Static Analysis with Separation Logic  
(Thesis Proposal)

Stephen Magill  
July 16, 2007

Abstract
In this proposal I claim that: 1) Separation logic provides a good foundation for an automatic shape analysis. 2) The connection separation logic provides between the concrete and abstract representations of the contents of a data structure can be leveraged together with suitable abstract domains to describe strong data structure invariants. I propose new research in both of these areas, as well as in entailment checking for separation logic, which is necessary to implement these static analyses.

1 Introduction
Automated program verification is a large and active field, with substantial research devoted to static analysis tools. However, these tools are based mostly on classical logic. This places a large burden on the verification procedure in the practical case of programs involving pointers. Because classical logic contains no primitives for expressing non-aliasing, all aliasing patterns must be considered when doing the program analysis. Any such program analysis must also take into account the global context, since any two pointer variables, regardless of scope, are potential aliases. Computing weakest preconditions and strongest postconditions then becomes exponential in the number of program variables. This burden can be lessened somewhat by utilizing a pointer analysis to rule out certain cases, but the results are often unsatisfactorily weak, particularly when allocation and deallocation are involved.

Contrast this with separation logic [30], a program logic with connectives for expressing aliasing patterns which provides concise weakest preconditions even for pointer operations. Separation logic also supports compositional reasoning. As such, it seems a promising foundation upon which to build static analysis methods and tools.

The most common type of automated analysis for data structures is a shape analysis. Shape analysis is concerned with reasoning about the internal invariants of pointer structures. For example, in the case of doubly-linked lists, a shape analysis would check the fact that if the forward link of memory cell \( a \) points to cell \( b \), then the back link of cell \( b \) points to cell \( a \). Shape properties also encompass heap reachability properties. Continuing with the example of linked lists, we would want to track whether the list is null-terminated. That is, whether a cell holding the value \( \text{null} \) is reachable from the head of the list by following “next” pointers.

In [27], we presented a shape analysis based on separation logic that leveraged the implicit non-aliasing information provided by the separating conjunction to provide support for local reasoning and strong updates of heap cells. The approach is based on defining symbolic execution rules which give post-conditions of a restricted form for each program command. An abstraction function is then used to weaken the symbolic description of the program state after each iteration through a loop. The abstraction is designed such that after a finite number of iterations, the set of reachable abstract states associated with the loop covers all possible executions. That is, after a finite number of iterations, we are guaranteed to find a loop invariant.

We then explored the case where safety of the program depends on properties other than just the shape of data structures. In [26], we present a system for reasoning about both the shapes and sizes of data structures. This allows us to prove memory safety of programs when such safety depends on both shape information and integer invariants. This can be seen as providing a more refined view of the data structure.
the analysis in [27] reveals only the shape of the structure, the analysis in [26] also supports reasoning about its size.

As my thesis work, I propose to extend this approach to allow reasoning about even more precise properties of data structures, for example sortedness and other ordering properties. I aim to formalize this approach to program verification and provide a general framework for combining shape and data analyses, of which the analyses in [27] and [26] would be specific instances.

1.1 Automated Versus Manual Verification

One might wonder as to the utility of pursuing automated analyses rather than focusing on a system in which invariants are provided by the programmer and simply checked by the machine. After all, the programmer must be aware of these invariants if he is writing correct code. And indeed, any verification system needs some input from the programmer. The programmer must at least specify which properties are to be checked. Our goal with this work is to strike a balance between what must be provided by the programmer and what can be automatically inferred. The programmer should be allowed (and in fact required) to specify the key data structure invariants, while enjoying a level of automatic support that frees him from having to annotate every function and loop.

Consider the case of writing a set of functions that implement various operations on a singly-linked list. We will write \( ls(a, b) \) to represent a linked list where the first element is stored at \( a \) and the next pointer of the last element contains \( b \). Almost every function will involve traversing the list, requiring a loop invariant of the form \( lsshd, curr * ls(curr, 0) \), saying that there is a list segment starting at the head of the list and ending at the current position and a segment starting at the current position that terminates with NULL (written “\( \emptyset \)”). Depending on the particular loop, the variable names may be different and a pointer to the previous node may be maintained. If the first node is treated specially, or the list manipulation depends on data values in the list, the invariant may involve several disjuncts. To require the programmer to write and maintain such invariants, while almost certainly helpful, adds substantially to the cost of the verification effort. Furthermore, these invariants must be altered whenever changes to the code affect the set of reachable states, so this cost persists through the lifetime of the code. Instead, we aim to have the programmer focus on the reasoning behind the correctness of the code. That is, while a list traversal may be safe because of the stated loop invariant, the reason this formula is invariant has to do with simple properties of the linked list structure. For example, that a state in which one has established \( ls(hd, x) * ls(x, \emptyset) \), implies that \( ls(hd, \emptyset) \) also holds of this state. It is these sorts of conditions, that specify how a data structure may be extended while still maintaining its internal invariants, that must be thoroughly understood in order to use the structure correctly.

I propose a system which requires the programmer to provide the data structure axioms that are used by the theorem prover and the abstraction framework. The system allows the programmer to write annotations, but does not require the program to be fully annotated, thus striking a balance between the ability of the programmer to specify what is expected and the ability of the system to fill in the details of the proof.

2 Outline

This proposal is concerned with the development of both the theory and tools required to support automated verification of programs that manipulate the heap. This verification process can be viewed as consisting of four main components.

1. An abstract domain of separation logic formulas with widening operations to guarantee convergence. This enables the automatic computation of invariants at each control point in the program and, in many cases, is sufficient to prove memory and assertion safety. We have formalized and implemented such a system for doubly-linked list structures. Future work includes extending this to tree-shaped structures. As I will show in Section 5, there are several possible shape predicates for trees. It will require further investigation to determine if there is a most general tree predicate and what the efficiency versus precision tradeoffs are for the various predicates. Work will focus on predicates which provide an abstract representation of the data involved to enable the combination that is the goal of item 3.
2. An entailment procedure for formulas in the abstract domain just described. This is necessary to detect when the invariant computation described above has reached a fixed point, as well as to check the validity of programmer-supplied assertions. The ideal would be to work with an abstract domain for which entailment is decidable. However, should the theory identified in item 1 turn out to be undecidable, we will have to accept an incomplete entailment procedure and rely on experimental results to show that the prover is strong enough to be useful in practice.

3. Combination of the shape analysis developed in item 1 with static analyses over other domains. The separation logic predicates we use for the shape analysis can be seen as providing a connection between the concrete representation of a data structure in terms of pointers and the abstract representation of its contents as a mathematical object. In [26], we showed how to combine a shape analysis involving list predicates with an arithmetic analysis. The shape predicate in this case exposed the length of the data structure to the arithmetic analysis engine. The advantage of a combination method such as the one we defined is that it separates the analysis of the abstract information (in this case, the length) from the shape analysis used to derive that information. I intend to extend this approach to reasoning about the full contents of data structures. In Section 3.2 I show how a sortedness analysis can be combined with shape analysis over various concrete representations to reason about sortedness of a variety of data structures. My goal is to use the experience gained in investigating these specific combinations to develop a general method for combining separation logic based shape analysis with analyses over other domains.

4. An implementation of the above. I have already implemented the shape analysis and an incomplete entailment procedure for doubly-linked lists, as well as the combination of this with integer program analyses. I intend to extend this implementation as my work progresses.

Section 3 presents a more thorough overview of the proposed research along with motivating examples. Section 4 presents a formal problem statement and revisits an example from Section 3 in this more formal setting. Section 5 discusses remaining work on the shape analysis side and Section 6 discusses related work. Section 7 concludes and Section 8 gives with a timeline for the research.

3 From Concrete to Abstract Views of Data

In this section I discuss how inductively-defined separation logic predicates can be used to establish a connection between the concrete representation of a pointer-based data structure and an abstract view of the data contained in that structure. I then present examples of various abstract domains which track approximations of the data involved. The abstract domains I will discuss include 1) a size domain which tracks properties such as the length of lists and the number of nodes in a tree, 2) a sortedness domain, which tracks sortedness of the data present in a list or tree, and 3) a domain based on regular tree grammars, that can be used to track invariants of various balanced tree data structures. The size domain derives from a recent paper with myself, Berdine, Clarke and Cook [26], which looked at the problem of reasoning about lengths of list structures. The domain based on regular tree grammars is inspired by the work of Freeman, Pfenning, Davies, and Dunfield on refinement types [18, 15, 17].

Concrete Representations Suppose we want to represent a sequence of integers. There are many possible concrete representations of such a structure, each appropriate in different circumstances. We consider here three concrete representations, given as inductively defined separation logic predicates. We use \(a \cdot b\) to indicate the concatenation of the sequences \(a\) and \(b\) and \(\epsilon\) to denote the empty sequence. We use \(\emptyset\) to represent the null pointer value.

First, we have a singly-linked list. The definition is given below. Note that this predicate establishes a connection between the concrete pointer structure contained in the heap and the abstract sequence, here denoted by \(A\).

\[
\text{ls } A \ (a, b) \ \overset{\text{def}}{=} \ (A = \epsilon \land a = b \land \text{emp}) \\
\lor \ (\exists x, A'. \ A = x \cdot A' \land \exists k. \ a \mapsto [\text{val} : x, \text{next} : k] \ast \text{ls } A' \ (k, b))
\]
We can represent the same sequence using a doubly-linked list.

\[
dll \ A (a, b, c, d) \overset{\text{def}}{=} (A = \epsilon \land a = c \land b = d \land \text{emp}) \\
\lor (\exists x, A'. A = x \cdot A' \land \exists k. b \mapsto [\text{val} : x, \text{next} : k, p : a] \ast dll \ A'(b, k, c, d))
\]

Finally, we present a tree-based predicate. The sequence represented by the tree is the sequence of values encountered during a depth-first, in-order traversal. There are domains other than sequences that would make sense here as well. We present a tree predicate with a more exact characterization of the structure of the data in Section 3.4.

\[
tree \ A a \overset{\text{def}}{=} (A = \epsilon \land a = 0 \land \text{emp}) \\
\lor (\exists x, A_1, A_2. A = A_1 \cdot x \cdot A_2 \land \exists k_1, k_2. a \mapsto [\text{val} : x, \text{left} : k_1, \text{right} : k_2] \\
\ast tree \ A_1 k_1 \ast tree \ A_2 k_2)
\]

**Abstract Representations** We now present some abstract domains that are useful in the analysis of list programs. In all cases, these are domains of pure values. That is, the formulas (both abstract and concrete) can be interpreted without reference to the heap. I will first re-examine some previous work of ours in the context of abstract interpretations over data. I then present the example of sortedness in detail, demonstrating how a single sortedness domain can be combined with multiple shape domains. I will then present other potentially useful abstract domains.

### 3.1 Combining Shape and Arithmetic

In [27], we showed how to prove memory safety by using shape predicates similar to those we presented above for lists. In this work, we did not track any properties of the abstract data. Thus, the invariants we obtained were of the form

\[
\exists A. \ ls \ A (a, b) \ast S
\]

where the actual list data is always existentially quantified and does not appear free in the rest of the abstract state \(S\) (in actuality, we simply omitted the argument \(A\) from the list predicate and developed the theory without reference to the contents of the list). In [26], we took advantage of the connection between the abstract data and the shape information that we get from our shape analysis. This work can be seen as inferring invariants of the form given below (we write \#(A) to denote the length of sequence \(A\))

\[
\exists A, k. \ ls \ A (a, b) \land k = \#(A) \ast S
\]

where \(k\) (but not \(A\)) is allowed to appear free in \(S\). The actual constraints on \(k\) are obtained via translating heap manipulating programs into purely stack-based integer programs. This was done to enable the use of other invariant inference schemes in addition to abstract interpretations. For example, it allows approaches based on interpolation [24] or those based on counter automata [1]. Let us consider an example. The code below builds a list of length \(n\) and then traverses it. We use brackets to indicate dereference.

```plaintext
1: curr = NULL;
2: i = 0;
3: while(i < n) {
4:   t = alloc();
5:   [t.next] := curr;
6:   curr = t;
7:   i = i + 1;
8: }
9: j = 0;
10: while(j < n) {
11:   curr = [curr.next];
12:   j = j + 1;
13: }
```
The loop invariant discovered by the shape analysis for the first loop is $\exists A. \text{ls} \ A \ (l, \text{curr})$. The corresponding invariant for the second loop is $\exists A, B. \text{ls} \ A \ (l, \text{curr}) \ast \text{ls} \ B \ (\text{curr}, \emptyset)$. The algorithm then uses the information from the shape analysis pass to generate an arithmetic program whose safety implies the safety of the original program. The arithmetic program that we would generate for this code, according to the algorithm given in [26], is an unrolling of the following.

```plaintext
1: int i = 0;
2: a = 0;
3: while(i < n) {
4:   a = a + 1;
5:   i = i + 1;
6: }
7: j = 0;
8: b = 0;
9: while(j < n) {
10:   if( a > 0 ) {
11:      a = a - 1;
12:      b = b + 1;
13:      j = j + 1;
14:   }
15: else abort;
16: }
```

Note that we have added two new variables $a$ and $b$ such that $a = \#(A)$ and $b = \#(B)$. The length information for the list segments involved in the iteration is thus made explicit and any integer invariant analysis can be run to discover relationships between the length variables and other program variables. Also, the dereference at line 11 in the original program, which is unsafe if $\#(A) = 0$, has been replaced with a branch on the length of this list. The unsafe case is marked with an `abort` command, the idea being that an integer invariance analysis can then be run on this program and, if it proves that `abort` is not reachable, this guarantees that the original program is memory safe (this is the statement of soundness from [26]). This technique of generating and then checking integer programs was used to accommodate the large variety of static analyses that have been developed for integer programs. The integer program can be checked using abstract interpretation [12], interpolation-based model checking [24], or algorithms for analyzing counter systems [1]. If we were to commit to static analysis techniques based on abstract interpretation, we could dispense with the translation to integer programs and simply apply the join operation and widening operator for the appropriate abstract domain. For example, in this case, after unrolling the first loop twice and using the abstraction function defined for the shape analysis, we would obtain the invariant

$$\exists A, i_0, i_1. \text{ls} \ A \ (\text{curr}, \emptyset) \land \#(A) = 2 \land i = i_1 + 1 \land i_1 = i_0 + 1 \land i_0 = 0$$

Abstracting to the polyhedra domain on $\#(A)$ and $i$ would then give

$$\exists A. \text{ls} \ A \ (\text{curr}, \emptyset) \land \#(A) = 2 \land i = 2$$

Taking the abstract postcondition again would yield

$$\exists A. \text{ls} \ A \ (\text{curr}, \emptyset) \land \#(A) = 3 \land i = 3$$

And the join of these in the polyhedra domain on $\#(A)$ and $i$ is the invariant

$$\exists A. \text{ls} \ A \ (\text{curr}, \emptyset) \land \#(A) = i$$

which is what we must conclude if the analysis is to discover that $i$ is tracking the length of the being built. The example code given above shows that memory safety can depend on the discovery of such length information. But even if more refined information is not necessary to prove memory safety, it may be necessary to prove properties beyond memory safety, such as showing that programmer-supplied assertions hold (which we call assertion safety). Such assertions can be used to check that datastructure invariants,
such as sortedness or balancing invariants, are being preserved. Failure to preserve such invariants may not result in the program aborting, but it will often lead to incorrect results.

We now look more closely at analyzing the abstract values produced by the shape analysis. We show that there are many abstract domains of interest and that the relationship between abstractions can be quite involved, demonstrating the need for a strong combination operation. At the same time, the tradeoff between efficiency and precision must be considered. A refinement process based on the analysis of abstract counterexamples may prove valuable in guiding the abstraction process.

### 3.2 Sortedness Domain

This domain tracks sortedness of sequences of integers. It is defined by the lattice 
\[ L = \{ \langle x, y \rangle \mid x, y \in \mathbb{Z} \land x \leq y \} \cup \{ \bot, \top \} \]
ordered by \( \sqsubseteq \) defined as follows

- \( a \sqsubseteq \top \) for all \( a \in L \)
- \( \bot \sqsubseteq a \) for all \( a \in L \)
- \( \langle x_1, y_1 \rangle \sqsubseteq \langle x_2, y_2 \rangle \) if \( x_2 \leq x_1 \) and \( y_1 \leq y_2 \)

The abstraction function \( \alpha \) is defined as follows. \( i \) is an integer and \( x \) and \( y \) are sequences.

\[
\begin{align*}
\alpha(\epsilon) &= \bot \\
\alpha(i) &= \langle i, i \rangle \\
\alpha(x \cdot y) &= \langle l_x, u_y \rangle \text{ if } \alpha(x) = \langle l_x, u_x \rangle \text{ and } \alpha(y) = \langle l_y, u_y \rangle \text{ and } u_x \leq l_y \\
\alpha(x \cdot y) &= \top \text{ otherwise}
\end{align*}
\]

And the concretization function is

\[
\begin{align*}
\gamma(\langle x, y \rangle) &= \{ x_0 \cdot x_1 \cdot \ldots \cdot x_n \mid x \leq x_0 \land x_n \leq y \land \forall 0 \leq i < n, x_i \leq x_{i+1} \} \\
\gamma(\bot) &= \{ \epsilon \} \\
\gamma(\top) &= \{ A \mid A \text{ is a sequence} \}
\end{align*}
\]

The lattice as defined includes infinite ascending chains (consider for example the sequence \( \langle 0, 0 \rangle, \langle 0, 1 \rangle, \ldots, \langle 0, n \rangle \)). To ensure convergence of the abstract interpretation, a widening operator can be used, such as that given in [14] for the interval domain.

#### 3.2.1 Sorted Lists

We can combine this with our shape analysis over lists in a manner similar to the combination with arithmetic presented above. We introduce a predicate \textit{sorted} \( A (x, y) \) which indicates that \( A \in \gamma(x, y) \). Such a combination allows us to prove that the insert routine in Figure 1 preserves sortedness of the input list. The routine is written in C and uses the following declarations for the list structure.

```c
typedef struct list {
    int val;
    struct list *next;
} *List;
```

Note that in the insert procedure, we pass a pointer to a List, which is itself already defined to be a pointer. That is, we pass a pointer to the head of the list instead of the passing the head directly. This is because the procedure must update the location of the head if it inserts an element before the current head of the list and is a consequence of C function arguments being passed by value.

In the following discussion, we will use typewriter font to refer to program variables and italics to refer to variables introduced during the reasoning (which we call logic variables). We assume that the precondition and postcondition of this procedure are both \( \exists A, x, y, list_0. \text{listp} \iff list_0 * is A (list_0, \emptyset) \land \text{sorted} A (x, y) \), so that at line 6 the invariant is

\[
\exists A, x, y. \text{listp} \iff \text{list} * is A (\text{list}, \emptyset) \land \text{sorted} A (x, y)
\]
1: void insert_list(List *listp, int v)
2: {
3:     List list = *listp;
4:     List new;
5:     /* empty list or beginning */
6:     if( list == NULL || list->val > v) {
7:         new = malloc(sizeof(list));
8:         new->next = list;
9:         new->val = v;
10:        *listp = new;
11:        return;
12:    }
13:    /* non-empty, middle */
14:    while( list != NULL ) {
15:        if( list->next == NULL || list->next->val > v ) {
16:            new = malloc(sizeof(list));
17:            new->val = v;
18:            new->next = list->next;
19:            list->next = new;
20:            break;
21:        }
22:        list = list->next;
23:    }
24: }
25: }

Figure 1: A routine to insert a new element into a sorted list.

At line 8, we have two cases (corresponding to the two conditions that can make us take this branch). The first, corresponding to the case where the list is empty, is

\[ \exists A, x, y. \text{listp} \mapsto \text{list} \ast \text{ls A (list, 0)} \land \text{sorted A (x, y)} \land \text{list} = \emptyset \]

which implies

\[ \exists x, y. \text{listp} \mapsto \text{list} \ast \text{emp} \land \text{list} = \emptyset \]

Executing lines 8-11 results in

\[ \exists x, y. \text{listp} \mapsto \text{new} \ast \text{new} \mapsto [\text{val : v}, \text{next : 0}] \]

which implies our postcondition. The second case (corresponding to a non-empty list) is, at line 8

\[ \exists A, A', x, y, a, b. \text{listp} \mapsto \text{list} \ast \text{list} \mapsto [\text{val : a}, \text{next : b}] \ast \text{ls A'} (b, 0) \land \text{sorted A'} (x, y) \land A = a \cdot A' \]

Taking postconditions for commands 8-11, we get

\[ \exists A, A', x, y, a, b. \text{listp} \mapsto \text{new} \ast \text{new} \mapsto [\text{val : v}, \text{next : list}] \ast \text{list} \mapsto [\text{val : a}, \text{next : b}] \]

\[ \ast \text{ls A'} (b, 0) \land \text{sorted A'} (x, y) \land A = a \cdot A' \land v \leq a \]

This implies the formula below, which captures all the inequalities and sortedness information we could derive for a and A’ given that A = a \cdot A’ and sorted A (x, y)

\[ \exists A, A', x', y, a, b. \text{listp} \mapsto \text{new} \ast \text{new} \mapsto [\text{val : v}, \text{next : list}] \ast \text{list} \mapsto [\text{val : a}, \text{next : b}] \]

\[ \ast \text{ls A'} (b, 0) \land \text{sorted A'} (x', y) \land x \leq x' \land x \leq a \land a \leq x' \land v \leq a \]
Rolling the atomic heap cells into the list predicate then gives us the following

\[ \exists A', y, a. \text{list} \mapsto \text{new} \ast \text{ls} \ (v \cdot a \cdot A') \ (\text{new}, \emptyset) \land \text{sorted} \ (v \cdot a \cdot A') \ (v, y) \]

which implies

\[ \exists B, v, y. \text{list} \mapsto \text{new} \ast \text{ls} \ B \ (v, \emptyset) \land \text{sorted} \ B \ (v, y) \]

which implies the postcondition. Note that to prove this implication, we used the fact that

\[ A = x' \cdot A' \land \text{sorted} \ A \ (x, y) \implies x' \leq x \land \exists x''. x \leq x'' \land \text{sorted} \ A' \ (x'', y) \]

Thus, even for analyzing non-looping code, the entailment procedure has to support reasoning over the pure abstract domain. Adding such support is one aspect of the proposed thesis work.

We now examine the loop. There is only one path that stays in the loop. It results in the following formula after one iteration

\[ \exists a, k, A', x, y. \text{list} \mapsto \text{list} \ast k \mapsto [\text{val} : a, \text{next} : \text{list}] \ast \text{ls} \ A' \ (\text{list}, \emptyset) \land \text{sorted} \ A \ (x, y) \land x \leq a \land A = a \cdot A' \]

and after two iterations, the formula

\[ \exists a, b, k, A'', x, y. \text{list} \mapsto k \ast k \mapsto [\text{val} : a, \text{next} : k'] \ast k' \mapsto [\text{val} : b, \text{next} : \text{list}] \ast \text{ls} \ A'' \ (\text{list}, \emptyset) \land \text{sorted} \ A \ (x, y) \land A = a \cdot b \cdot A'' \land x \leq a \land a \leq b \]

Applying the abstraction function for our shape analysis yields

\[ \exists a, b, k, B, A'', x, y. \text{list} \mapsto k \ast \text{ls} \ B \ (k, \text{list}) \ast \text{ls} \ A'' \ (\text{list}, \emptyset) \land \text{sorted} \ A \ (x, y) \land A = a \cdot b \cdot A'' \land B = a \cdot b \land x \leq a \land a \leq b \]

We then query the sortedness domain for the abstract representation of \( B \) and \( A'' \), obtaining the fact that \( \text{sorted} \ B \ (a, b) \) and \( \text{sorted} \ A'' \ (c, y) \land b \leq c \). Note that the abstraction into the sortedness domain of \( B \) yields an additional integer variable \( c \) and a constraint on this variable \( b \leq c \). These additional integer facts are then abstracted by whatever integer domain we use. This gives us the formula

\[ \exists a, b, c, k, B, A'', x, y. \text{list} \mapsto k \ast \text{ls} \ B \ (k, \text{list}) \ast \text{ls} \ A'' \ (\text{list}, \emptyset) \land \text{sorted} \ B \ (a, b) \land \text{sorted} \ A'' \ (c, y) \land x \leq a \land b \leq c \]

which is an invariant for the loop.

### 3.2.2 Sorted Trees

We now present a similar example for trees. We use the same sortedness domain, but use a shape predicate for trees instead of lists. The shape predicate we will use is:

\[ \text{tree} \ A \ a = (A = \epsilon \land a = \emptyset \land \text{emp}) \lor (A = A_1 \cdot x \cdot A_2 \land \exists k_1, k_2. a \mapsto [\text{val} : x, \text{left} : k_1, \text{right} : k_2] \ast \text{tree} \ A_1 \ k_1 \ast \text{tree} \ A_2 \ k_2) \]

With the following “fold” and “unfold” rules:

\[ a \mapsto [\text{val} : x, \text{left} : \emptyset, \text{right} : \emptyset] \implies \text{tree} \ x \ A \]
\[ a \mapsto [\text{val} : x, \text{left} : b, \text{right} : c] \ast \text{tree} \ b \ast \text{tree} \ c \ c \implies \text{tree} \ b \cdot x \cdot C \ a \]
\[ \text{tree} \ A \ a \land a \neq \emptyset \implies \exists x, b, c. a \mapsto [\text{val} : x, \text{left} : b, \text{right} : c] \ast \text{tree} \ b \ast \text{tree} \ c \ c \land A = B \cdot x \cdot C \]

We now consider the “sorted insert” function for trees, given in figure 2.
1: void insert_tree(Tree *treep, int val)
2: {
3:  Tree t = *treep;
4:  Tree new;
5: 
6:  if( t == NULL ) {
7:    new = malloc(sizeof(t));
8:    new->val = val;
9:    new->left = NULL;
10:   new->right = NULL;
11:   *treep = new;
12:   return;
13: }
14:  if( val > t->val )
15:    insert_tree(&(t->right), val);
16:  else
17:    insert_tree(&(t->left), val);
18: }

Figure 2: Procedure to insert an element in a binary search tree.

We wish to verify that the invariant $\exists tree_0, l, u. \ treep \mapsto tree_0 \land tree \land sorted A \langle l, u \rangle$ is preserved by the procedure. That is, we will check that this formula is both a precondition and postcondition of the procedure.

We have three cases to check. First is the base case (no recursive call). At line 7 we have

$$\exists l, u. \ treep \mapsto t \land tree \land sorted A \langle l, u \rangle \land t = \emptyset$$

which is equivalent to

$$\exists l, u. \ treep \mapsto t \land emp \land A = \epsilon \land t = \emptyset$$

Continuing, at line 12 we have

$$\exists l, u. \ treep \mapsto new \land new \mapsto [val : val, left : 0, right : 0] \land t = 0 \land A = \epsilon$$

which, applying the “fold” rules, abstracts to

$$\exists l, u. \ treep \mapsto new \land tree \land val \land new \land t = 0 \land A = \epsilon$$

which implies our goal.

Checking the path leading to the recursive calls, we get the following state at line 14

$$\exists l, u. \ treep \mapsto t \land tree \land sorted A \langle l, u \rangle \land t \neq \emptyset$$

which, expanding the definition of tree \ A \ t is equivalent to

$$\exists l, u, b, c. \ treep \mapsto t \land t \mapsto [val : x, left : b, right : c] \land tree \land sorted A \langle l, u \rangle \land t \neq \emptyset \land A = B \cdot x \cdot C$$

If we then take the branch at line 14, we obtain the following state at the first recursive call (line 15):

$$\exists l, u, b, c. \ treep \mapsto t \land t \mapsto [val : x, left : b, right : c] \land tree \land sorted A \langle l, u \rangle \land t \neq \emptyset \land A = B \cdot x \cdot C \land val > x$$

To apply our assumption about this procedure, we need to establish the precondition

$$\exists A, tree_0, l, u. \ treep \mapsto tree_0 \land tree \land sorted A \langle l, u \rangle \lor A = \epsilon$$
We use the frame rule to split the heap into two portions, one of which satisfies the precondition for the recursive call and one of which describes the rest of the heap. Note that this requires splitting the record at \( t \). This is not currently supported by our tool and we say more about how we could handle this in Section 5. Also important is the fact that \( \text{sorted } A \langle l, u \rangle \land A = B \cdot x \cdot C \implies \text{sorted } B \langle l, x_1 \rangle \land x_1 \leq x \land x \leq x_2 \land \text{sorted } C \langle x_2, u \rangle \).

The state we obtain is

\[
\exists l, u, b, c, A, B, C. \text{treep} \mapsto t * t \mapsto [\text{val} : x, \text{left} : b, \text{right} : c]
\]

\[
* \text{ tree } B \ b \land (\text{sorted } B \langle l, x \rangle \lor B = \epsilon)
\]

\[
* \text{ tree } C \ c \land (\text{sorted } C \langle x, u \rangle \lor C = \epsilon) \land A = B \cdot x \cdot C
\]

After execution of the recursive call, we obtain the same state, from which we can rebuild the tree at \( t \) and prove that sortedness has been preserved.

\[
\exists l, u, b, c, A, B, C, D. \text{treep} \mapsto t * \text{tree } D \ t \land D = B \cdot x \cdot C
\]

\[
\land \text{sorted } B \langle l, x_1 \rangle \land x_1 \leq x \land x \leq x_2 \land \text{sorted } C \langle x_2, u \rangle
\]

### 3.2.3 Sortedness Summary

To summarize, we see that there are three domains involved in this example: 1) the shape domain consisting of separation logic formulas with \(*\), \&, \& points-to, and is \( A \langle x, y \rangle \) (or tree \( A a \)), 2) the domain of sequences of integers, which we abstract to sorted intervals, and 3) the domain of integers, for which any abstract domain supporting inequalities of the form \( x \leq y \) will work for this example. Abstraction proceeds by first applying the abstraction operator for the shape analysis, which creates sequence expressions denoting the contents of the data structures involved in the program (be they lists, doubly-linked lists, or trees). An abstraction function is then applied to these sequence expressions, which may reveal additional arithmetic information. Finally, this arithmetic information is abstracted. Thus, we have a hierarchy of abstractions, each finding an abstract representation of elements in its domain and residuating information to be abstracted by the level below. When we consider that it is not uncommon for programs to involve structures such as lists of pairs, lists of lists of integers, etc., it becomes clear that this hierarchy of domains can be arbitrarily complex and this motivates the search for a general combination method for such domain hierarchies. This is the second and primary aspect of this thesis proposal.

### 3.3 Forall Domain

This domain keeps track of universal properties of lists and other data structures. Suppose we have a sequence with elements of type \( \tau \), where \( \alpha_\tau \) is an abstraction function over that type, \( \gamma_\tau \) is the concretization function, and \( \sqcap_\tau \) is the join operation in the abstract domain that is the range of \( \alpha_\tau \) (for a concrete example, take the integers with the octagon domain [28]). Finally, \( \bot_\tau \) is the bottom element in the abstract domain for \( \tau \).

Then the abstraction function for the forall domain is

\[
\alpha_{\forall(\tau)}(\epsilon) = (\bot_\tau)_\forall
\]

\[
\alpha_{\forall(\tau)}(x \cdot y) = \hat{x} \sqcap_\tau \hat{y} \text{ if } \alpha(x) = \langle \hat{x} \rangle_\forall \text{ and } \alpha(y) = \langle \hat{y} \rangle_\forall
\]

\[
\alpha_{\forall(\tau)}(t) = \alpha_\tau(t) \text{ if } t \in \tau
\]

Thus, the abstract representation of a sequence \( x \) is the least upper bound of the abstract representations of its elements. The concretization function is:

\[
\gamma_{\forall(\tau)}(\langle \hat{t} \rangle_\forall) = \{x_1 \cdot x_2 \cdot \ldots \cdot x_n \mid x_i \in \gamma_\tau(\hat{t})\}
\]

So far we have presented abstraction functions without any mention of implementation details. We give a more algorithmic description of the sortedness and forall domains in Section 4.
3.4 Regular Tree Grammars

As described in the work on refinement types [18, 15, 17], regular tree grammars provide a useful formalism for capturing certain classes of data structure invariants. Since regular tree grammars describe sets of terms in an abstract algebra, they are well-suited for describing properties of algebraic data types. By using a shape analysis based on separation logic and shape predicates that expose the term being manipulated, we can extend these techniques to imperative data structures.

**Definition 1** A regular tree grammar is a tuple \((S, N, \Sigma, R)\), where \(N\) is a set of non-terminals, and \(\Sigma\) is an alphabet where each symbol \(\sigma \in \Sigma\) has an associated arity \(a(\sigma)\). \(S \in N\) is the start symbol, and \(R\) is a set of rules of the form \(A \rightarrow t\) where \(A \in N\) and \(t \in T(\Sigma \cup N)\) (we use \(T(\Sigma \cup N)\) to denote the set of terms built from functions in \(\Sigma\) and constants in \(N\)).

Some examples of regular tree grammars include even / odd properties of lists:

\[
\begin{align*}
even\_list & \rightarrow \text{empty} \\
even\_list & \rightarrow x \cdot \odd\_list \\
odd\_list & \rightarrow x \cdot \even\_list
\end{align*}
\]

Here, if the start symbol is taken to be \(even\_list\), we get precisely the set of lists with even length. A more compelling example is the red / black invariant of red / black trees. This invariant states that each node is either red or black, all leaf nodes are black, and both children of a red node are black. Suppose we have a tree predicate that records the contents of the color fields of the tree as an algebraic entity.

\[
\begin{align*}
tree\ e\ x & \overset{\text{def}}{=} \text{emp} \land x = \emptyset \\
tree\ \text{node}(v, a, b)\ x & \overset{\text{def}}{=} \exists k_1, k_2, v. \ x \mapsto [\text{color} : v, \text{left} : k_1, \text{right} : k_2] \ast \text{tree a k}_1 \ast \text{tree b k}_2
\end{align*}
\]

In a C implementation of a red / black tree, we would define integer constants to represent the coloring information at each node:

```c
#define RED 0
#define BLACK 1
```

We can then define an abstract domain that tracks the color of a node

\[
\begin{align*}
\alpha(0) & = \text{red} \\
\alpha(1) & = \text{black} \\
\alpha(n) & = \top \text{ if } n \neq 0 \land n \neq 1
\end{align*}
\]

And the abstraction of a tree is then simply \(\alpha\) applied to the values at the nodes

\[
\begin{align*}
\alpha_{\text{tree}}(\epsilon) & = \epsilon \\
\alpha_{\text{tree}}(\text{node}(n, a, b)) & = \text{node}(\alpha(n), \alpha_{\text{tree}}(a), \alpha_{\text{tree}}(b))
\end{align*}
\]

The red / black invariant can then be represented as the following regular tree grammar

\[
\begin{align*}
\text{black} \_\text{root} & \rightarrow \epsilon \\
\text{black} \_\text{root} & \rightarrow \text{node(\text{black}, \text{tree}, \text{tree})} \\
\text{red} \_\text{root} & \rightarrow \text{node(\text{red}, \text{black} \_\text{root}, \text{black} \_\text{root})} \\
\text{tree} & \rightarrow \text{black} \_\text{root} \\
\text{tree} & \rightarrow \text{red} \_\text{root}
\end{align*}
\]

This grammar defines a finite lattice, where each of the non-terminals is a point in the lattice. The concretization \((\gamma_{rb})\) of a lattice element is the set of trees represented by the associated non-terminal. For example, \(\gamma_{rb}(\text{red} \_\text{root})\) is the set of finite trees produced by taking \(\text{red} \_\text{root}\) as the starting symbol and using the
rules above. Regular tree automata [10] provide an efficient implementation for regular tree grammars and include support for for the lattice operations $\sqcup$ (union) and $\sqcap$ (intersection). In [15], Davies shows that type refinements based on regular tree grammars are useful for expressing a range of datastructure invariants and develops an efficient implementation. By exploiting the connection between concrete and abstract data provided by separation logic shape predicates, we can utilize the same analysis techniques for low-level imperative programs that explicitly manipulate pointer structures.

Combination Summary

In each of the above examples, we see how shape analysis with separation logic predicates provides opportunities for abstract domains that capture properties of data structures. Furthermore, as datastructures contain elements of various types, which themselves require an abstract interpretation, we end up with a hierarchy of abstractions that reflects the type hierarchy we might have were we to write these examples in a purely functional language. However, it is important to note that operations are permitted in this setting which are not permitted in a purely functional setting. For example, doubly-linked lists permit concatenation at either end and iteration in either direction, whereas algebraic lists permit only recursion in one direction. In many applications, the distinction may not matter, and in fact there are functional implementations of deques with amortized constant-time access to either end. However, in some cases, such as time-sensitive applications, amortized constant time is not sufficient and an implementation based on doubly-linked lists is required. Furthermore, I would argue that in the world of safety-critical embedded software, where verification is most likely to be of interest, realtime constraints are likely to necessitate the use of some explicit pointer structures. And even in the realm of high-level languages, such as ML, state tends to play a role in many applications. Techniques such as those I am proposing for this thesis work can benefit all languages where programmers have a need to use pointer or reference types.

4 Combination Problem Statement (incomplete)

We now formally state the combination problem and give a more detailed presentation of the sortedness example, including a formal summary of our shape domain.

4.1 Multi-sorted Separation Logic

As our concrete domain, we take the formulas of multi-sorted separation logic. This is similar to standard separation logic, but enriched with a language of types that keeps track of the domain from which particular values are drawn. This logic is not the same as higher-order separation logic [4] or the separation logic used for Standard ML in [25]. These logics are much richer (and thus harder to work with in an automated setting).

4.1.1 Syntax

The type language is based on a set $\mathcal{T}$ of type constructors. Each $c \in \mathcal{T}$ has an associated arity $a(c)$ (type constants are functions with arity 0). The type language is then $T(\mathcal{T})$, the set of terms constructed from elements of $\mathcal{T}$. We use $\tau$ to refer to terms from this set. We assume the existence of two base types: $o$ the type of truth values and $addr$ the type of addresses.

For the term language, we assume a set $\mathcal{F}$ of functions and a function $a : \mathcal{F} \to \mathbb{N}$ which gives the arity of each element of $\mathcal{F}$. We also require a function $\sigma$, which for each $f \in \mathcal{F}$ gives the type of $f$ as a list of argument types paired with a result type. That is, $\sigma : \mathcal{F} \to T(\mathcal{T})$ list $\times$ $T(\mathcal{T})$. To simplify the system, we do not support quantification at the type level. The terms are then variables, indexed by their type, and function applications.

$$\text{terms} \quad t ::= \quad x_\tau \mid f(t_1,\ldots,t_{a(f)})$$

Note that our type and term languages support only application. The set of functions at each level is fixed. We do this in order to avoid the complexities of more expressive systems such as higher-order separation logic [4]. We also assume that terms of type “addr,” which we will represent with the meta-variables $e$, $f$ and $g$, are either variables ($x_{addr}$), or the constant $\Theta$, which denotes the null address. There are no other constants.
or functions of type “addr”. This is to rule out pointer arithmetic and ensure decidability of equality between addresses. This restriction can be relaxed provided that decidability is maintained, and in fact this work has been carried out in [7].

The type checking rules are simple. We write \( t : \tau \) to mean that \( t \) has the type \( \tau \).

\[
\frac{T-\text{VAR}}{x : \tau} \quad \frac{T-\text{APP}}{\sigma(f) = ([\tau_1, \ldots, \tau_n], \tau) \quad \forall i, t_i : \tau_i \quad f(t_1, \ldots, t_n) : \tau}
\]

The language of formulas is then a restricted form of separation logic formulas. The motivation for the restriction is that it enables a forward analysis of the program which avoids the need for magic wand. We divide the syntax into pure formulas (\( P \)) and spatial formulas (\( S \)). We will use \( p \) to represent a term of type \( o \). In the spatial formulas, we include only a predicate for doubly-linked lists, but other shape predicates, such as those we saw earlier for trees, can be added. The predicate \texttt{emp} denotes the empty heap, and \( e \mapsto [l_1 : t_1, l_2 : t_2, \ldots, l_n : t_n] \) describes the heap consisting of a single heap cell at address \( e \) that contains a record where field \( l_1 \) maps to value \( t_1 \), \( l_2 \) maps to \( t_2 \), etc. General formulas are specified by the grammar for \( M \) and consist of the possibly quantified conjunction of a spatial and pure formula. We write \( \bar{x}_P \) to represent \( x_1^{\tau_1}, \ldots, x_n^{\tau_n} \).

\[
\begin{align*}
\text{Pure} & : \quad P ::= \quad p \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid \neg P \\
\text{Spatial} & : \quad S ::= \quad \texttt{emp} \mid t \mapsto [l_1 : t_1, \ldots, l_n : t_n] \mid \text{dll} A (x_{\text{addr}}, y_{\text{addr}}, x\′_{\text{addr}}, y\′_{\text{addr}}) \mid S \land S \\
\text{General} & : \quad M ::= \quad \exists \bar{x}_P, S \land P
\end{align*}
\]

We will often leave out the type annotations when it is clear what they should be (for example, with \( x \mapsto [l_1 : t_1, \ldots, l_n : t_n] \), we know that \( x \) must have type “addr”).

### 4.1.2 Semantics

The semantics corresponds to the standard semantics for separation logic formulas, constrained by the type information. The meaning of a type \( \tau \) is written \( [\tau] \) and consists of the set of terms of type \( \tau \). We refer to \( [\text{addr}] \) as Addr and require that this set be infinite. A formula is evaluated with respect to an interpretation, which is a function \( \mathcal{I} \) with domain \( \mathcal{F} \) such that for each \( f \in \mathcal{F} \), if \( \sigma(f) = ([\tau_1, \ldots, \tau_n], \tau) \) then \( \mathcal{I}(f) \) is a function from \( [\tau_1] \times \ldots \times [\tau_n] \) to \( [\tau] \). The semantics of terms is given with respect to a valuation \( \phi \) that maps each \( x_{\tau} \) to an element of \( [\tau] \).

\[
\begin{align*}
[x_{\tau}] (\phi) &= \phi(x_{\tau}) \\
[f(t_1, \ldots, t_n)] (\phi) &= (\mathcal{I}(f)) ([t_1] (\mathcal{I}, \phi), \ldots, [t_n] (\mathcal{I}, \phi))
\end{align*}
\]

We refer to the pair \((\mathcal{I}, \phi)\) as a model and write \( \models_M \) to represent validity in the model \( M \). The validity of a formula is defined with respect to a valuation and a heap, which is a finite partial function from non-null addresses to records. Records are functions from a finite set of fields to values.

\[
\begin{align*}
\text{Val} & \overset{\text{def}}{=} \bigcup \{ [\tau] \mid a(\tau) = 0 \} \\
\text{Record} & \overset{\text{def}}{=} \text{Field} \to \text{Val} \\
\text{Heap} & \overset{\text{def}}{=} (\text{Addr} - 0) \overset{\text{fin}}{\to} \text{Record}
\end{align*}
\]

Validity of formulas is then defined as follows. We write \( \phi = \phi \setminus x \) if \( \phi \) and \( \phi' \) agree on variables not in \( x \). We write \( h_1 \cup h_2 \) to represent the union of \( h_1 \) and \( h_2 \) when they have disjoint domains. If the domains are
not disjoint, then $h_1 \cup h_2$ is undefined.

\[
(\phi, h) \models S \land P \quad \text{iff there exists } \phi' \text{ such that } \phi = \phi' \setminus \bar{x}_\pi \text{ and } (\phi', h) \models S \text{ and } (\phi', h) \models P
\]

\[
(\phi, h) \models \text{emp} \quad \text{iff } h = \emptyset
\]

\[
(\phi, h) \models t \mapsto [I_1 : t_1, \ldots, I_n : t_n] \quad \text{iff } h = \{(I_1, (\phi, \rho)) \ldots, (I_n, [I_n] (\phi))\}
\]

\[
(\phi, h) \models S_1 \ast S_2 \quad \text{iff } h = h_1 \cup h_2 \text{ and } (\phi, h_1) \models S_1 \text{ and } (\phi, h_2) \models S_2
\]

\[
(\phi, h) \models P \quad \text{iff } [p] (\phi) = \text{true}
\]

\[
(\phi, h) \models P_1 \land P_2 \quad \text{iff } (\phi, h) \models P_1 \text{ and } (\phi, h) \models P_2
\]

\[
(\phi, h) \models P_1 \lor P_2 \quad \text{iff } (\phi, h) \models P_1 \text{ or } (\phi, h) \models P_2
\]

\[
(\phi, h) \models \neg P \quad \text{iff } (\phi, h) \not\models P
\]

### 5. Shape Predicates

Finally, I propose to add support for trees to the tool and the shape analysis formalism. Berdine, et al. considered support for a simple tree predicate in [2], but as we will see here, this predicate is not strong enough to support iterative tree traversal.

The simple tree predicate is

\[
\text{tree } a \overset{\text{def}}{=} (a = \emptyset \land \text{emp}) \\
\quad \lor (\exists k_1, k_2. a \mapsto [\text{val} : x, \text{left} : k_1, \text{right} : k_2] \\
\quad \ast \text{tree } k_1 \ast \text{tree } k_2)
\]

This works for recursive procedures. Consider the sorted insert procedure given below.

```c
1: void insert_treeR(Tree *treep, int val)
2: {
3:   Tree t = *treep;
4:   Tree new;
5:   if( t == NULL ) {
6:     new = malloc(sizeof(t));
7:     new->val = val;
8:     new->left = NULL;
9:     new->right = NULL;
10:    *treep = new;
11:    return;
12:  }
13:  if( val > t->val )
14:    insert_treeR(&(t->right), val);
15:  else
16:    insert_treeR(&(t->left), val);
17:  }
18: }
```

Suppose the precondition that is given is $\exists k. (\text{treep} \mapsto k) \ast \text{tree } k$ and the postcondition is the same. If we look at the invariant at the first recursive call (line 15), we see that we have

\[
(treep \mapsto t) \ast t \mapsto [\text{left} : a, \text{right} : b, \text{val} : v] \ast \text{tree } a \ast \text{tree } b
\]

The frame rule lets us extend the pre- and postcondition for the function to this formula which then still holds after the recursive call. Since the formula above implies the postcondition, we are done with this case. Checking that the second recursive call and the base case both result in a state satisfying $\exists k. \text{treep} \mapsto k \ast \text{tree } k$ completes the verification task.

Now consider what happens if we write an iterative version of this same routine.
1: void insert_tree(Tree *treep, int val)
2: {
3:     Tree t = *treep;
4:     Tree new;
5:     /* allocate new node */
6:     new = malloc(sizeof(t));
7:     new->val = val;
8:     new->left = NULL;
9:     new->right = NULL;
10: 
11:     /* empty tree */
12:     if( t == NULL ) {
13:         *treep = new;
14:         return;
15:     }
16: 
17:     Tree prev;
18:     while( t != NULL ) {
19:         prev = t;
20:         if( val > t->val )
21:             t = t->right;
22:         else
23:             t = t->left;
24:     }
25: 
26:     if( val > prev->val )
27:         prev->right = new;
28:     else
29:         prev->left = new;
30: }

The invariant for this loop is not representable using the simple tree predicate defined above. We need some notion of a “tree with a hole” or context to indicate where we are in the data structure. For example, the following predicate will work

\[
\text{treeh}(a, b) \overset{\text{def}}{=} (a = b \land \text{emp}) \\
\lor (a \neq b \land \exists k_1, k_2. a \mapsto [\text{val} : x, \text{left} : k_1, \text{right} : k_2] \\
* \text{tree} (k_1, b) * \text{treeh}(k_2, b)) \\
\lor (a \neq b \land \exists k_1, k_2. a \mapsto [\text{val} : x, \text{left} : k_1, \text{right} : k_2] \\
* \text{treeh}(k_1, b) * \text{tree} (k_2))
\]

The invariant is then

\[
\exists k. \text{treep} \mapsto k * \text{treeh}(k, t) * \text{tree} (t)
\]

Note, however, that this predicate is not precise. Looking at the definition, we see that there is no way to know which of the two inductive cases holds. This manifests itself globally as a failure of the predicate to store where in the tree the hole is located. A possible remedy for this problem is to use a predicate corresponding to “zippers” [23], which are trees with holes that explicitly store the path from the root to the hole.

## 6 Related Work

There has been considerable work in the field of shape analysis, with the most successful system thus far being TVLA [31]. This is an abstract interpretation based on tree-valued logic and shape graphs. It provides
a very general analysis, but suffers from the fact that it does not permit compositional reasoning. That is, since non-aliasing information is not tracked, an exponential number of cases must be considered when taking postconditions. And since the analysis is over the global memory state the number of variables that must be considered and their potential interactions can be quite large. This results in TVLA being a relative expensive analysis. Recently, progress has been made in speeding up the TVLA analysis [5]. The exponential worst-case complexity is still present, but clever heuristics and implementation techniques can be used to speed up the analysis considerably. However, it still suffers from the lack of something like separation logic’s “frame rule” in that analysis of code is dependant on the context, so it is difficult to perform modular verification.

Our work on [27], together with the work of Distefano et. al. [16] was the first investigation of using separation logic formulas for shape analysis. There have since been a number of advancements in this area including work on interprocedural analyses [19] and support for concurrency [20]. All of these approaches require a human to specify the inductive shape predicates and associated axioms. There is recent work by Guo et. al. on a fully automated analysis which infers even these axioms [22]. This brings up the issue of how to provide useful feedback to the programmer in the case where the analysis fails, but they have not provided any annotations that we can say are definitely violated. As Bor-Yuh Chang et. al. point out, the invariant checking functions that programmers often write can be used to guide the analysis [9].

There has been a variety of work on combining abstract domains. Domains such as the direct and reducted product [13] have been known for quite some time. However, the direct product produces a very imprecise analysis and general techniques for computing the reduced product are not known. The closest work to that I have proposed is the congruence-closure abstract domain described by Bor-Yuh Chang in [8]. However, there are several differences. First, whereas in [8] each domain is aware of a disjoint set of operations, we have a richer interaction between our domains. For example, the sortedness domain is aware of arithmetic inequalities and uses their properties when performing abstraction, however it does not know how to abstract integer variables and relies on a relational domain for integers for that functionality. Also, whereas the domains in [8] are on equal footing, our are arranged hierarchically, which should result in lower worst-case complexity bounds for operations over the abstract domain. The work in [21] is similar, in that it takes a Nelson-Oppen approach to combination [29], requiring that theories be disjoint and ruling out the sort of interaction between domains that my proposed approach allows.

There is a great deal of work related to the specific combination of shape and arithmetic, which we explored in [26]. In [3], the authors describe a method in which the TVLA [31] shape analysis is lazily combined with an arithmetic analysis based on BLAST. This work reverses the strategy that we propose: they take an arithmetic analysis as their primary analysis and lazily provide support for spatial reasoning. My work, on the other hand, starts with a shape analysis and adds pure domains as necessary to prove safety. One consequence of this difference in focus is that the algorithm given in [3] cannot prove memory safety. In [6], list programs are reduced to integer programs which are then analyzed using counter automata. However, details of the abstraction process are lacking. The work in [26] provides the formal connection between a shape analysis over lists and integer programs over their lengths. And the research I propose in this document extends this formal connection to other data structure abstractions. Further discussion of related work on shape and arithmetic can be found in [26].

Another unique aspect of the proposed work is the use of abstract domains over algebraic terms representing the data being manipulated and its structure. The most closely related work, and in fact the inspiration for such analyses, is the work on refinement types [18, 15, 33, 17]. Datasort refinements [18, 15] correspond to adopting an abstract domain defined by a regular tree grammar. Index refinements [17, 33] are similar to our handling of integer domains, e.g. for the analysis of programs that depend crucially on the lengths of the lists involved. One difference in the proposed work is that I aim to develop an automated procedure based on computing an abstract interpretation of the program rather than specifying a type system which enables the checking of annotated programs. Of course the distinction is not so vast, since type inference can be viewed as an abstract interpretation [11]. The main difference is that the proposed analysis is for imperative programs that explicitly manipulate pointer structures in the heap rather than functional programs that manipulate elements of algebraic data types. Furthermore, I am to obtain a general method of combination, whereas the current work on refinement types has focused on careful combinations of decidable domains.
7 Conclusion

Separation logic [30] can be viewed in many ways. At its core, it is a Hoare logic capable of local reasoning about heap effects and can be used to support manual proofs of full correctness. A stripped down version of the logic can be used as an abstract domain in which to do shape analysis for the purpose of automatically generating proofs of memory safety for programs [27, 16]. By slightly modifying the shape predicates we use, we can view separation logic as providing a precise, formal connection between a concrete, pointer-based implementation of a data structure and that structure’s abstract, mathematical representation. It is this feature of separation logic that I will concentrate on in this proposal. Once we have a mathematical representation of a data structure, we can then look at abstract domains that allow us to reason automatically about invariants of that data structure. To accomplish this requires not only a general abstraction framework for hierarchical combinations of domains, but also support for these domains in the prover utilized by the tool. I propose investigating both the general abstraction framework and the necessary support in the entailment system and implementing these algorithms in the static analysis tool I developed for my previous work.

8 Timeline

I now present a timeline for completion of the proposed work. The steps are presented in the order in which I plan to address them. However this is tentative, as some steps are independent and may end up being handled in a different order.

Clean up implementation (1 month) In the rush to get experimental results for the SAS deadline, some compromises were made in terms of organization, robustness, and generality of the code. I would first like to take some time to fix these issues as it will make subsequent implementation work easier.

Combination with static analyses over sequences (4 months) Develop and experiment with a combination of shape analysis and sequence analyses, much like what we did for arithmetic. The sequence analyses would include sortedness, the forall domain, and possibly others.

Combination with static analyses over tree-structured data (5 months) Add support for tree-structured data to the tool and investigate abstractions based on regular tree grammars. In this case, the lattice of tree abstractions would be given by the programmer, just as it is in the refinement types work. If time permits and if it seems feasible, it would also be interesting to investigate the possibility of using the learning techniques developed by Nishant [32] to automatically generate appropriate abstractions.

Formal investigation of entailment (6 months) Can the entailment procedure be made complete for the fragment? What data structures can be introduced while retaining completeness? Is there a general class of data structures that can be decided? Some of these questions may end up being outside the scope of the thesis. For example, it is not currently known whether the fragment of separation logic that includes just $\ast, \rightarrow$, equality, and a doubly-linked list predicate is decidable. It is impossible to predict currently how difficult it would be to answer this decidability question. If it becomes clear that answering this would be a major undertaking on its own, we will settle for a formal and thorough description of a sound, terminating, but not necessarily complete entailment procedure. Also, this work will be carried out in parallel with the other work.

Writing (6 months) Time to write papers and thesis text corresponding to the above objectives.

References


