic: linked lists with variable length entries – or more briefly, *multiword lists* [8]. Multiword lists allow for arithmetic operations that split and coalesce blocks, and are one of the kinds of data structure used in memory managers.

We provide experimental results on a series of programs for dynamic memory management, essentially following the development in [20], the classic reference on the subject. Our most involved example is the `malloc` from Section 8.7 of [19].

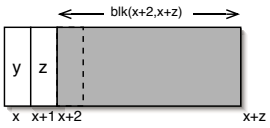
The second contribution concerns the use of logic in program analysis. We use two techniques to accelerate our analysis: a widening operator and a differential fixpoint algorithm. The way that they are used here makes it difficult to prove soundness by standard means (to do with prefixpoints). However, soundness is easy if we view the analysis algorithm as conducting a proof search in separation logic. The suggestion is that program logic provides a flexible way of exploring non-standard optimizations in program analysis, while maintaining soundness.

2 Basic Ideas

Although pointer arithmetic can potentially lead to an incredible mess, its disciplined use can be very regular. Programmers transit between a RAM-level or even bit-level view and a structured one where, say, a graph structure is laid on top of the sea of bits, even if the structured view is not enforced by the programming language. In this section we describe some of the basic ideas in our analysis in a semi-formal way, highlighting how it negotiates this kind of transit.

The formulae in our analysis take meaning in the standard RAM model of separation logic. The notation $[E]$ is used to dereference address E, where E is an arithmetic expression. We will not repeat the formal definition of the interpretation of formulae in this model, but instead will describe their meanings intuitively as we go along. Familiarity with [27] would be helpful.

We work with linked lists where a node x is a multiword structure storing a pointer y to the next node and an integer z which can be read out of the structure to determine the length of a block of memory associated with it.



In separation logic we can describe such nodes as follows. First, we consider a basic predicate $\mathsf{blk}(E, F)$, which denotes a (possibly empty) consecutive sequence of cells starting from E and ending in $F-1$. (This could be defined using the iterated separating conjunction [27], but we take blk as primitive.) We also use the usual points-to predicate $E{\mapsto}F$ which denotes a singleton heap where address E has contents F. Then, the predicate for multiword nodes has the definition

$$\mathsf{nd}(x, y, z) \stackrel{\text{def}}{=} (x{\mapsto}y) * (x{+}1{\mapsto}z) * \mathsf{blk}(x{+}2, x{+}z) \tag{1}$$

which corresponds directly to the picture above.

With the node predicate we can define the notion of a multiword linked-list segment from x to y

$$\mathsf{mls}(x, y) \stackrel{\text{def}}{=} (\exists z'.\, \mathsf{nd}(x, y, z')) \vee (\exists y', z'.\mathsf{nd}(x, y', z') * \mathsf{mls}(y', y)). \tag{2}$$

It is understood that this predicate is the least satisfying the recursive equation.

The blk, nd and mls predicates will form the basis for our abstract domain. In several of the memory managers that we have verified (see Section 7) the free list is a circular multiword linked list with header node `free`. Such a circular list, in the case that it is nonempty, can be represented with the formula $\mathsf{mls}(\texttt{free}, \texttt{free})$. In others the free list is acyclic, and we use $\mathsf{mls}(\texttt{free}, 0)$.

If one doesn't look inside the definition of nd, then there is no pointer arithmetic to be seen. The interesting part of the memory management algorithms, though, is how "node-ness" is broken and then re-established at various places.

Dynamic memory management algorithms often coalesce adjacent blocks when memory is freed. If we are given $\mathsf{nd}(x, y, z) * \mathsf{nd}(x{+}z, a, b)$ then an assignment statement $[x{+}1]{:=}[x{+}1]{+}[x{+}z{+}1]$ effects the coalescence. To reason about this assignment our analysis first breaks the two nodes apart into their constituents by unrolling the definition (1), giving us

$$\begin{array}{l}(x{\mapsto}y) * (x{+}1{\mapsto}z) * \mathsf{blk}(x{+}2, x{+}z)\\ * (x{+}z{\mapsto}a) * (x{+}z{+}1{\mapsto}b) * \mathsf{blk}(x{+}z{+}2, x{+}z{+}b).\end{array} \tag{3}$$

This exposes enough $\mapsto$ assertions for the basic forward-reasoning axioms of separation logic to apply to the formula. Even just thinking operationally, it should be clear that the assignment statement above applied to this state yields

$$\begin{array}{l}(x{\mapsto}y) * (x{+}1{\mapsto}z{+}b) * \mathsf{blk}(x{+}2, x{+}z)\\ * (x{+}z{\mapsto}a) * (x{+}z{+}1{\mapsto}b) * \mathsf{blk}(x{+}z{+}2, x{+}z{+}b)\end{array} \tag{4}$$

where the $x{+}1$ $*$-conjunct has been updated. At this point, we have lost node-ness of the portion of memory beginning at x and ending at $x{+}z$, because $z{+}b$ is stored at position $x{+}1$.

After transforming an input symbolic state we perform abstraction by applying selected sound implications. First, there is an implication from the rhs to the lhs of (1), which corresponds to rolling up the definition of nd. Applying this to formula (4), with a law of separation logic that lets us use implications within $*$-conjuncts (modus ponens plus identity plus $*$-introduction), results in

$$(x{\mapsto}y) * (x{+}1{\mapsto}z{+}b) * \mathsf{blk}(x{+}2, x{+}z) * \mathsf{nd}(x{+}z, a, b). \tag{5}$$

Next, there is a true implication

$$(x{\mapsto}y) * (x{+}1{\mapsto}z{+}b) * \mathsf{blk}(x{+}2, x{+}z) * \mathsf{nd}(x{+}z, a, b) \Longrightarrow \mathsf{nd}(x, y, z{+}b). \quad (6)$$

When we apply it to (5) we obtain the desired coalesced post-state $\mathsf{nd}(x, y, z{+}b)$. The implication (6) performs abstraction, in the sense that it loses the information that b is held at position $x{+}z{+}1$ and that $x{+}z$ has a pointer to a.

This discussion is intended to illustrate two features of our analysis method.

1. Before a heap access is made, predicate definitions are unrolled enough to reveal $\mapsto$ assertions for the cells being accessed.
2. After a statement is (symbolically) executed, sound implications are used to lose information as well as to establish that "higher-level" predicates hold.

These features were present already in the original Space Invader. The point here is that they give us a way to transit between the unstructured world where memory does not consist of discrete nodes and a higher-level view where memory has been correctly packaged together "as if" the node fiction were valid.

The real difficulty in defining the abstract domain is choosing the implications like (6) that lose enough information to allow fixpoint calculations to converge, without losing so much information that the results are unusably imprecise.

3 Programming Language and Abstract Domain

Programming Language. We consider a sequential programming language that allows arithmetic operations on pointers.

$$\begin{aligned}
[rclqTlql]e &::= n \mid x \mid e{+}e \mid e{-}e \\
B &::= e{=}e \mid e{\neq}e \mid e{\leq}e \\
S &::= x{:=}\,e \mid x{:=}\,[e] \mid [e]{:=}\,e \mid x{:=}\,\mathsf{sbrk}(e) \\
C &::= S \mid C\,;C \mid \texttt{if}(B)\ \{C\}\ \texttt{else}\ \{C\} \mid \texttt{while}(B)\,\{C\} \mid \texttt{local}\ x\,;C
\end{aligned}$$

We use the notation $[e]$ for the contents of the memory cell allocated at address e. Thus, $y{:=}\,[e]$ and $[e]{:=}\,e'$ represent look-up and mutation of the heap respectively. $\mathsf{sbrk}(e)$ corresponds to the UNIX system call which returns a pointer to e contiguous cells of memory. The other commands have standard meaning.

The programs in this language are interpreted in the usual RAM model of separation logic. Concrete states are defined by

$$\mathsf{States} \stackrel{\text{def}}{=} \mathsf{Stacks} \times \mathsf{Heaps} \qquad \mathsf{Stacks} \stackrel{\text{def}}{=} \mathsf{Vars} \to \mathsf{Ints} \qquad \mathsf{Heaps} \stackrel{\text{def}}{=} \mathsf{Nats}^{+} \rightharpoonup_{\text{fin}} \mathsf{Ints}$$

where Ints is the set of integers, Nats^{+} is the set of positive integers, and Vars is a finite set of variables. The concrete semantics of programs as state transformers can be defined in the standard way (see [15, 27]).

Table 1. Symbolic Heaps

$E, F ::= n \mid x \mid x' \mid E{+}E \mid E{-}E$	$H ::= E{\mapsto}E \mid \mathsf{blk}(E,E) \mid \mathsf{nd}(E,E,E)$
$P \quad ::= E{=}E \mid E{\neq}E \mid E{\leq}E \mid \mathsf{true}$	$\mid \mathsf{mls}(E,E) \mid \mathsf{true} \mid \mathsf{emp}$
$\Pi \quad ::= P \mid \Pi \wedge \Pi$	$\Sigma ::= H \mid \Sigma * \Sigma$
	$Q ::= \Pi \wedge \Sigma$

Symbolic Heaps. A symbolic heap Q is a separation-logic formula of a special form, consisting of a *pure part* Π and a *spatial part* Σ. Symbolic heaps are defined in Table 1. Due to pointer arithmetic, we use a richer collection of pure predicates than in [3]. As in [15] the primed variables are a syntactic convenience, which indicates that they are existentially quantified. Note that E is an integer expression, but unlike program expression e, it can contain primed variables. Spatial predicates blk, nd and mls have the meanings alluded to in Section 2.

The concretization $\gamma(Q)$ of Q is the set of the concrete states that satisfy $\exists\vec{y'}.\ Q$ according to the usual semantics of separation logic formulae, where $\vec{y'}$ consists of all the primed variables in Q. We will use the notations $Q * H$ and $P \wedge H$ to express $\Pi \wedge (\Sigma * H)$ and $(P \wedge \Pi) \wedge \Sigma$, respectively. We treat symbolic heaps as equivalent up to commutativity and associativity for $*$ and $\wedge$, identity laws $H * \mathsf{emp} = H$ and $P \wedge \mathsf{true} = P$, and idempotence law $\mathsf{true} * \mathsf{true} = \mathsf{true}$.

Let SH denote the set of all symbolic heaps Q. The abstract domain $\mathcal{D}$ consists of finite sets of symbolic heaps and an extra element $\top$:

$$\begin{array}{rcl} \mathcal{S} \in \mathcal{D} & \stackrel{\text{def}}{=} & \mathcal{P}_{\text{fin}}(\mathsf{SH}) \cup \{\top\} \\ \gamma(\mathcal{S}) & \stackrel{\text{def}}{=} & \textbf{if } (\mathcal{S} \neq \top) \textbf{ then } (\bigcup_{Q \in \mathcal{S}} \gamma(Q)) \textbf{ else } (\mathsf{States} \cup \{\text{fault}\}). \end{array}$$

Intuitively, $\mathcal{S}$ means the disjunction of all symbolic heaps in $\mathcal{S}$. The elements $\mathcal{S}, \mathcal{S}'$ of $\mathcal{D}$ are ordered by the subset relation extended with $\top$:

$$\mathcal{S} \sqsubseteq \mathcal{S}' \iff (\mathcal{S}' = \top \ \vee \ (\mathcal{S} \in \mathcal{P}(\mathsf{SH}) \ \wedge \ \mathcal{S}' \in \mathcal{P}(\mathsf{SH}) \ \wedge \ \mathcal{S} \subseteq \mathcal{S}')).$$

4 Abstraction Rules

The main part of our analysis is the abstraction function $\mathsf{Abs}\colon \mathcal{D} \to \mathcal{D}$, which establishes a fictional view of memory as consisting of nodes and multiword lists (forgetting information, if necessary, to do so). It is applied at the beginning of a loop and after each iteration; in the bodies of loops the fiction can be broken by operations on the RAM level.

The abstraction function has five steps, which successively: synthesize nodes from RAM configurations; simplify arithmetic expressions to control the potential explosion of arithmetic constraints; abstract size fields; reason about multiword lists; and filter out inconsistent symbolic heaps. We will specify the first four steps in terms of rewriting rules on SH. The rules in each step will always be

Table 2. Node Synthesis Rules

Package Rule	Swallow Rule
Precondition: $2 \leq G \leq H$	Precondition: $H+1 \leq G \leq H+K$
$Q * (E \mapsto F, G) * \mathsf{blk}(E+2, E+H)$	$Q * (E \mapsto F, G) * \mathsf{blk}(E+2, E+H) * \mathsf{nd}(E+H, I, K)$
$\Rightarrow Q * \mathsf{nd}(E, F, G) * \mathsf{blk}(E+G, E+H)$	$\Rightarrow Q * \mathsf{nd}(E, F, G) * \mathsf{blk}(E+G, E+H+K)$

Package2 Rule

Precondition: $2 \leq G \leq H$ with x' fresh

$Q * \mathsf{blk}(E, E+1) * (E+1 \mapsto G) * \mathsf{blk}(E+2, E+H) \Rightarrow Q * \mathsf{nd}(E, x', G) * \mathsf{blk}(E+G, E+H)$

normalizing. Thus, a particular strategy for applying the rules induces a function from SH to SH, which will then be lifted to a function on $\mathcal{D}$. For the last step, we will define a partial identity function on SH and lift it to a function on $\mathcal{D}$.

4.1 Node Synthesis

Node synthesis recognizes places where a portion of low-level memory can be packaged into a node. The synthesis rules are in Table 2. The idea of the first, Package, is just to package up a node using the definition of the nd predicate.[2] When we do this, we sometimes have to split off part of the end of a block in order to have the right information to form a node. Figure 1 gives a pictorial view of the Package rule. A node is indicated by a shaded box with a sub-part, G in the diagram, and an outgoing pointer, F there. The picture emphasizes the way in which the abstraction function transfers from the RAM-level view to the structured view where a group of cells becomes a unique entity (a node).

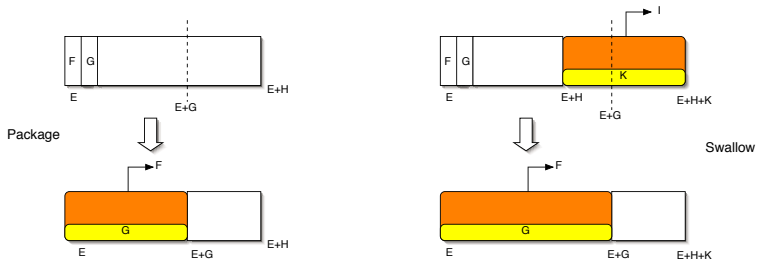


Fig. 1. Package Rule (left) and Swallow Rule (right)

The idea of the second rule, Swallow, is that when we already have a node to the right of a block as well as link and size cells, we might be able to swallow the preceding cells into a node. In doing this we again might have to chop off

[2] In these rules $E \mapsto F, G$ is the standard separation logic abbreviation for $E \mapsto F * E+1 \mapsto G$.

the end of the node (see Figure 1 for a depiction of the rule). The special case of this rule where $G = H{+}K$ corresponds to the discussion in Section 2.

The Package2 rule comes from a situation where a block has been split off to be returned to the user, and the size G of the node has been discovered by the allocation routine.

The technical meaning of these rules is that we can apply a rewriting $Q \Rightarrow Q'$ when Q implies the stated precondition. So, for the Package rule to fire we must establish an entailment

$$Q * (E{\mapsto}F, G) * \mathsf{blk}(E{+}2, E{+}H) \vdash 2{\leq}G{\leq}H.$$

Our analysis does this by calling a theorem prover for entailments $Q \vdash Q'$.

The theorem prover we have implemented builds on the prover used in Smallfoot [3]. It is incomplete, but fast, and it always terminates. The description of the analysis in this paper can be considered as parameterized by a sound prover. The prover is used in the abstraction phase, described in this section, as well as in the widening and rearrangement phases described in Sections 5 and 6.

Finally, there are inference rules which allow us to apply rewriting when the specific quantities in these rules do not match syntactically. For instance, given $E{+}H{=}x' \wedge ((E{\mapsto}F, H) * \mathsf{blk}(E{+}2, x'))$ we would like to apply the Package rule but we cannot do so literally, because we can only get the formula into the right form after substituting $E{+}H$ for x' (as mandated by the equality in the formula). For this, we apply the rule

$$\frac{Q[E] \;\Rightarrow\; Q' \qquad Q[F] \vdash E{=}F}{Q[F] \;\Rightarrow\; Q'} \quad \mathsf{Match1}$$

Here, $Q[\cdot]$ is a formula with a hole.

4.2 *n*-Simple Form

The analysis has a non-negative integer n as a parameter. It is used to limit offset arithmetic with a constant. (In our memory manager programs the choice $n = 4$ is sufficient).

When abstraction establishes the fictional view of the heap we must be careful to keep around some arithmetic information in the pure part, for example remembering that a found block packaged into a node was big enough to satisfy a malloc request. Keeping such important arithmetic information but dropping all the other information in the pure part is the purpose of the second abstraction step.

The second abstraction step transforms symbolic heaps to *n-simple form*, keeping information about only simple numerical relationships among variables and parameters of spatial predicates. The transformation prevents one source of divergence: the generation of increasingly complex arithmetic expressions. This abstraction reflects our intuitive understanding of programs for dynamic memory management: complex numerical relationships only express how heap cells form nodes, but they become unimportant once the nodes are synthesized.

Table 3. Rules for Transforming Symbolic Heaps to n-Simple Form

Substitution1 Rule	Substitution2 Rule
$x{=}E \wedge Q \Rightarrow x{=}E \wedge (Q[E/x])$	$x'{=}E \wedge Q \Rightarrow Q[E/x']$
(if $x{=}E$ is n-simple and $x{\in}\mathsf{fv}(Q)$)	(if $x'{=}E$ is n-simple and $x'{\in}\mathsf{fv}(Q)$)
Merge Rule	**Simplify Rule**
$E{\neq}0 \wedge 0{\leq}E \wedge Q \Rightarrow 0 \leq E{-}1 \wedge Q$	$Q[E/y'] \Rightarrow Q[x'/y']$
	(if E is not n-simple, $y'{\in}\mathsf{fv}(Q)$ and $x'{\notin}\mathsf{fv}(Q,E)$)
Drop Rule	
$P \wedge Q \Rightarrow Q$ (if atomic predicate P is not n-simple, or it contains some primed x')	

An expression is called *n-simple*, for $n \geq 0$, if it is either a primed variable or an instance of N in the following definition:

$$N, M ::= x_1{+}\cdots{+}x_k{-}y_1{-}\cdots{-}y_l{+}m$$

where all x_i, y_j are mutually disjoint nonprimed variables and m is an integer with $|m| \leq n$. For instance, neither $x{+}x{-}y$ nor $x{+}y{-}z{-}5$ is 3-simple, since x appears twice in the first, and $|{-}5| > 3$ in the second. An atomic pure predicate is *n-simple* if it is of the form $x{=}N$ or $0{\leq}N$ where N is n-simple and $x \notin \mathsf{fv}(N)$.

A symbolic heap $Q \equiv \Pi \wedge \Sigma$ is in *n-simple form* iff the following hold:

1. Q contains only n-simple expressions.
2. Π does not contain any primed variables.
3. $\Pi \equiv x_1{=}N_1 \wedge \ldots \wedge x_k{=}N_k \wedge 0{\leq}M_1 \wedge \ldots \wedge 0{\leq}M_l$ where all x_i's are distinct variables that occur in Q only in the left of equation $x_i{=}N_i$.

The third condition ensures that disequalities are dropped from Q, that the equalities define program variables $x_1, \ldots, x_k$ in terms of other program variables, and that these equalities have already been applied in Q. The transformation to n-simple form ensures that the analysis cannot diverge by repeatedly generating symbolic heaps with new pure parts. There are only finitely many pure parts of symbolic heaps in n-simple form, since the number of program variables is finite.

Table 3 shows the rewriting rules for transforming to n-simple form. The first two rules expand a primed or nonprimed variable into its definition. The third rule Merge encodes the $\neq$ relation using the $\leq$ relation. Note that none of these three rules loses information, unlike the last two. The Simplify rule loses some numerical relationships between parameters of spatial predicates in Q, and the Drop rule drops pure conjuncts P which are not in n-simple form. For instance $0{\leq}x{+}x \wedge \mathsf{nd}(x, x{+}x, x{+}x)$ gets transformed first to $0{\leq}x' \wedge \mathsf{nd}(x, x', x')$ by Simplify, then to $\mathsf{nd}(x, x', x')$ by Drop.

The Substitution and Merge rules require that an input symbolic heap should have a specific syntactic form. We have another matching rule to apply them more liberally:

$$\frac{P \wedge Q \;\Rightarrow\; Q' \qquad \vdash P \Leftrightarrow P'}{P' \wedge Q \;\Rightarrow\; Q'} \quad \mathsf{Match2}$$

where P is an atomic pure predicate, such as $E \neq E'$. Our implementation uses Match1 and Match2 in a demand-driven manner, building them into rules in Tables 2 and 3; we omit description of the demand-drivel variants for simplicity.

The reader might be wondering why we didn't use an existing abstract domain for numerical properties, such as [13], in the pure part. The short answer is that it is not obvious how to do so, for example, because of the way that symbolic heaps use existential quantification. An important direction for future work is to find ways to marry symbolic heaps with other abstractions, either directly or, say, through a suitable reduced product construction [10].

4.3 Abstraction at the Structured Level

Abstraction of the Size Field of Nodes. The third step of abstraction renames primed variables that are used to express the size fields of nodes, using the

Size Rule

$$Q * \mathsf{nd}(E, F, x') \;\Rightarrow\; Q * \mathsf{nd}(E, F, y') \quad (\text{if } x' \in \mathsf{fv}(Q, E, F) \text{ but } y' \notin \mathsf{fv}(Q, E, F))$$

This rule loses information about how the size x' of the node E is related to other values in Q. For instance, the rule abstracts $\mathsf{nd}(x, y, v') * \mathsf{nd}(y, 0, v')$ to $\mathsf{nd}(x, y, v') * \mathsf{nd}(y, 0, w')$, thereby losing that the nodes x and y are the same size.

After this step, every primed variable with multiple occurrences in a symbolic heap denotes the address, not the size, of a node. This implicit type information of primed variables is used in the remaining steps of the abstraction.

Multiword-List Abstraction. Next, the analysis applies abstraction rules for multiword-list segments. We use variants of the rewriting rules in [15], which are shown in Table 4[3].

The Append rule merges two smaller list segments, which are expressed by mls or nd. The side condition is for precision, not soundness. The first conjunct in the condition prevents abstraction when x' denotes a shared address: i.e. that two spatial predicates contain x' in their link fields. This case is excluded by the condition $x' \notin \mathsf{fv}(Q, G)$, for the second predicate witnessing the sharing could be in Q or it could be L_1. The second conjunct prevents abstraction when $L_0(E, x')$ is a node predicate which indirectly expresses relationships between variables. For instance, $L_0(E, x') \equiv \mathsf{nd}(y, x', z)$ expresses that z is stored in cell $y{+}1$.

The three forgetting rules drop atomic predicates. Forget1 removes empty blocks, and Forget2 drops list segments and nodes that cannot be accessed in a "known" way. In the presence of pointer arithmetic we can never conclude that a cell is absolutely inaccessible. Rather, if we cannot be sure of how to safely access it then our analysis decides to forget about it. Forget3 forces abstraction to establish the fictional view of memory: when we have a cell or a block that has not been made into a node in the synthesis phase, we forget it.

[3] The rules are slight modifications of the ones in [15], different because of the possible cyclicity of list segments in this paper.

Table 4. Rules for Multiword-List Abstraction

Notation: $L(E,F) ::= \mathsf{mls}(E,F) \mid \mathsf{nd}(E,F,H) \quad U(E,F) ::= \mathsf{blk}(E,F) \mid E{\mapsto}F$

Append Rule

$Q * L_0(E,x') * L_1(x',G) \;\Rightarrow\; Q * \mathsf{mls}(E,G)$

(if $x' \notin \mathsf{fv}(Q,G)$ and $(L_0 \equiv \mathsf{nd}(E,x',F) \;\Rightarrow\; E$ or F is a primed variable))

Forget1 Rule	Forget2 Rule	Forget3 Rule
$Q * \mathsf{blk}(E,E) \;\Rightarrow\; Q * \mathsf{emp}$	$Q * L(x',E) \;\Rightarrow\; Q * \mathsf{true}$ (if $x' \notin \mathsf{fv}(Q)$)	$Q * U(E,F) \;\Rightarrow\; Q * \mathsf{true}$

Filtering Inconsistent Symbolic Heaps. Finally, the analysis filters out symbolic heaps that are proved to be inconsistent by our theorem prover[4]. Concretely, given the result $\mathcal{S} \in \mathcal{D}$ of the previous four abstraction steps, the last step returns $\mathcal{S}'$ defined by:

$$\mathcal{S}' \stackrel{\text{def}}{=} \textbf{if } (\mathcal{S} = \top) \textbf{ then } \top \textbf{ else } \{Q \in \mathcal{S} \mid Q \not\vdash \mathsf{false}\}\,.$$

4.4 *n*-Canonical Symbolic Heaps

The results of Abs form a subdomain $\mathcal{C}_n$ of $\mathcal{D}$, whose elements we call *n-canonical symbolic heaps*. In this section, we define $\mathcal{C}_n$, and we prove the result that relates canonical symbolic heaps to the termination of the analysis.

Let n be a nonnegative integer. Intuitively, a symbolic heap Q is n-canonical if it is n-simple and uses primed variables in the first position of spatial predicates only for two purposes: to represent shared addresses, or to represent the destination of the link field of a node that is pointed to by a program variable. For instance, the following symbolic heaps are n-canonical:

$$\mathsf{mls}(x,x') * \mathsf{mls}(y,x') * \mathsf{mls}(x',z), \quad \mathsf{nd}(x,x',y) * \mathsf{mls}(x',z).$$

They are both n-simple, and they use the primed variable x' for one of the two allowed purposes. In the first case, x' expresses the first shared address of the two lists x and y, and in the second case, x' means the link field of a node that is pointed to by the program variable x.

To give the formal definition of n-canonical symbolic heap, we introduce some preliminary notions. An expression E *occurs left (resp. right)* in a symbolic heap Q iff there exists a spatial predicate in Q where E occurs as the first (resp. second) parameter. E is *shared* in Q iff it has at least two right occurrences. E *is directly pointed to* in Q iff Q contains $\mathsf{nd}(E_1,E,E_2)$ where both E_1 and E_2 are expressions without primed variables.

[4] For Proposition 4 below the prover must at least detect inconsistency when a symbolic heap explicitly defines the same location twice: Q contains $A_1(E) * A_2(E)$ where A_i ranges over $\mathsf{mls}(E,F)$, $E{\mapsto}F$ and $\mathsf{nd}(E,F,G)$.

Definition 1 (n-Canonical Form). *A symbolic heap Q is* n-canonical *iff*

1. *it is n-simple and $Q \not\vdash \mathsf{false}$,*
2. *it contains neither* blk *nor* $\mapsto$*,*
3. *if x' occurs left in Q, it is either shared or directly pointed to, and*
4. *if x' occurs as size of a node predicate in Q, it occurs only once in Q.*

We define CSH_n to be the set of n-canonical symbolic heaps, and write $\mathcal{C}_n$ for the restriction of $\mathcal{D}$ by CSH_n, that is $\mathcal{C}_n = \mathcal{P}(\mathsf{CSH}_n) \cup \{\top\}$.

Proposition 2 (Canonical Characterization). *Let $Q \in \mathsf{SH}$ be n-simple and such that $Q \not\vdash \mathsf{false}$. Q is n-canonical iff $Q \not\Rightarrow$ for rules in Section 4.3.*

Corollary 3. *The range of the abstraction function* Abs *is precisely $\mathcal{C}_n$.*

The main property of n-canonical symbolic heaps is that there are only finitely many of them. This fact is used in Section 6 for the termination of our analysis.

Proposition 4. *The domain $\mathcal{C}_n$ is finite.*

5 Widening Operator

In this section, we define a *widening* operator $\nabla : \mathcal{C}_n \times \mathcal{C}_n \to \mathcal{C}_n$, which is used to accelerate the fixpoint computation of the analysis.

Intuitively, the widening operator ∇ is an optimization of the ($\top$-extended) set union. When ∇ is given two sets $\mathcal{S}, \mathcal{S}'$ of symbolic heaps, it adds to $\mathcal{S}$ only those elements of $\mathcal{S}'$ that add new information. So, $\gamma(\mathcal{S} \nabla \mathcal{S}')$ and $\gamma(\mathcal{S} \cup \mathcal{S}')$ should be equal. For instance, when ∇ is given

$$\mathcal{S} = \{\mathsf{mls}(x, 0)\} \quad \text{and} \quad \mathcal{S}' = \{x{=}0 \wedge \mathsf{emp},\ \mathsf{nd}(x, 0, y),\ \mathsf{nd}(x, y', y) * \mathsf{nd}(y', 0, z)\},$$

it finds out that only the symbolic heap $x{=}0 \wedge \mathsf{emp}$ of $\mathcal{S}'$ adds new information to $\mathcal{S}$. Then, ∇ combines that symbolic heap with $\mathcal{S}$, and returns

$$\{\mathsf{mls}(x, 0),\ x{=}0 \wedge \mathsf{emp}\}.$$

The formal definition of ∇ is parameterized by the theorem prover $\vdash$ for showing some (not necessarily all) semantic implications between symbolic heaps. Let rep be a procedure that takes a finite set $\mathcal{S}$ of symbolic heaps and returns a subset of $\mathcal{S}$ such that

$$(\forall Q, Q' \in \mathsf{rep}(\mathcal{S}).\, Q \vdash Q' \ \Rightarrow\ Q = Q') \ \wedge\ (\forall Q \in \mathcal{S}.\, \exists Q' \in \mathsf{rep}(\mathcal{S}).\, Q \vdash Q')$$

The first conjunct forces rep to get rid of some redundancies while the second, in conjunction with the assumption $\mathsf{rep}(\mathcal{S}) \subseteq \mathcal{S}$, ensures that $\gamma(\mathsf{rep}(\mathcal{S})) = \gamma(\mathcal{S})$. (Our implementation of rep selects $\vdash$-maximal elements of $\mathcal{S}$; in case two elements are $\vdash$-equivalent a fixed ordering on symbolic heaps is used to select one.)

Using $\vdash$ and rep, we define ∇ as follows:

$$\mathcal{S} \nabla \mathcal{S}' = \begin{cases} \mathcal{S} \cup \{Q' \in \mathsf{rep}(\mathcal{S}') \mid \neg(\exists Q \in \mathcal{S}.\, Q' \vdash Q)\} & \text{if } \mathcal{S} \neq \top \text{ and } \mathcal{S}' \neq \top \\ \top & \text{otherwise} \end{cases}$$

This definition requires heaps added to $\mathcal{S}$ to, first, not imply any elements in $\mathcal{S}$ (that would be redundant) and, second, to be "maximal" in the sense of rep.

Our operator ∇ satisfies nonstandard axioms [11], which have also been used in the work on the ASTRÉE analyzer [28, 12].

Proposition 5. *The ∇ operator satisfies the following two axioms:*

1. *For all $\mathcal{S}, \mathcal{S}' \in \mathcal{C}_n$, we have that $\gamma(\mathcal{S}) \cup \gamma(\mathcal{S}') \subseteq \gamma(\mathcal{S} \nabla \mathcal{S}')$.*
2. *For every infinite sequence $\{\mathcal{S}'_i\}_{i \geq 0}$ in $\mathcal{C}_n$, the widened sequence $\mathcal{S}_0 = \mathcal{S}'_0$ and $\mathcal{S}_{i+1} = \mathcal{S}_i \nabla \mathcal{S}'_{i+1}$ converges.*

The first axiom means that ∇ overapproximates the concrete union operator, and it ensures that the analysis can use ∇ without losing soundness. The next axiom means that ∇ always turns a sequence into a converging one, and it guarantees that ∇ does not disturb the termination of the analysis.

The first axiom above is not standard. Usually [9], one uses a stronger axiom where ∇ is required to be extensive for both arguments, i.e.,

$$\forall \mathcal{S}, \mathcal{S}' \in \mathcal{C}_n.\ \mathcal{S} \sqsubseteq (\mathcal{S} \nabla \mathcal{S}') \ \wedge \ \mathcal{S}' \sqsubseteq (\mathcal{S} \nabla \mathcal{S}').$$

However, we cannot use this usual stronger axiom here, because our widening operator is not extensive for the second argument. For a counterexample, consider $\mathcal{S} = \{\mathsf{mls}(x, y)\}$ and $\mathcal{S}' = \{\mathsf{nd}(x, y, z)\}$. We do not have $\mathcal{S}' \sqsubseteq (\mathcal{S} \nabla \mathcal{S}')$ because the rhs is $\{\mathsf{mls}(x, y)\}$, which does not include $\mathcal{S}'$. Note that although $\mathcal{S}' \not\sqsubseteq (\mathcal{S} \nabla \mathcal{S}')$, we still have that

$$\gamma(\mathcal{S}') \ = \ \gamma(\mathsf{nd}(x, y, z)) \ \subseteq \ \gamma(\mathsf{mls}(x, y)) \ = \ \gamma(\mathcal{S} \nabla \mathcal{S}'),$$

as demanded by our first axiom for the ∇ operator.

Note that $S \nabla S'$ is usually smaller than the join of S and S' in $\mathcal{C}_n$. Thus, unlike other typical widening operators, our ∇ does not cause the analysis to lose precision, while making fixpoint computations converge in fewer iterations.

The widening operator is reminiscent of the ideas behind the Hoare powerdomain, and one might wonder why we do not use those ideas more directly by changing the abstract domain. For instance, one might propose to use the domain $\mathcal{C}'$ that consists of $\top$ and sets $\mathcal{S}$ of symbolic heaps where no (provably) redundant elements appear (i.e., for all $Q, Q' \in \mathcal{S}$, if $Q \vdash Q'$, then $Q = Q'$). The Hoare order of $\mathcal{C}'$ would be

$$\mathcal{S} \sqsubseteq_H \mathcal{S}' \iff \mathcal{S}' = \top \ \vee \ (\mathcal{S}, \mathcal{S}' \in \mathcal{P}(\mathsf{CSH}_n) \ \wedge \ \forall Q \in \mathcal{S}.\, \exists Q' \in \mathcal{S}'.\, Q \vdash Q').$$

Unfortunately, the proposal relies on the transitivity of the provable order $\vdash$, which is nontrivial for a theorem prover to achieve in practice. If $\vdash$ is not transitive, then $\sqsubseteq_H$ is not transitive. Hence, the fixpoint computation of the analysis might fail to detect that it has already reached a fixpoint, which can cause the analysis to diverge.

On the other hand, our approach based on widening does not require any additional properties of a theorem prover other than its soundness. And neither does it require that the transfer functions be monotone wrt $\vdash$; it only requires that they be sound overapproximations. So, our approach is easier to apply.

Table 5. Abstract Semantics

Let $A[e]$ and A be syntactic subclasses of atomic commands defined by:

$$A[e] ::= [e]{:=}\, e \mid x{:=}\,[e] \qquad A ::= x{:=}\, e \mid x{:=}\,\mathsf{sbrk}(e).$$

The abstract semantics $[\![C]\!]\colon \mathcal{D} \to \mathcal{D}$ is defined as follows:

$$\begin{aligned}
[\![C_0\,;C_1]\!]\mathcal{S} &= ([\![C_1]\!] \circ [\![C_0]\!])\mathcal{S}\\
[\![\texttt{if}(B)\ \{C_0\}\ \texttt{else}\ \{C_1\}]\!]\mathcal{S} &= ([\![C_0]\!] \circ \mathsf{filter}(B))\mathcal{S} \sqcup ([\![C_1]\!] \circ \mathsf{filter}(\neg B))\mathcal{S}\\
[\![\texttt{local}\ x\,;C]\!]\mathcal{S} &= \textbf{if}\,([\![C]\!](\mathcal{S}[y'/x]) = \top)\ \textbf{then}\ \top\ \textbf{else}\ ([\![C]\!](\mathcal{S}[y'/x]))[x'/x]\\
[\![\texttt{while}(B)\{C\}]\!]\mathcal{S} &= (\mathsf{filter}(\neg B) \circ \mathit{wfix})(\mathcal{S}_0, F)\\
&\qquad (\text{where } \mathcal{S}_0 = \mathsf{Abs}(\mathcal{S}) \text{ and } F = \mathsf{Abs} \circ [\![C]\!] \circ \mathsf{filter}(B))\\
[\![A[e]]\!]\mathcal{S} &= \textbf{if}\ (\mathcal{S} = \top\ \vee\ \exists Q \in \mathcal{S}.\, Q \rightsquigarrow_e^* \text{fault})\ \textbf{then}\ \top\\
&\qquad \textbf{else}\ \{Q_1 \mid Q \in \mathcal{S}\ \wedge\ Q \rightsquigarrow_e^* Q_0\ \wedge\ (Q_0, A[e] \Longrightarrow Q_1)\}\\
[\![A]\!]\mathcal{S} &= \textbf{if}\ (\mathcal{S} = \top)\ \textbf{then}\ \top\ \textbf{else}\ \{Q_0 \mid Q \in \mathcal{S}\ \wedge\ (Q, A \Longrightarrow Q_0)\}
\end{aligned}$$

where primed variables are assumed fresh, and $\mathsf{filter}\colon \mathcal{D} \to \mathcal{D}$ and $-\colon \mathcal{C}_n \times \mathcal{C}_n \to \mathcal{C}_n$ and $\mathit{wfix}\colon \mathcal{C}_n \times [\mathcal{C}_n \to \mathcal{C}_n] \to \mathcal{C}_n$ are functions defined below:

$$\mathsf{filter}(B)(\mathcal{S}) = \textbf{if}\ (\mathcal{S}{=}\top)\ \textbf{then}\ \top\ \textbf{else}\ \{B \wedge Q \mid Q \in \mathcal{S}\ \text{ and }\ (B \wedge Q \not\vdash \mathsf{false})\}.$$

$$\mathcal{S}_0 - \mathcal{S}_1 = \textbf{if}\ (\mathcal{S}_0{\neq}\top \wedge \mathcal{S}_1{\neq}\top)\ \textbf{then}\ (\mathcal{S}_0 - \mathcal{S}_1)\ \textbf{else}\ \Big(\textbf{if}\ (\mathcal{S}_1{=}\top)\ \textbf{then}\ \emptyset\ \textbf{else}\ \top\Big)$$

$\mathit{wfix}(\mathcal{S}, F)$ is the first stabilizing element $\mathcal{S}_k$ of the below sequence $\{\mathcal{S}_i\}_{i\geq 0}$:

$$\mathcal{S}_0 = \mathcal{S} \qquad \mathcal{S}_1 = \mathcal{S}_0 \nabla F(\mathcal{S}) \qquad \mathcal{S}_{i+2} = \mathcal{S}_{i+1} \nabla (F(\mathcal{S}_{i+1} - \mathcal{S}_i)).$$

6 Abstract Semantics

The abstract semantics of our language follows the standard denotational-style abstract interpretation. It interprets commands as $\top$-preserving functions on $\mathcal{D}$ in a compositional manner. The semantic clauses for commands are given in Table 5. In the table, we use the macro $\neg$ that maps $E{=}F$, $E{\neq}F$, $E{\leq}F$ to $E{\neq}F$, $E{=}F$, $F{\leq}E{-}1$, respectively. The semantics of compound commands other than while loops is standard. The interpretation of while loops, however, employs non-standard techniques. First, when the interpretation of the loop does the fixpoint computation, it switches the abstract domain from $\mathcal{D}$ to the finite subdomain $\mathcal{C}_n$, thereby ensuring the termination of the fixpoint computation. Concretely, given a loop $\texttt{while}(B)\,\{C\}$ and an initial abstract element $\mathcal{S}$, the semantics constructs F and $\mathcal{S}_0$ with types $\mathcal{C}_n \to \mathcal{C}_n$ and $\mathcal{C}_n$, respectively. Then, the semantics uses F and $\mathcal{S}_0$, and does the fixpoint computation in the finite domain $\mathcal{C}_n$. Note the major role of the abstraction function Abs here; Abs abstracts the initial abstract element $\mathcal{S}$, and it is used in F to abstract the analysis results of the loop body, so that the fixpoint computation lives in $\mathcal{C}_n$.

Intuitively, the analysis works at the "RAM level" inside loops and the higher, structured, view of lists and nodes is re-established at every iteration. For the

purpose of the analysis described in this paper, this is a necessary choice since very often the precision of the RAM level is needed to meaningfully execute atomic commands; abstracting at every step usually produces imprecise results.

Second, the abstract semantics of while loops uses an optimized fixpoint algorithm called *widened differential fixpoint algorithm*. The main idea of this algorithm is to use two known optimization techniques. The first is to use a widening operator to help the analysis reach a fixpoint in fewer iterations [9].[5] The second is to use the technique of subtraction to prevent the analysis from repeating the same computation in two different iteration steps [17]. Given an abstract element $\mathcal{S} \in \mathcal{C}_n$ and an abstract transfer function $F{:}\,\mathcal{C}_n \to \mathcal{C}_n$, the widened differential fixpoint algorithm generates a sequence whose $i{+}2$-th element has the form:

$$\mathcal{S}_{i+2} = \mathcal{S}_{i+1} \nabla (F(\mathcal{S}_{i+1} {-} \mathcal{S}_i)),$$

and returns the first stabilizing element of this sequence.[6] The key to the algorithm is to use the widening operator, instead of the join operator, to add newly generated analysis results to $\mathcal{S}_{i+1}$, and to apply F to the subtracted previous step $\mathcal{S}_{i+1} {-} \mathcal{S}_i$, rather than to the whole previous step $\mathcal{S}_{i+1}$.

Usually, these two techniques have been used separately in the denotational-style abstract interpretation. The problem is that the common soundness argument for the subtraction technique does not hold in the presence of widening. The argument relies on at least one of monotonicity, extensiveness or distributivity of F[7] but if the widening operator is used (to define F itself), F does not necessarily have any of these properties. In Section 6.2, we use an alternative approach for showing the soundness of the analysis and prove the correctness of the widened differential fixpoint algorithm.

Finally, the semantic clauses for atomic commands are given following the rules of symbolic execution in [3]. The semantics classifies atomic commands into two groups depending on whether they access existing heap cells. When an atomic command accesses an existing heap cell e, such as $[e]{:=}\,e_0$ and $x{:=}\,[e]$, the semantics first checks whether the input $\mathcal{S}$ always guarantees that e is allocated. If not, the semantics returns $\top$, indicating the possibility of memory errors. Otherwise, it transforms each symbolic heap in $\mathcal{S}$ using the rearrangement rules $\rightsquigarrow^*$ (see Section 6.1) in order to expose cell e. Then, the semantics symbolically runs the command using $\Longrightarrow$ in Table 6. For the atomic commands that do not access heap cells, the semantics skips the rearrangement step.

6.1 Rearrangement Rules

When symbolic execution attempts to access a memory cell e which is not explicitly indicated in the symbolic heap, it appeals to a set of rules (called rearrangement rules) whose purpose is to rewrite the symbolic heap in order to

[5] Theoretically, since the analyzer works on the set of canonical heaps which is finite, the use of widening is not necessary for termination. However, widening significantly speeds up the convergence of the fixpoint computation.

[6] $\mathcal{S}_k$ is a stabilizing element iff $\mathcal{S}_k = \mathcal{S}_{k+1}$.

[7] F is distributive iff for all $\mathcal{S}, \mathcal{S}'$, $F(\mathcal{S} \sqcup \mathcal{S}') = F(\mathcal{S}) \sqcup F(\mathcal{S}')$.

Table 6. Rules of Symbolic Execution

$Q,\ x{:=}\,e$	$\Longrightarrow x{=}e[x'/x] \wedge Q[x'/x]$	
$Q * e{\mapsto}F,\ x{:=}\,[e]$	$\Longrightarrow x{=}F[x'/x] \wedge (Q * e{\mapsto}F)[x'/x]$	
$Q * e_0{\mapsto}F,\ [e_0]{:=}\,e_1$	$\Longrightarrow Q * e_0{\mapsto}e_1$	
$Q,\ x{:=}\,\mathsf{sbrk}(e)$	$\Longrightarrow Q[x'/x] * \mathsf{blk}(x, x{+}(e[x'/x]))$	(when $Q \vdash e{>}0$)
$Q,\ x{:=}\,\mathsf{sbrk}(e)$	$\Longrightarrow x{=}{-}1 \wedge Q[x'/x]$	

where primed variables are assumed fresh.

Table 7. Rearrangement Rules for Built-in Predicates

Switch Rule	Blk Rule
Precondition: $F{=}e$	Precondition: $F{\leq}e{<}G$ with x' fresh
$Q * F{\mapsto}G \ \leadsto_e\ Q * e{\mapsto}G$	$Q * \mathsf{blk}(F,G) \ \leadsto_e\ Q * \mathsf{blk}(F,e) * e{\mapsto}x' * \mathsf{blk}(e{+}1,G)$

reveal e. In this paper, we have three sets of rearrangement rules. The first set, in Table 7, handles built-in predicates. The rules in the second set follow the general pattern

Allocated: e
Consistency: $Q * Q'[\vec{F}/\vec{x}]$
$Q * H(\vec{F}) \ \leadsto_e\ Q * Q'[\vec{F}/\vec{x}]$

Here $H(\vec{F})$ is either nd or mls, and Q' is one of the disjuncts in the definition of H with all the primed variables in Q' renamed fresh. Instead of the precondition requirement as in the abstraction rules in Section 4, the rules generated by this pattern have an *allocatedness* requirement and a *consistency* requirement. Allocatedness guarantees that the heap contains the cell e that we are interested in. This correspond to the check:

$$\Pi_Q \wedge (H(\vec{F}) * e{\mapsto}x') \vdash \mathsf{false}$$

where Π_Q is the pure part of Q and x' is a fresh primed variable. The consistency requirement of the rule enforces its post-state to be meaningful. This means that in order for the rule to fire the following extra consistency condition should hold:

$$Q * Q'[\vec{F}/\vec{x}] \nvdash \mathsf{false}.$$

The rearrangement rules for nd and mls are reported in Table 8.

The third set consists of a single rule that detects the possibility of memory faults and it is described below:

Fault Rule $\quad Q \ \leadsto_e\ \text{fault} \quad$ (if Q does not contain $e{\mapsto}F$, but $Q \not\leadsto_e$).

Table 8. Rearrangement Rules for Multiword Lists

Mls1 Rule	Mls2 Rule
Allocated: e	Allocated: e
Consistency: $Q * \mathsf{nd}(F, G, z')$	Consistency: $Q * \mathsf{nd}(F, y', z') * \mathsf{mls}(y', G)$
$Q * \mathsf{mls}(F, G) \rightsquigarrow_e Q * \mathsf{nd}(F, G, z')$	$Q * \mathsf{mls}(F, G) \rightsquigarrow_e Q * \mathsf{nd}(F, y', z') * \mathsf{mls}(y', G)$

Node Rule
Allocated: e
Consistency: $-$
$Q * \mathsf{nd}(F, G_0, G_1) \rightsquigarrow_e Q * F{\mapsto}G_0, G_1 * \mathsf{blk}(F{+}2, F{+}G_1)$

6.2 Soundness

Common soundness arguments in program analysis proceed by showing that the results of an analysis are prefix points of some abstract transfer functions. Then one derives that the analysis results overapproximate program invariants. Unfortunately, we cannot use this strategy, because the fixpoint algorithm described here does not necessarily compute prefix points of abstract transfer functions.

We use an alternative approach that proves soundness by compiling the analysis results into proofs in separation logic. More specifically, we prove:

Proposition 6. *Suppose that $[\![C]\!]\mathcal{S} = \mathcal{S}'$. If both $\mathcal{S}$ and $\mathcal{S}'$ are non-$\top$ abstract values, then there is a proof of a Hoare triple $\{\mathcal{S}\}\, C\, \{\mathcal{S}'\}$ in separation logic.*

Note that since the proof rules of separation logic are sound, this proposition implies that the results of our analysis overapproximate program invariants.

The proposition can be proved by induction on the structure of C. Most of the cases follow immediately, because the abstract semantics uses sound implications between assertions or proof rules in separation logic. Cases like these have been done in previous work [21, 15, 18], and we will not repeat them here. The treatment of while loops, however, requires a different argument.

Suppose that $[\![\texttt{while}(B)\,\{C\}]\!]\mathcal{S} = \mathcal{S}'$ for some non-$\top$ elements $\mathcal{S}$ and $\mathcal{S}'$. Let $\mathcal{S}''$ and F be the two parameters, $\mathsf{Abs}(S)$ and $\mathsf{Abs} \circ [\![C]\!] \circ \mathsf{filter}(B)$, of *wfix* in the interpretation of this loop. Then, by the definition of *wfix*, the abstract element $\mathit{wfix}(\mathcal{S}'', F)$ is the first stabilizing element $\mathcal{S}_k$ of the sequence $\{S_i\}_{i\geq 0}$:

$$\mathcal{S}_0 = \mathcal{S}'' \qquad \mathcal{S}_1 = \mathcal{S}_0 \nabla F(\mathcal{S}) \qquad \mathcal{S}_{i+2} = \mathcal{S}_{i+1} \nabla F(\mathcal{S}_{i+1} - \mathcal{S}_i).$$

Moreover, we have that $\mathcal{S}' = \mathsf{filter}(\neg B)(\mathcal{S}_k)$. The following lemma summarizes the relationship among $\mathcal{S}$, $\mathcal{S}_0$, ..., $\mathcal{S}_{k-1}$ and $\mathcal{S}_k$, which we use to construct the separation-logic proof for the loop.

Lemma 7. *For all $i \in \{1, \ldots, k\}$, let $\mathcal{T}_i$ be $\mathcal{S}_i - \mathcal{S}_{i-1}$.*

1. *None of $\mathcal{S}_0$, $\mathcal{T}_1$, ..., $\mathcal{T}_k$ and $\mathcal{S}_k$ is $\top$.*
2. *$\mathcal{S} \Rightarrow \mathcal{S}_k$.*

Program	LOC	Max Heap (KB)	States (Loop Inv)	States (Post)	Time (sec)
malloc_firstfit_acyclic	42	240	18	3	0.05
free_acyclic	55	240	6	2	0.09
malloc_besttfit_acyclic	46	480	90	3	1.19
malloc_roving	61	240	33	5	0.13
free_roving	68	720	16	2	0.84
malloc_K&R	179	26880	384	66	502.23
free_K&R	58	3840	89	5	9.69

Fig. 2. Experimental Results

3. $\mathcal{S}_k \Rightarrow \mathcal{S}_0 \vee \mathcal{T}_1 \vee \ldots \vee \mathcal{T}_k$.
4. $F(\mathcal{S}_0) \Rightarrow \mathcal{S}_k$ *and* $F(\mathcal{T}_i) \Rightarrow \mathcal{S}_k$ *for all* $i \in \{1, \ldots, k\}$.

We now construct the required proof. Because of the fourth property in Lemma 7 and the induction hypothesis, we can derive the following proof trees:

$$\dfrac{\dfrac{\dfrac{\text{Ind. Hypo.}}{\{\mathsf{filter}(B)(\mathcal{U})\}\, C\, \{([\![C]\!] \circ \mathsf{filter}(B))(\mathcal{U})\}}}{\{B \wedge \mathcal{U}\}\, C\, \{([\![C]\!] \circ \mathsf{filter}(B))(\mathcal{U})\}}\ (B \wedge \mathcal{U} \Leftrightarrow \mathsf{filter}(B)(\mathcal{U}))}{\{B \wedge \mathcal{U}\}\, C\, \{F(\mathcal{U})\}}\ \text{Soundness of } \mathsf{Abs},\ F = \mathsf{Abs} \circ [\![C]\!] \circ \mathsf{filter}(B)}{\{B \wedge \mathcal{U}\}\, C\, \{\mathcal{S}_k\}}\ \text{Prop.4 of Lem.7}$$

twhere $\mathcal{U}$ is $\mathcal{S}_0$ or $\mathcal{T}_i$. We combine those proof trees, and build the required tree for the loop:

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\{B \wedge \mathcal{S}_0\}\, C\, \{\mathcal{S}_k\}} \quad \overline{\{B \wedge \mathcal{T}_1\}\, C\, \{\mathcal{S}_k\}} \quad \ldots \quad \overline{\{B \wedge \mathcal{T}_k\}\, C\, \{\mathcal{S}_k\}}}{\{(B \wedge \mathcal{S}_0) \vee (B \wedge \mathcal{T}_1) \vee \ldots (B \wedge \mathcal{T}_k)\}\, C\, \{\mathcal{S}_k\}}\ \text{Disjunction}}{\{B \wedge \mathcal{S}_k\}\, C\, \{\mathcal{S}_k\}}\ \text{Prop.3 of Lem.7}}{\{\mathcal{S}_k\}\, \mathtt{while}(B)\, \{C\}\, \{\neg B \wedge \mathcal{S}_k\}}\ \text{While}}{\{\mathcal{S}\}\, \mathtt{while}(B)\, \{C\}\, \{\mathcal{S}'\}}\ \text{Prop.2 of Lem.7},\ \mathcal{S}' = \mathsf{filter}(\neg B)\mathcal{S}_k$$

Note that the proof tree indicates that the subtraction technique of our fixpoint algorithm corresponds to the disjunction rule

$$\dfrac{\{P_1\}\ C\ \{Q_1\} \quad \{P_2\}\ C\ \{Q_2\}}{\{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

of Hoare logic.

7 Experimental Results

We implemented our analysis in OCaml, and conducted experiments on an Intel Pentium 3.2GHz with 4GB RAM; our results are in Figure 2. Our implementation contains a postprocessor that simplifies computed post abstract values

using the abstraction Abs and rep in Section 5. `malloc_K&R` is the only one of the programs with a nested loop; the size of the invariant for it in Figure 2 refers to the outer loop. (The test-programs can be found on the authors' web pages.)

The `malloc_firstfit_acyclic` and `free_acyclic` programs are Algorithms A and B from Section 2.5 of [20]. They both manage an acyclic free list maintained in address-sorted order. `malloc_firstfit_acyclic` walks the free list until a big enough block is found to satisfy a malloc request. That block is returned to the caller if the size is exactly right, and otherwise part of the block is chopped off and returned to the caller, with the leftover resized and kept in the free list. If a correctly-sized block is not found then the algorithm returns 0. The `free` algorithm inserts a block in the appropriate place in this list, maintaining sorted order and coalescing nodes when possible. `malloc_bestfit_acyclic` traverses the entire free list to find the best fit for a request, and returns it.

In simplistic first-fit allocators small blocks tend to pile up at the front of the free list. A way to combat this problem is to use a cyclic rather than acyclic free list, with a "roving pointer" that moves around the list [20]. The roving pointer points to where the last allocation was done, and the next allocation starts from it. `malloc_roving` and `free_roving` implement this strategy.

The first five programs assume that a fixed amount of memory has been given to the memory manager at the beginning. A common strategy is to extend this by calling a system routine to request additional memory when a request cannot be met. This is the strategy used in the memory manager from Section 8.7 of [19]. When a request cannot be met, `sbrk` is called for additional memory. In case `sbrk` succeeds the memory it returns is inserted into the free list by calling `free`, and then allocation continues. The memory manager there uses the roving pointer strategy. To model this program in the programming language used for our analysis we had to inline the call to `free`, as what we have described is not an interprocedural analysis. Also, we had to model multiple-dereferences like `p→s.ptr→s.size` using several statements, as the form in our language has at most one dereference per statement; this is akin to what a compiler front end might do. These points, inline free and basic dereferences, account for the 179 LOC in our program for malloc compared to 66 LOC in the original.

The manager in [19] uses a nonempty circular list with a fixed head node that is never returned to the caller. Its correctness relies on the (unstated) assumption that `sbrk` will return a block whose address is larger than the head node; otherwise, there are cases in which the header will be coalesced with a block gotten from `sbrk`, and this can lead to a situation where the same block is allocated twice in a row. Our model of `sbrk` in Section 6 does not make this assumption explicit, and as a result running the analysis on the original `malloc` reveals this "problem". By changing our model of `sbrk` to reflect the assumption we were able to verify the original; the model, though, is not as simple as the one in Section 6. We then altered `malloc` so that it did not rely on this assumption, and this is the program `malloc_K&R` reported in Figure 2, for which we used the simple `sbrk`.

The speedup obtained from the widening operator can be observed in the analysis of `malloc_K&R`; with widening turned off the analysis took over 20 hours

Prog : `malloc_firstfit_acyclic` and `malloc_bestfit_acyclic`
Pre : $\mathtt{n}{-}2{\geq}0 \wedge \mathsf{mls}(\mathtt{free}, 0)$
Post : $(\mathtt{ans}{=}0 \wedge \mathtt{n}{-}2{\geq}0 \wedge \mathsf{mls}(\mathtt{free}, 0)) \;\vee\; (\mathtt{n}{-}2{\geq}0 \wedge \mathsf{nd}(\mathtt{ans}, \mathtt{p}', \mathtt{n}) * \mathsf{mls}(\mathtt{free}, 0))$
$\vee\; (\mathtt{n}{-}2{\geq}0 \wedge \mathsf{nd}(\mathtt{ans}, \mathtt{p}', \mathtt{n}) * \mathsf{mls}(\mathtt{free}, \mathtt{p}') * \mathsf{mls}(\mathtt{p}', 0))$

Prog : `free_acyclic`
Pre : $\mathsf{mls}(\mathtt{free}, 0) * \mathsf{nd}(\mathtt{ap}{-}2, \mathtt{p}', \mathtt{n}')$
Post : $\mathsf{mls}(\mathtt{free}, \mathtt{ap}{-}2) * \mathsf{mls}(\mathtt{ap}{-}2, 0) \;\vee\; \mathsf{mls}(\mathtt{free}, 0)$

Fig. 3. Sample Computed Post Abstract Values

to terminate. For the other programs the analysis times were similar, except for `free_K&R` where widening resulted in a speedup of a factor of 2.

Memory safety and memory leaks are general properties, in the sense that they can be specified once and for all for all programs. For our analysis if a postcondition is not $\top$ then it follows that the program does not dereference a dangling pointer starting from any concrete state satisfying the precondition. If true does not appear in the post then the program does not have a memory leak. For all seven programs, our analysis was able to prove memory safety and the absence of memory leaks.

But in fact we can infer much more: the postconditions give what might be regarded as full functional specifications.[8] Figure 3 shows sample post abstract values computed by the analysis (with widening enabled). Take the postcondition for `malloc_firstfit_acyclic` and `malloc_bestfit_acyclic`. The first disjunct, when ans=0, is the case when the algorithm could not satisfy the malloc request. The second and third correspond to cases when the request has been met. These two cases are different because the analysis distinguishes when the link field of a node happens to point back into the free list. The third disjunct implies the second, so if the user were to write the first two disjuncts as the desired postcondition, which would be intuitive, then a theorem prover could tell us that the computed post in fact established a reasonable specification of partial correctness[9]. Similar remarks apply to the other postcondition for `free_acyclic`.

Finally, it is easy to trick the analysis into reporting a false bug. Our algorithm abstracts to the "fictional level" after each loop iteration. If a program fails to package a portion of RAM into a node before leaving a loop, then the unpackaged RAM will be abstracted to true. If then subsequently, after the loop, the program attempts to package up the node with a heap mutation, then the analysis will return $\top$.

[8] The specifications are for *partial* correctness, but using the techniques of [5] we could likely establish termination as well if we were to track lengths of multiword lists.

[9] This specification would not rule out the manager doing things like returning some nodes of the free list to the system, but would be a reasonable spec of the interface to `malloc` nonetheless.

8 Conclusions and Related Work

We believe that the results in this paper may pave the way for improved automatic verification techniques for low-level, "dirty" programs.

As we mentioned earlier, the approach in this paper is a development of the Space Invader shape analysis [15] (also, [22]). Compared to the original, the differences here are the following. First, we use different basic predicates, which are oriented to multiword lists, and different abstraction rules appropriate to the reasoning about multiword lists; this results in a much more complex abstract domain. Some of the abstraction rules, in particular, are perhaps not the first that come to mind, and we settled on them only after some experimentation. Second, in order to accelerate the analysis, we used a particular widening operator. Also, we used much more intricate test-programs in our experiments. These, and the complexity of our abstract domain, are such that the widening had a significant impact on performance.

There have been two previous works on doing mechanical proofs of memory managers in separation logic [33, 25], which both work by embedding separation logic into Coq. They do not consider the exact same algorithms that we do. The most significant difference, though, is degree of automation. They require loop invariants to be provided, and even then the proofs in Coq are not automatic, whereas our proof construction is completely automatic. Or course, working with a proof assistant allows one to say more than is typically done in the lightweight assertions that are used in program analysis. For example, one could say that the free list increases in size on deallocation, where we do not say that here.

Shallow pointer analyses track points-to relationships between fixed-length access paths. They are fast compared to shape analyses, but give imprecise answers on deep heap updates, which occur when linked structures are altered after traversing some distance. There have been a number of shallow pointer analyses that deal with pointer arithmetic before (e.g., [32, 16, 29, 26]), but as far as we are aware not any deep ones.

Between the fast, shallow analyses and the comparatively expensive, deep shape analyses is the recency-abstraction [2]. It is one of the most relevant pieces of related work. The recency-abstraction allows for pointer arithmetic and also connects low-level and high-level, fictional, views of memory. It can distinguish mutations of pointers coming from the same allocation site, but is imprecise on deep heap update. A way to handle deep updates was pioneered in [30], using the method of materialization of summary nodes[10]; recency-abstraction (purposely) avoids materialization, in order to gain efficiency, and experimental results justify the lower precision for its targeted applications. In contrast, here we must handle deep update precisely if we are to obtain reasonable results for the memory management algorithms that we used in our experiments.

An interesting question is whether some other previous shape abstraction might be modified to obtain an effective analysis of multiword lists or similar structures. In any case the problem paper addresses is existence, not uniqueness, of shape abstraction beyond reachability.

[10] The rearrangement rules here and in [3, 15] are cousins of materialization.

Acknowledgments. We are grateful to Josh Berdine, Tom Reps and Mooly Sagiv for comments on pointer analysis and pointer arithmetic, and to Xavier Rival and Kwangkeun Yi for advice on widening and non-standard fixpoint operators. Byron Cook's emphasis on the relevance of analyzing memory managers gave us an initial push. We had helpful discussions on memory models and the K&R `malloc` with Richard Bornat. We acknowledge support from the EPSRC. Yang was partially supported by R08-2003-000-10370-0 from the Basic Research Program of Korea Science & Engineering Foundation.

References

1. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. 6th VMCAI, pp164–180, 2005.
2. G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. 13th SAS (this volume), 2006.
3. J. Berdine, C. Calcagno, and P.W. O'Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS 2005*, volume 3780 of *LNCS*, 2005.
4. J. Berdine, C. Calcagno, and P.W. O'Hearn. Automatic modular assertion checking with separation logic. Proceedings of FMCO'05, to appear, 2006.
5. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. 18th CAV, to appear, 2006.
6. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. 11th TACAS, pp13–29, 2005.
7. D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. PLDI, pp296–310, 1990.
8. W.T. Comfort. Multiword list items. CACM 7(6), pp357-362, 1964.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. 4th POPL, pp238-252, 1977.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. 6th POPL, pp269-282, 1979.
11. P. Cousot and R. Cousot. Abstract interpretation frameworks. J. Log. Comput. 2(4), pp511-547, 1992.
12. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. 14th ESOP, pp21-30, 2005.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. 5th POPL, pp84-96, 1978.
14. D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? On the automated verification of linked list structures. 24th FSTTCS, pp250-262, 2004.
15. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. 16th TACAS, pp287–302, 2006.
16. N. Dor, M. Rodeh, and M. Sagiv. Towards a realistic tool for statically detecting all buffer overflows in C. PLDI, pp155-167, 2003.
17. H. Eo, K. Yi, and H. Eom. Differential fixpoint iteration with subtraction for non-distributive program analysis. Sumitted, 2006.
18. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. 13th SAS, to appear, 2006.

19. B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, New Jersey, 1988. 2nd Edition.
20. D.E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1973. 2nd Edition.
21. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. 14th ESOP, pp124-140, 2005.
22. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in Separation Logic for imperative list-processing programs. 3rd SPACE Workshop, 2006.
23. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. 11th SAS, pp265-279., 2004.
24. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. 6th VMCAI, pp181-198, 2005.
25. N. Marti, R. Affeldt, and A. Yonezawa. Verification of the heap manager of an operating system using separation logic. 3rd SPACE Workshop, 2006.
26. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. *PEPM'06*, pp100-111, 2006.
27. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp 55-74, 2002.
28. Xavier Rival. Personal communication. 2005.
29. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM TOPLAS*, 27(2):185–235, 2005.
30. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.
31. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
32. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. Proceedings of NDSS, 2000.
33. D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. 12th ESOP, pp363–379, 2003.