To appear in: The 14th International Static Analysis Symposium, SAS 2007.

Arithmetic Strengthening for Shape Analysis *

Stephen Magill[1], Josh Berdine[2], Edmund Clarke[1], and Byron Cook[2]

[1] Carnegie Mellon University
[2] Microsoft Research

Abstract. Shape analyses are often imprecise in their numerical reasoning, whereas numerical static analyses are often largely unaware of the shape of a program's heap. In this paper we propose a lazy method of combining a shape analysis based on separation logic with an arbitrary arithmetic analysis. When potentially spurious counterexamples are reported by our shape analysis, the method constructs a purely arithmetic program whose traces over-approximate the set of counterexample traces. It then uses this arithmetic program together with the arithmetic analysis to construct a refinement for the shape analysis. Our method is aimed at proving properties that require comprehensive reasoning about heaps together with more targeted arithmetic reasoning. Given a sufficient precondition, our technique can automatically prove memory safety of programs whose error-free operation depends on a combination of shape, size, and integer invariants. We have implemented our algorithm and tested it on a number of common list routines using a variety of arithmetic analysis tools for refinement.

1 Introduction

Automatic formal software verification tools are often designed either to prove arithmetic properties (*e.g. is* `x` *always greater than 0 at program location 35?*) or data structure properties (*e.g. does* `p` *always point to a well-formed list at program location 45?*). Shape analyses are developed to reason about the linked structure of data on the heap, while arithmetic analyses are designed to reason about the relationships between integer values manipulated by a program. Since integers can be stored in the heap and certain properties of data structures (such as the length of lists) are integer valued, there is non-trivial interaction between

* This research was sponsored by the International Collaboration for Advancing Security Technology (iCAST), the Gigascale Systems Research Center (GSRC), the Semiconductor Research Corporation (SRC) under grant TJ-1366, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, the Defense Advanced Research Projects Agency, the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution.

the two theories. Thus, combining a shape analysis and an arithmetic analysis is not just a matter of applying each analysis separately.

We propose a new technique for combining a shape analysis based on *separation logic* [24] with an arbitrary arithmetic analysis. The combination technique operates by using the arithmetic analysis as a back-end for processing abstract counterexamples discovered during the shape analysis. Our shape analysis is based on those described in [4] and [21]. It is an application of abstract interpretation [11] where the abstract domain uses a fragment of separation logic. As in [4], we assume that the shape analysis supports arithmetic reasoning in its symbolic execution engine, but does not maintain enough arithmetic information in its widening step. To refine this widening step will be the job of the arithmetic analysis tool.

The shape analysis communicates with the arithmetic analysis via *counterexample programs*—integer programs that represent the arithmetic content of the abstract counterexamples. Because the language of communication consists of integer programs, any integer analysis tool can be used without modification to strengthen our shape analysis. Viewed another way, this technique allows any tool targeting integer programs to be applied—again without modification—to programs that manipulate the kinds of heap-based data structures that our shape analysis supports.

In summary, we present a new combination of shape and arithmetic analyses with the following novel collection of characteristics:

- Any arithmetic analysis can be used. The combination is not tied to any particular verification paradigm, and we can use tools based on abstract interpretation, such as Astrée[7], just as easily as those based on model checking, such as Blast[19], SLAM[2], and ARMC[23].
- The arithmetic analysis explicitly tracks integer values which appear quantified in the symbolic states but are absent in the concrete states, such as list lengths. This use of new variables in the arithmetic program to reason about quantified values makes soundness of the combination technique non-obvious. This conjunction under quantifiers aspect also makes it difficult to see the combination technique as an instance of standard abstract domain constructions such as the direct or reduced product, or as a use of Hoare logic's conjunction rule.
- The shape analysis which will be strengthened explores the same abstract state space as the standard one would. That is, we do not explore the cartesian product of the shape and arithmetic state spaces. In this way the combined analysis treats the shape and arithmetic information independently (as in independent attribute analyses) except for the relations between shape and arithmetic information identified by the shape analysis as critical to memory- or assert-safety.
- Arithmetic refinement is performed only on-demand, when the standard shape analysis has failed to prove memory safety on its own.
- Because we track shape information at all program points, our analysis is able to verify properties such as memory-safety and absence of memory leaks.

2 Motivating Example

Consider the example code fragment in the left half of Figure 1. This program creates a list of length n and then deletes it. Neither an arithmetic static analysis nor a traditional shape analysis alone can prove that `curr` is not equal to `NULL` at line 15. As we will see, our analysis is able to prove that this program is memory-safe.

Consider how a shape analysis without arithmetic support would treat this program. Using symbolic execution and widening, the analysis might find an invariant stating that, at location 4, *curr* is a pointer to a well-formed singly-linked and *NULL*-terminated list and i is a pointer to a single heap cell. In separation logic, we would express this invariant as $\exists k, v.\ ls^k(curr, NULL) * i \mapsto v$ where ls is a recursively-defined list predicate and k represents the length of the list. Note that the shape analysis has not attempted to infer any invariance properties of the integer values k and v.

From this point the analysis might explore the path $4 \to 12 \to 13 \to 14 \to 15$, obtaining

$$\exists k.\ ls^k(curr, NULL) \wedge j = 0 \wedge j < n \tag{1}$$

```
 1   List * curr = NULL;
 2   int i = malloc(sizeof(int));
 3   *i = 0;

 4   while(*i < n) {
 5       t = (List*) malloc(sizeof(List));
 6       t->next = curr;
 7       t->data = addr;
 8       addr += next_addr(addr);
 9       curr = t;
10       *i = *i + 1;

11   }
12   free(i);
13   int j = 0;
14   while(j < n) {
15      t = curr->next;




16      free(curr);
17      curr = t;
18      j++;

19   }
```

```
 1   curr = 0;
 2   skip;
 3   int v = 0;
     int k = 0;
 4   while(v < n) {
 5       t = nondet();
 6       skip;
 7       skip;
 8       addr += next_addr(addr);
 9       curr = t;
10       v = v + 1;
         k = k + 1;
11   }
12   skip;
13   int j = 0;
14   while(j < n) {
15      if(k > 0)
           b := nondet();
           t := b;
        else error();
16      skip;
17      curr = t;
18      j++;
        k = k - 1;
19   }
```

Fig. 1. Left: Example showing motivation for combined shape and arithmetic reasoning. Right: Arithmetic counterexample program produced by the shape analysis.

At line 15, the program looks up the value in the *next* field of *curr*. But if the list is empty, then $curr = NULL$ and the lookup will fail. Because (1) does not imply that $curr \neq NULL$, this case cannot be ruled out and the analysis would report a potential violation of memory safety.

However, this case cannot actually arise due to the fact that the second loop frees only as many heap cells as the first loop allocates. To rule out this spurious counterexample, we need to strengthen the invariants associated with the loops, essentially discovering that the value stored in the heap cell at i tracks the length of the list being created in the first loop and j tracks the length of the unprocessed portion of the list in the second loop. Our algorithm achieves this by generating a *counterexample program* representing all paths that satisfy the shape formulas and could lead to the potential memory error.

The program we generate for this counterexample is given in the right half of Figure 1. We have numbered each line with the line number in the original program from which it is derived. Newly added commands are un-numbered. The counterexample program involves two new variables, k and v, which represent the length of the list and the value pointed to by i, respectively.[3] New variables are added whenever the shape analysis encounters an integer value, such as the length of a list or the contents of an integer-valued heap cell.

Note that the control flow of the counterexample program is reminiscent of the control flow of the original program. The only difference here is that the counterexample program has an additional branch at location 15. This corresponds to a case split in the shape analysis—the memory access at location 15 in the original program is safe provided that k (the length of the list) is greater than 0. Also note that heap commands have been replaced by purely arithmetic commands that approximate their effect on the arithmetic program's stack variables. Two examples of this are the command at location 5, where allocation is replaced by nondeterministic assignment, and the command at location 10, where the heap store command that updates the contents of i is replaced by a command that updates the integer variable v.

Another unique aspect of our counterexample programs is that they may contain looping constructs. As such, they represent not just a single counterexample, but rather a set of counterexamples. Returning to the example in Figure 1, recall that the loop invariant at location 4 is $\exists k.\ ls^k(curr, NULL)$. To evaluate the memory safety of the command at location 15, we start with this invariant and compute postconditions along the path from 4 to 15. We then discover that the resulting postcondition is too weak to prove memory safety at location 15 and wish to generate a counterexample. Because the error state in the counterexample follows from the loop invariant at location 4, the counterexample can contain any number of unrollings of this loop. Rather than commit to a specific number and risk making overly specific conclusions based on the counterexample, we instead include a loop in the counterexample program. As we will see, this makes the set of paths through the counterexample program correspond to the full set

[3] The role of the third new variable, b, is more subtle. It arises due to expansion of a definition during theorem proving. This is discussed in detail in Section 4.1.

of abstract counterexamples. This ensures that the arithmetic tool generates a strengthening that rules out all spurious counterexamples (i.e. it is forced to discover a strengthening that is also a loop invariant) and is key to making the collaboration between the shape analysis and arithmetic analysis tool work.

Now let us look at this collaboration in more detail. While trying to prove that `error()` in the counterexample program (Figure 1) is not reachable, an arithmetic analysis tool such as ASTRÉE[7], BLAST [19], or ARMC [23] might prove the following arithmetic invariant at location 15: $k = n - j$. The soundness theorem for our system establishes that this invariant of the arithmetic counterexample program is also an invariant of the original program. As such, it is sound to conjoin this formula with our shape invariant at this location, obtaining $\exists k.\ ls^k(curr, NULL) \wedge k = n - j$. Note that the arithmetic invariant is conjoined inside the scope of the quantifier. This is sound because the variables we add to the counterexample program (such as k) correspond to the existentially quantified variables and their values correspond to the witnesses we used when proving those existential formulas. We formally prove soundness in Section 5.

Now, armed with the strengthened invariant, the shape analysis can rule out the false counterexample of *NULL*-pointer dereference at location 15. We will have the formula $ls^k(curr, NULL) \wedge k = n - j \wedge j < n$, from which we can derive $k > 0$—a sufficient condition for the safety of the memory access.

3 Preliminaries

Our commands include assignment ($e{:=}f$), heap load ($x{:=}[e]$), heap store ($[e]{:=}f$), allocation ($x{:=}\text{alloc}()$), disposal ($\text{free}(e)$), non-deterministic assignment (x := ?), and an `assume` command, which is used to model branch conditions. Note that brackets are used to indicate dereference. We use C to denote the set of commands and the meta-variable c to range over individual commands. The concrete semantics are standard (see [24]) and are omitted. We present only the concrete semantic domains and then move directly to a presentation of the abstract domain and its associated semantics.

The concrete semantic domain consists of pairs (s, h), where s is the *stack* and h is the *heap*. Formally, the stack is simply a mapping from variables to their values, which are either integers or addresses.

$$Val \stackrel{\text{def}}{=} Int \cup Addr$$

$$Stack \stackrel{\text{def}}{=} Var \rightarrow Val$$

The heap is a finite partial function from non-null addresses to *records*, which are functions from a finite set of fields to values: $Record \stackrel{\text{def}}{=} Field \rightarrow Val$, and $Heap \stackrel{\text{def}}{=} (Addr - \{0\}) \stackrel{\text{fin}}{\rightharpoonup} Record$. We also have a state **abort** which is used to indicate failure of a command. This may be occur due to a failed assert statement or an attempt to dereference an address that is not in the domain of the heap.

Our analysis uses a fragment of separation logic [24] as an abstract representation of the contents of the stack and the heap. We have expressions for denoting

addresses and records. Address expressions are simply variables or the constant *NULL*, which denotes the null address. Integer expressions include variables and the standard arithmetic operations. Value expressions refer to expressions that may denote either integers or addresses. Record expressions are lists of field labels paired with value expressions.

$$\begin{array}{ll} \textit{Address} & e, f, g ::= x \mid \mathit{NULL} \\ \textit{Integer Expressions} & m, n ::= x \mid i \mid v_1 + v_2 \mid v_1 - v_2 \mid \ldots \\ \textit{Value Expressions} & v, k ::= e \mid m \\ \textit{Record} & \rho ::= \mathit{label}\colon v, \rho \mid \epsilon \end{array}$$

We assume a standard semantics for expressions and records, such that if $s \in \mathit{Stack}$ then $[\![e]\!]\ s \in \mathit{Addr}$ and $[\![\rho]\!]\ s \in \mathit{Record}$. The meaning of predicates is given in terms of the $\mathit{Stack} \times \mathit{Heap}$ pairs that satisfy them. When giving the semantics, we use sets of pairs to describe the finite partial functions that constitute heaps.

Our predicates are divided into *spatial* predicates, which describe the heap, and *pure* predicates, which describe the stack. The predicate **emp** denotes the empty heap, and $e \mapsto [l_1\colon v_1, l_2\colon v_2, \ldots, l_n\colon v_n]$ describes the heap consisting of a single heap cell at address e that contains a record where field l_1 maps to value v_1, l_2 maps to v_2, etc. The atomic pure predicates include the standard arithmetic predicates ($<$, $\leq$, $=$, etc.) and equality and disequality over address expressions. Spatial formulas are built from conjunctions of atomic spatial predicates using the * connective from separation logic. Intuitively, $P * Q$ is satisfied when the domain of the heap described by P is disjoint from that described by Q. Thus, $(e{\mapsto}\rho_1) * (f{\mapsto}\rho_2)$ implies that $e \neq f$.

We also allow existential quantification and adopt the convention that unmentioned fields are existentially quantified. That is, if a record always contains fields s and t, we write $e{\mapsto}[s\colon v]$ to abbreviate $\exists z.\ e{\mapsto}[s\colon v, t\colon z]$.

From the atomic predicates we can inductively define predicates describing data structures, such as the following predicate for singly-linked list segments.

$$ls^k(e, f) \stackrel{\text{def}}{=} \big(k > 0 \wedge \exists x'.\ e{\mapsto}[n\colon x'] * ls^{k-1}(x', f)\big) \vee \ \big(\mathbf{emp} \wedge k = 0 \wedge e = f\big)$$

The length of the list is given by k, while e denotes the address of the first cell (if the list is non-empty) and f denotes the address stored in the "next" field (n) of the last cell in the list. If the list is empty, then $k = 0$ and $e = f$.

Our implementation actually uses a doubly-linked list predicate. However, in this paper we will use the simpler singly-linked list predicate in order to avoid letting the details of the shape analysis obscure the arithmetic refinement procedure, which is our main focus.

Our abstract states are drawn from the following grammar, where we use the notation $\vec{x}$ to represent a list of variables.

$$\begin{array}{ll} \textit{Spatial Form} & \Sigma ::= e{\mapsto}\rho \mid ls^k(e, f) \mid \mathbf{emp} \mid S_1 * S_2 \\ \textit{Pure Form} & \Pi ::= x \mid e \leq f \mid e = f \mid \neg P \mid P_1 \wedge P_2 \\ \textit{Memory} & M ::= \exists \vec{x}.\ \Sigma \wedge \Pi \mid \top \end{array}$$

The formula $\top$ is satisfied by all concrete states, including **abort**, and is used to indicate failure of a command. Elements of Π are called pure formulas, while elements of Σ are called spatial formulas. We take terms from M as the elements of our abstract domain and refer to them as *abstract state formulas*. We will use the meta-variables S, P, and Q to refer to such formulas.

In the left column of Figure 3, we give the postcondition rules for our commands. These are given as Hoare triples $\{P\}\ c\ \{Q\}$, where P and Q are abstract state formulas. To take the postcondition of state S with respect to command c, we search for an S' such that $S \Rightarrow S'$ and $\{S'\}\ c\ \{Q\}$ is an instance of the rule for c in Figure 3. The formula Q is then the postcondition of the command. If we cannot find such an S', this corresponds to a failure to prove memory safety of command c and the abstract postcondition is $\top$. For more on this process, see the discussion of the "unfold" rule in [21] and the section on "rearrangement rules" in [13].

4 Algorithm

A shape analysis based on separation logic, such as those in [21] and [13], will generate an *abstract transition system* (ATS), which is a finite representation of the reachable states of the program given as a transition system (A) with states labeled by abstract state formulas. Such formulas are either formulas of separation logic or $\top$, which indicates a potential violation of memory safety. If a path from the initial state to $\top$ is found (a *counterexample* to memory safety), our algorithm translates this path into an arithmetic program ($\mathrm{Tr}(A)$). This arithmetic program is then analyzed to obtain strengthenings for the invariants discovered during shape analysis. The results of the arithmetic analysis are then combined with the shape analysis results to produce a more refined ATS ($\hat{A}$). A particular property of this combination is that if $\top$ can be shown to be unreachable in the arithmetic program, then the original program is memory safe.

Definition 1 *An **abstract transition system** is a tuple $(Q, L, \iota, \rightsquigarrow)$ where Q is a set of states, $\iota \in Q$ is the start state, and $L\colon Q \to S$ is a function that labels each state with a separation logic formula describing the memory configurations associated with that state (or $\top$). The last component, $\rightsquigarrow$ is a labeled transition relation. The labels are either program commands (c) or an empty label (ϵ). Thus, $\rightsquigarrow \subseteq Q \times (C \cup \{\epsilon\}) \times Q$. For convenience, if $t \in (C \cup \{\epsilon\})$ and $q, q' \in Q$, we will write $q \overset{t}{\rightsquigarrow} q'$ to abbreviate $(q, t, q') \in \rightsquigarrow$.*

We assume that quantified variables in the state labels are α-renamed to be disjoint from the set of variables present in the commands labeling the edges. We will refer to the edges labeled with commands as *postcondition edges* and the edges labeled with ϵ as *weakening edges*. The reason for these names can be seen in the following definition of well-formedness, which we require of our ATSs.

Definition 2 *An ATS $(Q, L, \iota, \rightsquigarrow)$ is **well-formed** iff for all $q, q' \in Q$ and $c \in C$, i) $q \overset{c}{\rightsquigarrow} q'$ implies that $\{L(q)\}c\{L(q')\}$ is a valid separation logic triple and ii) $q \overset{\epsilon}{\rightsquigarrow} q'$ implies $(L(q) \vdash L(q'))$ is a valid separation logic entailment.*

This ensures that the annotations associated with the abstract states are consistent with the commands labeling the edges. That is, if $q \overset{c}{\rightsquigarrow} q'$ and c terminates when executed from a state satisfying $L(q)$, then it terminates in a state satisfying $L(q')$. Well-formedness also ensures that the weakening edges are valid entailments. The algorithms defined in [21] and [13] automatically construct an abstract transition system that satisfies this condition.

In order to focus on the specifics of generating arithmetic programs from counterexamples, which is the main contribution of this paper, we assume that the abstract transition system has already been generated by running a separation logic based shape analysis on the input program. The interested reader can refer to [21] and [13] for details on how the ATS is generated.

An example of the abstract transition system that the shape analysis might generate is given in Figure 2. This ATS corresponds to the program discussed in Section 2. Dotted lines are used for weakening edges, while solid lines denote postcondition edges. We abbreviate `assume`(e) as $a(e)$. Note that the shape analysis has discovered an invariant for the loop at control location 4, indicated by the cycle at the bottom of the second column of states.

At control location 15, the system splits based on the value of k, the length of the list. This is the one non-standard modification we make in our separation logic shape analysis. Such an analysis would ordinarily try to execute `curr := [curr.next]` at location 15 given the precondition $\exists k.\ ls^k(t, NULL)$. Since this precondition does not imply that the command is memory safe (the list could be empty), the analysis would simply conclude $\top$. Instead, our shape analysis will check to see if there is some condition under which the memory access would be safe. More precisely, our theorem prover internally performs case splits and if one of these cases results in safe execution, it returns this condition to the analysis. The analysis then splits based on this condition and continues exploring the safe branch (the unsafe branch remains labeled with $\top$). For our definition of lists, this condition is always a check on the length of the list. This is a key component of our technique as it makes explicit the way in which size information about data structures affects the safety of the program. It will then be the job of the arithmetic analysis tool to show that the unsafe branch is infeasible due to arithmetic constraints among the variables.

4.1 Generating Arithmetic Programs

The arithmetic program is generated by converting edges in the ATS to commands that do not reference the heap. This translation involves making use of the information about heap cells that the shape analysis has provided. For example, given the state $x \mapsto [data\colon y+2]$, we know that the command `z = [x.data]` will result in z containing the value $y + 2$. We can achieve the same effect with the command `z = y + 2`, which does not reference the heap but instead exploits the fact that the shape analysis has determined the symbolic value for the contents of the data field of x.

The fact that our formulas can involve existential quantifiers makes the combination more expressive, and the translation more involved. Given the formula

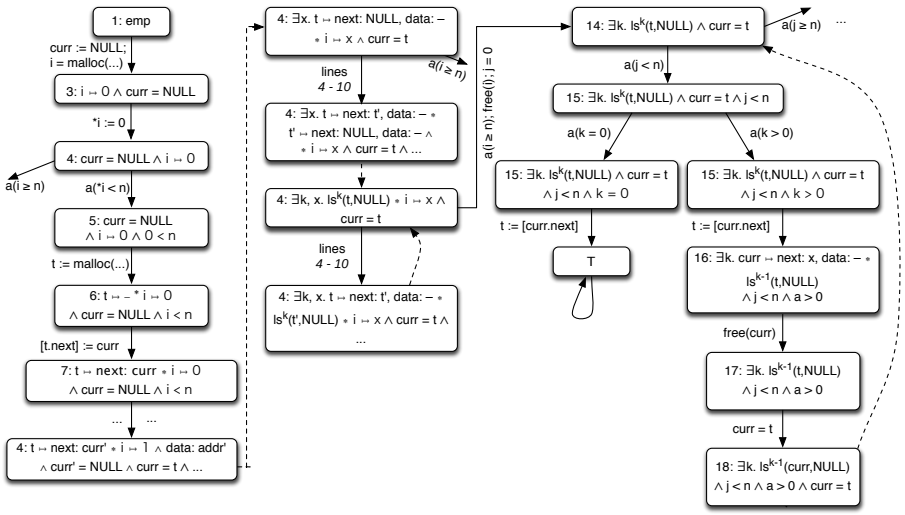

Fig. 2. Sample ATS after shape analysis.

$\exists y.\ x \mapsto [data\colon y + 2]$, it is clearly no longer sound to replace the command `z = x->data` with the command `z = y + 2`. Since y is not a program variable, its value is not specified from the point of view of the arithmetic analysis tool. We must therefore ensure that the arithmetic program we generate contains a variable y, corresponding to the quantified variable in the formula and that in executions of the arithmetic program, y's value is constrained in such a way that it satisfies the separation logic formulas we received from the shape analysis.

We can formalize this idea of using the arithmetic commands to enable reasoning about quantified variables with the following definition, which describes the properties the arithmetic command (c' in the definition) must have.

Definition 3 *Let $q \overset{c}{\rightsquigarrow} q'$ be an edge in an ATS $(Q, L, \iota, \rightsquigarrow)$. Let $L(q) = \exists \vec{x}.\ P$ and $L(q') = \exists \vec{y}.\ Q$ be abstract state formulas. A command c' is a **quantifier-free approximation** (QFA) of the edge $q \overset{c}{\rightsquigarrow} q'$ iff for any pure formulas P' and Q', the triple $\{P'\}\ c'\ \{Q'\}$ implies the triple $\{\exists \vec{x}.\ P \wedge P'\}\ c\ \{\exists \vec{y}.\ Q \wedge Q'\}$.*

That is, reasoning using c' is an over-approximation of reasoning under the quantifier in the pre- and postconditions of c. In Section 5 on soundness, we show that such reasoning can be extended to the whole program by replacing each command in the original ATS with a quantifier-free approximation of that command and reasoning about the ATS thus obtained.

Translating Postcondition Edges To find a purely arithmetic QFA for each of the heap-manipulating commands, let us first look at the rules that are used for adding postcondition edges to the ATS. These are given in the left column of

Shape Analysis Postcondition Rule			Arith. Cmnd.
$\{\exists \vec{z}.\ S\}$	$x{:=}E$	$\{\exists x', \vec{z}.\ x = E[x'/x] \wedge S[x'/x]\}$	$x'{:=}x;$ $x{:=}E[x'/x]$
$\{\exists \vec{z}.\ S\}$	$x{:=}?$	$\{\exists x', \vec{z}.\ S[x'/x]\}$	$x'{:=}x;\ x{:=}?$
$\{\exists \vec{z}.\ S\}$	$x{:=}\mathrm{alloc}()$	$\{\exists x', \vec{z}.\ S[x'/x] * (x \mapsto [\,])\}$	$x'{:=}x;\ x{:=}?$
$\{\exists \vec{z}.\ S * (E \mapsto [\rho, t\colon F])\}$	$x{:=}[E.t]$	$\{\exists x', \vec{z}.\ x = F[x'/x] \wedge (S * (E \mapsto [\rho, t\colon F]))[x'/x]\}$	$x'{:=}x;$ $x{:=}F[x'/x]$
$\{\exists \vec{z}.\ S * (E \mapsto [\rho])\}$	$\mathrm{free}(E)$	$\{\exists \vec{z}.\ S\}$	ϵ
$\{\exists \vec{z}.\ S * (E \mapsto [\rho, t\colon G])\}$	$[E]{:=}F$	$\{\exists \vec{z}.\ S * (E \mapsto [\rho, t\colon F])\}$	ϵ
$\{\exists \vec{z}.\ S\}$	$\mathrm{assume}(P)$	$\{\exists \vec{z}.\ S \wedge P\}$	$\mathrm{assume}(P)$

Fig. 3. Rules for generating arithmetic commands from abstract postcondition edges.

Figure 3. They are presented as Hoare triples where the pre- and postconditions are abstract state formulas. We use the notation $S[x'/x]$ to mean S with x' substituted for x.

Note that the first four rules result in the abstract post-state having one more quantifier than the abstract pre-state: they each have the form $\{\exists \vec{z}.\ S\}\ c\ \{\exists x, \vec{z}.\ S'\}$. Our goal is to find an arithmetic command c' corresponding to the original command c, and to use c' to reason about c. As such, we would like c' to contain the new quantified variable. To do this in a way such that c' is a QFA, we need c' to record the witness for the existential in the postcondition. As an example, consider the command for assignment.

$$\{\exists \vec{z}.\ S\}\ x{:=}E\ \{\exists x', \vec{z}.\ x = E[x'/x] \wedge S[x'/x]\}$$

The variable x' in the postcondition represents the old value of x. Thus, the value of x before the assignment is the witness for x' in the postcondition. We can record this fact using the sequence of commands $x'{:=}x;\ x{:=}E$. We use the same idea to handle the other two rules that add a quantifier.

Capturing the quantification in the new command is only part of the process. We must also over-approximate the effect of the command on the program variables. For commands like allocation ($x{:=}\mathrm{alloc}()$), the best we can do is replace this with the nondeterministic assignment $x{:=}?$. However, for lookup we can use the technique mentioned at the beginning of this section: if the precondition tells us that the t field of cell E contains the value F, we can replace $x{:=}[E.t]$ with $x{:=}F$ (and the precondition for lookup will always have this form).

The other heap commands (heap store and free) are replaced with no-ops. This may be surprising since these commands can have indirect effects on the values of integer variables in the abstract state formulas. Values stored in the heap can later be loaded into variables. This case is already handled by our rule for lookup, as can be seen by considering what happens when we translate the command sequence `[x.data] := y + 3; z := [x.data]` to arithmetic commands. The first command will be converted to a no-op. To translate the second command, we need to know its precondition. Supposing we start from the state $x \mapsto [\,]$, the postcondition of the first command is $x \mapsto [data\colon y + 3]$. This means that the translation will convert the second command to $z{:=}y + 3$, which has

the same effect on the program variable z as the original commands. So indirect updates to program variables through the heap will be properly tracked.

Also, freeing memory cells can decrease the size of lists in the heap. To incorporate reasoning about the length of lists, we must talk about how we translate weakening edges in the ATS.

Translating Weakening Edges Weakening edges are added by the shape analysis to the abstract transition system for two reasons. First, they are used to rewrite abstract states into a form to which we can apply one of the postcondition rules. For example, to execute `x := [a.next]` from the state

$$\exists k.\ ls^k(a, NULL) \wedge a \neq NULL$$

we must first notice that this formula implies

$$\exists y, k.\ a \mapsto [next\colon y] * ls^k(y, NULL) \wedge a \neq NULL$$

We can then apply the third postcondition rule to this state to get

$$\exists y, k.\ a \mapsto [next\colon y] * ls^k(y, NULL) \wedge a \neq NULL \wedge x = y$$

The other use of weakening edges is to show that certain formulas are invariant over executions of a loop. For example, suppose we start in a state

$$\exists k.\ ls^k(a, NULL)$$

And after executing some commands, reach the state

$$\exists x, k.\ a \mapsto [next\colon x] * ls^k(x, NULL)$$

If both these states are associated with the same program location, then we have found a loop invariant since the second formula implies the first. This fact is recorded in the ATS by connecting the second state to the first with a weakening edge.

In both cases, we need to record information about the quantified variables so that our arithmetic analysis can discover arithmetic relationships involving these quantified variables. As with postcondition edges, we do this by recording the witnesses for the quantified variables.

Recall that we have a weakening edge in the ATS only if $\exists \vec{x}.\ P \vdash \exists \vec{y}.\ Q$. Our goal then is to find an arithmetic command c' such that for any P', Q', if $\{P'\}c'\{Q'\}$ then $\exists \vec{x}.\ P \wedge P' \vdash \exists \vec{y}.\ Q \wedge Q'$. We generate such a c' by analyzing the proof of entailment between $\exists \vec{x}.\ P$ and $\exists \vec{y}.\ Q$. As we are interested in tracking the values of existentially quantified variables, it is the rules for existential quantifiers that end up being important for generation of the arithmetic commands. In Figure 4 we present the standard rules for introduction and elimination of existential quantifiers, modified to produce the appropriate arithmetic commands. The full details of entailment for our fragment of separation logic are omitted for space reasons, but the system is similar to that described in [3].

$$\textsc{E-Elim}\quad \frac{\exists \vec{y}.\ P[a/x] \vdash \exists \vec{z}.\ Q \quad \langle c' \rangle}{\exists x, \vec{y}.\ P \vdash \exists \vec{z}.\ Q \quad \langle a{:=}\, x\,;\ c'\,;\ a{:=}\,? \rangle} \qquad \textsc{E-Intro}\quad \frac{\exists \vec{y}.\ P \vdash \exists \vec{z}.\ Q[t/x] \quad \langle c' \rangle}{\exists \vec{y}.\ P \vdash \exists x, \vec{z}.\ Q \quad \langle x{:=}\, t\,;\ c' \rangle}$$

Fig. 4. Rules for generating arithmetic commands from proofs for weakening edges.

The notation $P \vdash Q\langle c \rangle$ is used to mean that P entails Q and c is the arithmetic command that is a quantifier-free approximation of this entailment. For existential elimination, we simply record the new constant that was introduced for reasoning about the quantified variable on the left. We also nondeterministically assign to the constant once we are done with it to ensure that it will not appear free in any invariants the arithmetic tool produces. For existential introduction, we record the witness used to establish the existential formula on the right. We do this by having our entailment checker return a witness in addition to returning a yes/no answer to the entailment question. This is possible because the entailment procedure sometimes proves existentials constructively. When entailment is proved without finding a witness (*e.g.* as happens when unrolling an inductive definition with a quantified body), t in the premise is a fresh logical constant, and so $x{:=}\,t$ is equivalent to $x{:=}\,?$.

As an example, suppose we want to generate arithmetic commands that model the entailment $\exists k.\ ls^k(a, NULL) \wedge a \neq NULL \vdash \exists x, k.\ a \mapsto [next\colon x] * ls^k(x, NULL) \wedge a \neq NULL$. We first introduce a new constant b for the existential on the left, resulting in the formula $ls^b(a, NULL) \wedge a \neq NULL$ and the arithmetic command $b{:=}\,k$. We then unroll the list segment predicate according to the definition, obtaining $\exists x.\ a \mapsto [next\colon x] * ls^{b-1}(x, NULL) \wedge a \neq NULL$. Since x arises due to the expansion of a definition, we use nondeterministic assignment in the generated command producing $x{:=}\,?$. We then apply existential elimination again, obtaining $a \mapsto [next\colon c] * ls^b(c, NULL) \wedge a \neq NULL$ and $c{:=}\,x$. Finally, we prove the formula on the right side of the entailment, obtaining witnesses for the existentially quantified variables x (witness is c) and k (witness is $b-1$). We then "forget" about the constants we added with the commands $b{:=}\,?\,;\ c{:=}\,?$. Thus, the full sequence of commands for this entailment is

```
b := k; x := ?; c := x; k := b - 1; b := ?; c := ?
```

The updates to k here reflect the fact that, at this point in the execution, the length of the list predicate being tracked by the shape analysis has decreased in size by 1. Due to the commands `b := ?` and `c := ?`, any quantifier free invariant that holds after executing this sequence of commands will be expressed without reference to b and c.

4.2 Precision

We can get a sense for the precision of this analysis by examining the places in which nondeterministic assignment is used to over-approximate a command.

One such place is the rule for allocation. This should not concern us as the goal is to use these arithmetic programs to discover properties of the integer values involved in the program, whereas allocation returns a pointer value, which the shape analysis is already capable of reasoning about. We can use this observation to optimize our approach. If we keep track of type information we can ensure that we only generate arithmetic commands when those commands result in the update of integer-valued variables.

The other place where nondeterministic assignment occurs is in the rule for existential elimination when the entailment checker does not return a witness. This is actually the source of all imprecision in the arithmetic translation. It can happen that an integer value such as 3 is stored in a list element, resulting in the state

$$\exists k, d.\ x \mapsto [data\colon 3, next\colon k] * ls^d(k, NULL)$$

If we then abstract this state to $\exists d.\ ls^d(x, NULL)$, we lose the information about the value stored in the data field of the heap cell at x. If this field is accessed again, it will be assigned a nondeterministic value by the shape analysis. To remedy this would require a notion of refinement on the shape analysis side of the procedure. And indeed our technique would interact well with such a shape refinement system. One could interleave arithmetic refinement and shape refinement, calling one when the other fails to disprove a counterexample. We leave development of such a system for future work.

4.3 Combined Analysis

Using the translation of individual edges described above, we can define the translation of ATSs:

Definition 4 (Translated arithmetic program) *For an ATS* $A = (Q, L, i, \rightsquigarrow)$*, the* translated arithmetic program $\mathrm{Tr}(A) = (Q, L', i, \rightsquigarrow')$ *is an ATS defined such that if* $q \overset{c}{\rightsquigarrow} q'$ *and* c' *is the arithmetic command associated with this edge, then we have* $q \overset{c'}{\rightsquigarrow}' q'$*.*

Finally, the results of the combined analysis are given by:

Definition 5 (Combination) *Given an ATS* $A = (Q, L, \iota, \rightsquigarrow)$ *and its well-formed translation* $\mathrm{Tr}(A) = (Q, L', \iota, \rightsquigarrow')$*, where* $L'(q)$ *is a pure formula for each* q*, the* combination *of* A *and* $\mathrm{Tr}(A)$ *is defined to be the ATS* $\hat{A} = (Q, \hat{L}, \iota, \rightsquigarrow)$ *where if* $L(q) = \exists \vec{z}.\ S$ *and* $L'(q) = S'$ *then* $\hat{L}(q) = \exists \vec{z}.\ S \wedge S'$*.*

Note that **false** $\wedge \top$ is equivalent to **false**. So for an abstract state where the shape analysis obtained $\top$, indicating a potential safety violation, if an arithmetic analysis can prove the state is unreachable (has invariant **false**), then it is also unreachable in the combined analysis.

5 Soundness

The soundness result hinges on the fact that the translation for commands defined in Section 4 results in a quantifier-free approximation.

Theorem 1 *For each postcondition rule in Figure 3 the associated arithmetic command is a quantifier-free approximation of the original command.*

Theorem 2 *If* $\exists \vec{x}.\ P \vdash \exists \vec{y}.\ Q\ \ \langle c \rangle$ *and* $\{P'\}\ c\ \{Q'\}$ *then* $\exists \vec{x}.\ P \wedge P' \vdash \exists \vec{y}.\ Q \wedge Q'$.

Given this we can show that results based on analyzing the arithmetic program can be soundly conjoined to the formulas labeling states in the ATS.

Theorem 3 (Soundness) *For an ATS A, suppose that we have run an arithmetic analysis on* $\mathrm{Tr}(A)$ *and obtained (pure) invariants at each program point. Then* $\hat{A}$ *is well-formed.*

6 Experimental Results

We have developed a preliminary implementation of our analysis and tested it on a number of programs where memory safety depends on relationships between the lengths of the lists involved. For example, a function may depend on the fact that the result of filtering a list has length less than or equal to that of the original list. As arithmetic back-ends we have used OctAnal [22], Blast [19], and ARMC [23]. Preliminary results show two trends. First, there is no tool among those we tried that is strictly stronger than the others. That is, there is no tool among these three that is able to prove memory safety for all of our sample programs. However each program was able to be proven by some tool. In such cases, the ability to choose any arithmetic tool allows one to prove the greatest number of programs. Secondly, the performance characteristics of the tools are highly dependent on the type of input they are given. As our examples are all relatively small, OctAnal outperformed the tools based on model checking. However, for large programs that contain many arithmetic commands which are not relevant to proving memory safety, we would expect the relative performance of model checking tools to improve, as these tools only consider the variables needed to prove the property of interest. More experiments are necessary to fully explore the advantages and disadvantages of various arithmetic provers in the context of our combination procedure.

7 Related Work

The work presented here describes a way of lazily combining two abstract interpreters: the shape analysis produces abstracted versions of the input program for which an arithmetic analysis is then called. More eager combination approaches have been previously discussed in the literature (*e.g.* [11, 17, 18]).

Recent work [6] has described a method in which the TVLA [26] shape analysis is lazily combined with an arithmetic analysis based on Blast. This work reverses the strategy that we propose: they are lazily providing some additional spatial support for what is primarily an arithmetic analysis, whereas we are lazily providing additional arithmetic support for a shape analysis. Which approach is better depends on the program in question. Programs that are concerned primarily with integer calculations, but occasionally use a heap data structure may be better analyzed with the approach in [6]. Programs which have as their main function manipulation of heap data, or for which memory safety must be verified, would be better analyzed with our approach.

Another related approach is the shape analysis in [21], which uses predicate abstraction to retain facts about integer values during widening, but does not provide a predicate inference scheme. Thus, these predicates must be supplied by the user. Since our method uses a separate arithmetic tool to perform the refinement, we inherit any predicate inference that tool may perform.

Connections between shape and arithmetic reasoning are exploited throughout the literature (*e.g.* [1, 15, 10, 8, 28, 14]). Also, people have looked at ways of combining abstract interpreters over different domains [11, 17, 18]. For example, one could imagine combining the shape analysis in [21] or [13] with an abstract interpretation over the domains of convex polyhedra [12] or octagons [22]. Our approach has the advantage of allowing the use of any of these abstract domains as well as arithmetic analyses that are not based on abstract interpretation. Furthermore, given the way in which information about quantified values is shared between the analyses, it is not clear that our approach can be seen as an instance of one of the standard constructions for combinations of abstract domains.

Other shape analyses are known to support arithmetic reasoning, but typically in only very limited ways that allow them to use naive arithmetic widening steps. For example, the shape analysis described in [4] provides a combined analysis that maintains arithmetic information. In this case the set of arithmetic variables in the abstract domain is extremely limited: each list-segment in the shape analysis invariant is associated with an arithmetic variable. Furthermore, only one inequality per variable is allowed, as the inequalties only occur between a variable and its "old version". Given these restrictions, the widening operation in [4] can be naive in terms of its handling of arithmetic. Our refinement-based procedure uses arbitrary arithmetic analysis tools to strengthen the shape analysis invariant being inferred, meaning that we have access to the most sophisticated widening operations available. More arithmetic is supported in [9], but also with an aggressive widening since the arithmetic reasoning is targeted to within a loop body.

Another combination of shape and arithmetic is given in [25], which presents a means of reasoning about size properties of data structures tracked via a shape analysis based on reference counting and must-alias information.

A number of approaches based on combining a numerical analysis with a shape analysis based on shape graphs (such as [26]) have been explored. Examples include [16] and [27]. However ours is the first attempt to carry out

such general arithmetic reasoning in a shape analysis where the abstract domain consists of separation logic formulas.

Our method makes use of a notion of *generalized path* (*i.e.* a path through the program where the number of unrollings through some loops are unspecified). Uses of this concept can be found elsewhere in the literature (*e.g.* [20, 5]). In particular, our work can be seen as fitting nicely into the framework proposed in [5]. As in this work, we use a refinement procedure based upon analyzing generalized paths. However, our work is unique in that the paths arise due to a shape analysis based on abstract interpretation rather than a software model checker. Furthermore, the way in which quantifiers in the generalized path are expressed as variables in the translated path is not present in this other work.

8 Conclusion

Shape analyses are typically imprecise in their support for numerical reasoning. While an analysis that fully tracks correlations between shape and arithmetic information would typically be overkill, we often need a small amount of arithmetic information in shape analysis when arithmetic and spatial invariants interact. We have proposed a lazy method of combining a fixed shape analysis with an arbitrary arithmetic analysis. This method treats shape and arithmetic information independently except for key relationships identified by the shape analysis. Crucially, these relationships may be over values which are only present in the abstract states. When potentially spurious counterexamples are reported by our shape analysis, our method constructs a purely arithmetic program and uses available invariant inference engines as a form of refinement. This new adaptive analysis is useful when a proof of memory safety or `assert`-validity requires deep spatial reasoning with targeted arithmetic support.

References

1. A. Armando, M. Benerecetti, and J. Mantovani. Model checking linear programs with arrays. *Electr. Notes Theor. Comput. Sci.*, 144(3):79–94, 2006.
2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'2001: Programming Language Design and Implementation*, volume 36, pages 203–213. ACM Press, 2001.
3. J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.
4. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV'06*, 2006.
5. D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Path invarints. In *PLDI*, 2007.
6. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *CAV'2006: Computer Aided Verification*, LNCS 4144, pages 532–546, 2006.
7. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI'2003: Programming Language Design and Implementation*, pages 196–207. ACM Press, 2003.

8. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV'06*, LNCS. Springer, 2006.
9. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS'2006: Static Analysis Symposium*, 2006.
10. Y. Choi, S. Rayadurgam, and M. P. Heimdahl. Automatic abstraction for model checking software systems with interrelated numeric constraints. In *Proc. of ESEC/FSE*, pages 164–174. ACM Press, 2001.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'1979: Principles of Programming Languages*, pages 269–282. ACM Press, 1979.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.
13. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
14. N. Dor, M. Rodeh, and M. Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, New York, NY, USA, 2003. ACM Press.
15. C. Flanagan. Software model checking via iterative abstraction refinement of constraint logic queries. CP+CV'04, 2004.
16. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *TACAS*, 2004.
17. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In P. Sestoft, editor, *European Symp. on Programming, ESOP 2006*, volume 3924 of *LNCS*, pages 279–293, 2006.
18. S. Gulwani and A. Tiwari. Combining abstract interpreters. In T. Ball, editor, *ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2006*, 2006.
19. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL'2002: Principles of Programming Languages*, pages 58–70. ACM Press, 2002.
20. D. Kroening and G. Weissenbacher. Counterexamples with loops for predicate abstraction. In *CAV'2006: Computer Aided Verification*, LNCS 4144, 2006.
21. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *SPACE 2006: Third Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2006.
22. A. Miné. The Octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006. (to appear).
23. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
24. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
25. R. Rugina. Quantitative shape analysis. In *SAS*, 2004.
26. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *TOPLAS*, 2002.
27. T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *SAS*, 2002.
28. T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for queues with integer constraints. In R. Ramanujam and S. Sen, editors, *FSTTCS*, volume 3821 of *Lecture Notes in Computer Science*, pages 225–237. Springer, 2005.