Automatic Numeric Abstractions for Heap-Manipulating Programs [*]

Stephen Magill
Carnegie Mellon University
smagill@cmu.edu

Ming-Hsien Tsai
National Taiwan University
mhtsai208@gmail.com

Peter Lee
Carnegie Mellon University
Peter.Lee@cs.cmu.edu

Yih-Kuen Tsay
National Taiwan University
tsay@im.ntu.edu.tw

Abstract

We present a logic for relating heap-manipulating programs to numeric abstractions. These numeric abstractions are expressed as simple imperative programs over integer variables and have the property that termination and safety of the numeric program ensures termination and safety of the original, heap-manipulating program. We have implemented an automated version of this abstraction process and present experimental results for programs involving a variety of data structures.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Logics of Programs, Mechanical Verification

General Terms Languages, Reliability, Theory, Verification

Keywords shape analysis, separation logic, termination, program verification, abstraction

1. Introduction

Current static analysis tools can check a wide variety of both safety and liveness properties for programs involving integer variables. Tools such as BLAST [20], SLAM [1], ARMC [29], ASTRÉE [15], SPEED [18] and TERMINATOR [14] all focus on this class of programs. Some of these have support for pointers, but the heap reasoning is generally kept as simple as possible for the given problem domain.

A major challenge is integrating these methods with very precise methods for heap analysis. Such combinations generally involve a large increase in complexity, both in terms of the verification problem and the implementation. In this paper, we offer a solution to this problem in the form of a logic for relating programs that allows a heap program to be related to a numeric program in a way that is useful for automated verification. The numeric program simulates the original program, ensuring that safety and liveness results obtained by analyzing the numeric program also hold of the original, heap-manipulating program. Also, invariants of the numeric program can be translated into invariants of the heap program. Finally, the simulation result also implies that bounds on variables are preserved, which, when combined with some additional instrumentation, allows us to use the numeric program to calculate bounds on execution time and memory usage.

The numeric program may include additional variables, called *instrumentation variables*, which are not present in the input program. These variables track numeric properties of heap-based data structures, such as the height of a tree, the maximal element in a list of integers, or the length of a path between two points in a data structure. Generating a proof that two programs are related involves following a separation logic approach to reasoning about the heap-manipulating program and using the invariants thus discovered to generate appropriate commands and branches on the instrumentation variables. These commands and branches record the connection between operations and conditions on pointer variables and corresponding operations and conditions involving the instrumentation variables.

Our proof system specifies the conditions under which certain program transformations are sound for safety and liveness and permits a great deal of reasoning power, allowing the production of numeric abstractions for programs using complicated data structures that cannot currently be handled by automated tools. For the class of programs and data structures to which automated shape analyses such as [16] and [23] are applicable, we have produced an implementation that can automatically generate numeric abstractions.

The implementation we obtain involves only small modifications to an existing shape analysis algorithm. Thus, when viewed from a static analysis point of view, this paper provides a technique for combining shape analyses based on separation logic with arbitrary numerical analyses that requires no re-implementation of the numerical analysis and only small modifications to the shape analysis. Our implementation is built on the THOR [24] shape analysis tool and we demonstrate the practicality of our approach by proving safety and termination of a number of C programs using a variety of numeric static analysis tools.

2. Related Work

In [22] we presented a static analysis algorithm for performing a similar translation for programs involving only linked lists. The translation was used for safety reasoning and, while technically sound for termination, it was unusable in this context, as the numeric programs it generated were too imprecise to allow us to obtain termination results for any examples. However, the biggest shortcoming is that soundness was only shown for the specific analysis described in the paper. In contrast, the logic presented here provides a general characterization of the terms under which a numeric abstraction is sound, in much the same way that Hoare logic provides a general specification of when program invariants are satis-

[*] This work was partially supported by the iCAST project sponsored by the National Science Council (NSC), Taiwan, under the grants No. NSC96-3114-P-001-002-Y and No. NSC97-2745-P-001-001. The second and fourth authors were also partially supported by NSC grants No. NSC97-2221-E-002-074-MY3 and No. NSC98-2219-E-011-001.

fied (although Hoare logic is relatively complete, whereas we have not investigated completeness properties of the system presented here). This provides a general goal for static analysis work such as [22]: ideally we would be able to generate a numeric abstraction for any program that can be related "by hand" to a numeric program by the logic presented in this paper. In addition to defining this target, the implementation developed for this paper also advances the state of the art in terms of achieving this goal by providing support for automatic generation of numeric abstractions for programs using user-defined inductive data structures and customized notions of data structure size, both of which were missing in [22].

In [13], an early implementation of this work is used to obtain upper bounds on allocated memory to enable the synthesis of hardware from C programs that use dynamic data structures. A means of obtaining numeric abstractions for C code that yields correct upper bounds is a crucial component of that work, but is not described in the paper. This is the first presentation of these ideas.

The concept of relating two programs at different levels of abstraction is used heavily in the area of program refinement [32]. The use of this concept to automatically generate a more abstract version of a program has been very successfully applied in predicate abstraction [1]. The goal of our work is similar, but we generate numeric programs over unbounded integers, which have an infinite state space, whereas predicate abstractions are finite-state.

Another approach to relating programs, based on a relational version of Hoare logic, is given by Benton in [2] and Yang gives a similar relational version of Separation Logic in [33]. The goal is to relate two programs when their total correctness properties are the same. In our work, since we are only concerned with obtaining an over-approximation of the original program, the numeric program may diverge in cases where the original program terminates. We also are able to get by with a logic where the annotations represent sets of states rather than relations. Indeed, the main goal of our work is to offload the relational reasoning to separate analysis tools.

Our instrumentation variables are similar in usage to auxiliary variables in Hoare logic [27]. Both auxiliary variables and instrumentation variables are not permitted to affect the values of the original variables nor the control flow of the original program. The work here presents a more thorough treatment of the concept of auxiliary variable, more clearly specifying how auxiliary variables may affect execution and how auxiliary variables relate to existentially quantified variables. The standard rule for proof by auxiliary variables can be viewed as an instance of Corollary 2.

Our treatment of existential quantifiers is also a key difference between this work and other work in logics for relating programs. Because we state soundness in terms of simulation, we are able to use the INST-EXISTS rule, which is explained in Section 5, Figure 11 to insert and update variables representing values that are quantified in the original program. We thus obtain information about how these values change without resorting to relational invariants.

Termination proving for heap-manipulating programs has been described in [21] and [30]. Both of these approaches utilize a different shape analysis framework and [21] does not involve the production of numeric abstractions, instead incorporating a rank-finding algorithm directly in the analysis. The work in [30] does involve the production of numeric abstractions, but they are produced from counter-example traces generated by the termination analysis and used to communicate with the heap analysis, which is run only on-demand. By contrast, we convert an entire program to a numeric abstraction before doing any termination analysis, which permits a looser coupling between the termination tool and the shape analysis tool. In [7], Brotherston et al. give a method of showing termination of programs using separation logic, based on the notion of cyclic proofs. However, they do not give a static analysis capable of automatically generating these proofs. It is also not clear that such an approach can handle cases where more complicated termination arguments, such as lexicographic orderings, are needed. In [4] a method is presented for using a separation logic shape analysis to prove termination. However, that work is tied to a specific rather weak abstract domain for tracking size changes. The approach described here is able to obtain much more precise information by tracking the actual change in data structure size rather than only the presence and direction of change.

The shape analysis portion of our implementation is not new and has much in common with other recent work on shape analysis with separation logic [3, 12, 17]. There has also been previous work on extending shape analysis with support for tracking integer properties. Chang et al. have extended their approach to support numeric invariants of data structures [11]. Calcagno et al. handle the case where arithmetic is allowed in the domain of the heap [10]. For approaches based on TVLA, there is the work of Beyer et al. [5]. Rugina develops an analysis targeting balance properties of tree-shaped data structures [31]. Nguyen et al. present a verification condition-based procedure that can handle shape plus size properties when loop invariants and pre- and post-conditions are provided [26]. However, none of these use the method described here of generating numeric programs as an intermediate step in the verification process.

The closest work to ours is that of Boujjani et al. [6] which gives a bi-simulation between programs manipulating singly-linked lists and counter automata and Habermehl et al. [19] which provides a termination result for trees by relating tree-manipulating programs to tree automata. By focusing on specific data structures, these papers are able to obtain very precise results. In our work, we obtain a simulation result rather than bi-simulation, but the result holds of arbitrary inductively-defined data structures.

3. Examples

Consider the program in Figure 1. This C-style code performs a left-to-right, depth-first traversal of the tree at `root`. It does this by maintaining a stack of nodes to be processed. The stack is implemented using a linked-list with nodes of type `TreeList` and initially contains a single node with a pointer to the root of the tree. On each iteration, the top element of the stack is removed and its children are added. Empty trees are discarded and when the entire stack is empty, execution terminates.

There are a number of properties one might want to prove about this code. First, we might like to show that it terminates. We might also be interested in obtaining a bound on the amount of memory allocated by the procedure. Both these questions are really questions about numeric properties of the code. In the case of termination, we want to demonstrate that some rank function decreases during each iteration. For a bound on the number of memory cells used, we can imagine adding a variable `mem_usage` to the program, which is initially zero and increments each time memory is allocated and decrements each time memory is freed. We might be interested in obtaining a bound on `mem_usage` in terms of the size of the input tree.

In this example, answering either of these questions requires some reasoning about the shape and size properties of heap-allocated data structures. What we show in this paper, and demonstrate in our experiments, is that the shape reasoning can be separated from the numeric reasoning by constructing a numeric program that explicitly tracks changes in data structure sizes.

A numeric program for this example is given in Figure 2. This program can be constructed from the original using the rules in Section 5 and an equivalent, though larger program can be constructed automatically by the analysis implementation discussed in Section 7. In each case, the variables in the numeric program correspond to size properties of the data structures involved.

```
struct Tree {
  Tree left;
  Tree right;
}
struct TreeList {
  Tree tree;
  TreeList next;
};

TreeList push(Tree r, TreeList next) {
  TreeList t;
  t = malloc();
  t->tree = r;
  t->next = next;
  return t;
}
void traverse(Tree root) {
  TreeList stack, tail;

  stack = push(root,0);
  while(stack != 0) {
    tail = stack->next;
    if(stack->tree == 0) { // remove empty trees
      free(stack);
      stack = tail;
    }
    else { // process non-empty trees
      tail = push(stack->tree->right,tail);
      tail = push(stack->tree->left,tail);
      free(stack);
      stack = tail;
    }
  }
}
```

Figure 1. A depth-first traversal of a tree rooted at `root`.

```
    void traverse(int tsize_root) {
 1:   assume(tsize_root >= 0);
 2:   slen = 1;
 3:   ssize = tsize_root;
 4:   while(slen > 0) {
 5:     tsize = ?; ssize_tail = ?;
 6:     assume(tsize >= 0 && ssize_tail >= 0);
 7:     assume(ssize == tsize + ssize_tail);

 8:     if(tsize == 0) // remove empty trees
 9:       slen--;
10        else {         // process non-empty trees
11:       tsize_l = ?; tsize_r = ?;
12:       assume(tsize_l >= 0 && tsize_r >= 0);
13:       assume(tsize == tsize_l + tsize_r + 1);
14:       ssize = tsize_l + tsize_r + ssize_tail;
15:       slen++;
        }
      }
    }
```

Figure 2. A numeric abstraction of the program in Figure 1.

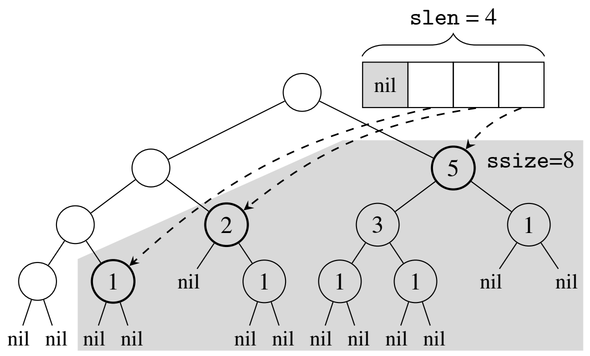

Figure 3. An example showing `slen` and `ssize` used in the program in Figure 2. `slen` is the number of nodes in the stack and `ssize` is the sum of the values in the bold circles. The shaded area contains the nodes that contribute to `ssize` and nodes in this area are labeled with the size of the subtree rooted at that node. Empty trees (denoted by "nil") have size 0.

Informally, `tsize_root` is the number of nodes in the tree at `root`, `slen` is the length of the list representing the stack, and `ssize` is the number of nodes in the trees linked to by nodes in the stack, as depicted in Figure 3. `ssize` and `slen` are the main integer variables present in the program. A number of temporary variables are then used to perform updates to these. These updates are sometimes non-deterministic. For example, in the main loop, we examine the first element of the stack and, if it links to a non-empty tree, we replace it with two nodes, each of which links to one of that tree's children. Thus, in the numeric program we must represent how removing an element from the stack changes the values `slen` and `ssize`. In the case of `slen`, the length simply decreases by one. For `ssize`, however, the effect of removing an element is not deterministic. The most we can conclude is that `ssize` can be broken into `tsize`, the size of the tree linked to by the element we just removed, and `ssize_tail`, the size of the remaining portion of the stack. This is accomplished by the non-deterministic assignments on line 5 coupled with the assume statements at lines 6 and 7. A similar situation occurs on line 13, when we record the relationship between `tsize` and the sizes of the left and right children (`tsize_l` and `tsize_r`, respectively).

While `assume` statements are not part of standard C, they are accepted by many verification tools, allowing us to pass the code in Figure 2 directly to ARMC or TERMINATOR in order to check termination. In this case, the termination argument involves a lexicographic order on `ssize` and `slen`. By producing numeric abstractions such as that given in Figure 2, we allow ourselves and our program analysis tool to concentrate on the shape analysis problem, while leaving details of lexicographic rankings or disjunctive well-foundedness [28] to other tools.

We can also ask bounds analysis tools as described in [18] and [13] for a bound on the length of the stack. In this case, the stack can grow as long as $\texttt{tsize\_root} + 1$ if the tree is maximally unbalanced. The theory presented in Section 5, coupled with an inductive description of a balanced tree, also allows us to obtain a numeric program that demonstrates the expected logarithmic bound on stack length for balanced trees.

Alternate Abstractions It is often the case that there are different notions of data structure size. The measures used in Figure 2 are fairly natural, in that the number of allocated heap cells is the sum of `slen` and `ssize`. If we abandon this correspondence, we can obtain the simpler numeric abstraction given in Figure 4. In this case we have only one main size variable, `ssize`, which tracks the sum of the sizes of the subtrees reachable through the stack. However, we alter the notion of tree size such that empty trees have size equal to one, as depicted in Figure 5. This simplifies the termina-

```
void traverse(int tsize_root) {
  assume(tsize_root > 0);
  ssize = tsize_root;
  while(ssize > 0) {
    tsize = ?; ssize_tail = ?;
    assume(tsize > 0 && ssize_tail >= 0);
    assume(ssize == tsize + ssize_tail);

    if(tsize == 1) // remove empty trees
      ssize = ssize_tail;
    else {         // process non-empty trees
      tsize_l = ?; tsize_r = ?;
      assume(tsize_l > 0 && tsize_r > 0);
      assume(tsize == tsize_l + tsize_r + 1);
      ssize = tsize_l + tsize_r + ssize_tail;
    }
  }
}
```

Figure 4. A numeric abstraction of the program in Figure 1 with a different notion of "ssize" and "tsize."

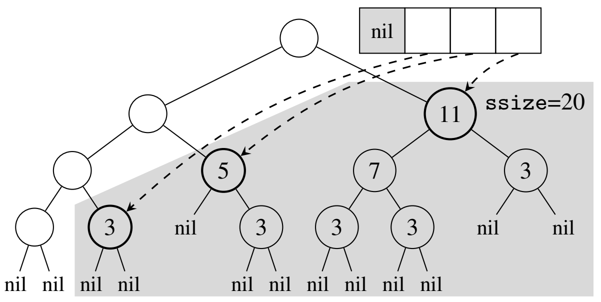

Figure 5. An illustration of the notion of `ssize` used in the program in Figure 4. The shaded area contains the nodes contributing to `ssize`. Empty trees (denoted by "nil") have size 1. Non-empty nodes are labeled with the size of the subtree rooted at that node. `ssize` is the sum of the values in the bold circles, plus 1 for the first element in the stack, as "nil" has size 1 using this notion of size.

tion argument, as there is now only a single count, `ssize`, which decreases during every iteration. However, we lose the ability to talk about the length of the stack when computing bounds and we lose the close connection between our counts and the number of allocated heap cells. The technique described in this paper has the flexibility to allow either approach to numeric abstraction, and the implementation is not tied to any fixed notion of size. Instead, we allow the user to specify the definition of size they have in mind when running the tool. The numeric abstraction corresponding to the input C program is then automatically generated for that notion of size.

4. Preliminaries

In this section, we define our programming language, which is based on C and thus includes explicit memory allocation and deallocation as well as unstructured flow of control.

4.1 Program Syntax

Figure 6 gives the syntax of programs. A program P is a list of labeled continuations, which can also be viewed as a mapping from labels to continuations (and we will often use function syntax for P, writing $P(l)$ for the continuation labeled with l in program P). We take the first location l_0 to be the initial location, at which

Variables	x^τ	$\in$	Vars_τ
Fields	f^τ	$\in$	Fields_τ
Labels	l	$\in$	L
Integers	n	$\in$	$\mathbb{Z}$
Integer Expns	e^{i}	$::=$	$x^{\mathrm{i}} \mid n \mid e^{\mathrm{i}}_1 + e^{\mathrm{i}}_2 \mid e^{\mathrm{i}}_1 - e^{\mathrm{i}}_2 \mid e^{\mathrm{i}}_1 \times e^{\mathrm{i}}_2$
Address Expns	e^{a}	$::=$	$x^{\mathrm{a}} \mid \mathrm{nil}$
Boolean Expns	e^{b}	$::=$	$\mathit{true} \mid \mathit{false} \mid \neg e^{\mathrm{b}} \mid e^{\mathrm{a}}_1 = e^{\mathrm{a}}_2 \mid$ $e^{\mathrm{i}}_1 \leq e^{\mathrm{i}}_2 \mid e^{\mathrm{b}}_1 \wedge e^{\mathrm{b}}_2 \mid e^{\mathrm{b}}_1 \vee e^{\mathrm{b}}_2$
Commands	c	$::=$	$x^\tau := e^\tau \mid x^{\mathrm{a}} := ?^{\mathrm{a}} \mid x^{\mathrm{i}} := ?^{\mathrm{i}} \mid$ $x^\tau_1 := x^{\mathrm{a}}_2.f^\tau \mid x^{\mathrm{a}}.f^\tau := e^\tau \mid$ $x^{\mathrm{a}} := \mathtt{alloc}(f_1, \ldots, f_n) \mid \mathtt{free}\ x^{\mathrm{a}}$
Continuations	k	$::=$	$c; k \mid \mathtt{halt} \mid \mathtt{abort} \mid \mathtt{goto}\ l \mid$ $\mathtt{branch}\ e^{\mathrm{b}}_1 \Rightarrow k_1, \ldots, e^{\mathrm{b}}_n \Rightarrow k_n\ \mathtt{end}$
Programs	P	$::=$	$l_0 : k_0;\ \ldots;\ l_n : k_n$

Figure 6. Syntax of programs

execution starts. A continuation is a branching structure consisting of conditional branches and commands that update the state. At the leaves of each branching continuation, we have either a goto or an indication that execution should halt or abort. Commands include the standard commands for variable assignment, heap lookup, heap mutation, memory allocation, and deallocation. The commands range over variables drawn from the infinite set Vars and field names drawn from the infinite set Fields.

We will write $k \in \mathit{subterms}(P)$ if k is a sub-term of some continuation in the range of P. A program P is considered *well-formed* iff $\{l \mid \mathtt{goto}\ l \in \mathit{subterms}(P)\} \subseteq \mathit{dom}(P)$, where $\mathit{dom}(P)$ is the domain of P (set of labels prefixing continuations in P). This ensures that all jumps are to locations defined by P. We also write $\mathit{fv}(P)$, $\mathit{fv}(c)$, $\mathit{fv}(e)$ to denote the set of free variables of the program P, command c, or expression e.

Variables and expressions are typed, with the types drawn from the set $\{\mathrm{a}, \mathrm{i}, \mathrm{b}\}$ (representing addresses, integers, and Booleans, respectively). We assume that the set Vars contains two infinite, disjoint subsets $\mathrm{Vars}_{\mathrm{a}}$ and $\mathrm{Vars}_{\mathrm{i}}$ such that $\mathrm{Vars} = \mathrm{Vars}_{\mathrm{a}} \cup \mathrm{Vars}_{\mathrm{i}}$. We do not include variables of type b in our syntax or states. We write x^{a} to denote an element of $\mathrm{Vars}_{\mathrm{a}}$ and x^{i} for an element of $\mathrm{Vars}_{\mathrm{i}}$. We use τ to stand for either a or i. Often, types can be inferred from the context and, in such cases, we will omit them. We take a similar approach to typing of record fields. We assume the set Fields can be partitioned into two disjoint subsets $\mathrm{Fields}_{\mathrm{a}}$ and $\mathrm{Fields}_{\mathrm{i}}$ and write f^{a} for elements of $\mathrm{Fields}_{\mathrm{a}}$ and f^{i} for elements of $\mathrm{Fields}_{\mathrm{i}}$.

4.2 Program Semantics

The semantics is given in terms of transitions between states. Each non-error state includes a *store* paired with a *heap*. Formally, a store is a mapping from variables to their values. We require that this mapping respect types and indicate this by using the notation $\rightarrow_\tau$ to denote the function space. A function f is in Vars $\rightarrow_\tau$ Values iff $f \in$ Vars $\rightarrow$ Values and variables in $\mathrm{Vars}_{\mathrm{i}}$ are mapped to integers while variables in $\mathrm{Vars}_{\mathrm{a}}$ are mapped to addresses.

$$\mathrm{Values} \overset{\mathrm{def}}{=} \mathbb{Z} \cup \mathrm{Addr}$$
$$s \in \mathrm{Stores} \overset{\mathrm{def}}{=} \mathrm{Vars} \rightarrow_\tau \mathrm{Values}$$

The heap is a finite partial function from non-null addresses to *records*, which are finite partial functions from fields to values of the appropriate type. The set Addr is assumed to be infinite. We will use the meta-variable s to represent an element of Stores and h to represent an element of Heaps. As with stores, the functions that serve as the denotation of records must respect types. Unlike stores,

SEMANTICS OF COMMANDS

$$
\begin{array}{rcll}
[\![x^\tau := e^\tau]\!]\,(s,h) & = & \{(s[x^\tau \to [\![e^\tau]\!]\,s], h)\} & \\
[\![x^{\mathrm{a}} := ?^{\mathrm{a}}]\!]\,(s,h) & = & \{(s',h) \mid s' = s[x^{\mathrm{a}} \to v] \wedge v \in \mathrm{Addr}\} & \\
[\![x^{\mathrm{i}} := ?^{\mathrm{i}}]\!]\,(s,h) & = & \{(s',h) \mid s' = s[x^{\mathrm{i}} \to v] \wedge v \in \mathbb{Z}\} & \\
[\![x_1^\tau := x_2^{\mathrm{a}}.f^\tau]\!]\,(s,h) & = & \{(s[x_1^\tau \to (h(s(x_2^{\mathrm{a}})))\, f^\tau], h)\} & \text{if } s(x_2^{\mathrm{a}}) \in \mathit{dom}(h) \\
 & & & \quad \wedge\, f^\tau \in \mathit{dom}(h(s(x_2^{\mathrm{a}}))) \\
 & & \{\mathbf{error}\} & \text{otherwise} \\
[\![x^{\mathrm{a}}.f^\tau := e^\tau]\!]\,(s,h) & = & \{(s, h[v.f^\tau \to ([\![e^\tau]\!]\,s)])\} & \text{if } s(x^{\mathrm{a}}) \in \mathit{dom}(h) \\
 & & \qquad\qquad \text{where } v = s(x^{\mathrm{a}}) & \quad \wedge\, f^\tau \in \mathit{dom}(h(s(x^{\mathrm{a}}))) \\
 & & \{\mathbf{error}\} & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcll}
\multicolumn{4}{l}{[\![x^{\mathrm{a}} := \mathtt{alloc}(f_1^{\tau_1}, \ldots, f_n^{\tau_n})]\!]\,(s,h) =} \\
 & & \{(s',h') \mid h' - \{v\} = h \text{ and } \mathit{dom}(h'(v)) = \{f_1^{\tau_1}, \ldots, f_n^{\tau_n}\} & \\
 & & \quad \text{and } s' = s[x^{\mathrm{a}} \to v] \text{ and } v \in \mathrm{Addr} \text{ and } v \notin \mathit{dom}(h) & \\
 & & \quad \text{and } h'(v)(f_i^{\tau_i}) \in \mathbb{Z} \text{ if } \tau_i = \mathrm{i} & \\
 & & \quad \text{and } h'(v)(f_i^{\tau_i}) \in \mathrm{Addr} \text{ if } \tau_i = \mathrm{a}\} & \\
[\![\mathtt{free}\; x^{\mathrm{a}}]\!]\,(s,h) & = & \{(s, h - \{s(x^{\mathrm{a}})\})\} & \text{if } s(x) \in \mathit{dom}(h) \cup \{\mathrm{nil}\} \\
 & & \{\mathbf{error}\} & \text{otherwise}
\end{array}
$$

Figure 7. Semantics of commands. $\mathit{dom}(g)$ indicates the domain of function g. The notation $g[x \to v]$ indicates the function that is the same as g, except that x is mapped to v. The notation $h[v_1.f \to v_2]$ indicates the heap that is the same as h except the record at v_1 maps field f to v_2. We write $h - X$ to indicate the function obtained by restricting the domain of h to $\mathit{dom}(h) - X$.

they need not be defined on all elements of the domain (different heap cells may contain different sets of fields).

$$\text{Records} \overset{\text{def}}{=} \text{Fields} \xrightarrow{\text{fin}}_{\tau} \text{Values}$$

$$h \in \text{Heaps} \overset{\text{def}}{=} (\text{Addr} - \{\mathbf{nil}\}) \xrightarrow{\text{fin}} \text{Records}$$

We write $[\![e^\tau]\!]$ for the meaning of an expression of type τ. This is a function of type Stores $\longrightarrow$ ($\mathbb{Z} \cup$ Addr $\cup$ Bool) where Bool $= \{\mathbf{true}, \mathbf{false}\}$. The definition of $[\![e^\tau]\!]$ is standard and omitted. As expected, $[\![e^{\mathrm{i}}]\!] \in \mathbb{Z}$ and $[\![e^{\mathrm{a}}]\!] \in$ Addr, etc.

The semantics of commands is given in Figure 7. The command $x := e$ is a standard assignment statement, $x := ?^{\mathrm{a}}$ and $x := ?^{\mathrm{i}}$ are non-deterministic assignments, $x_1 := x_2.f$ reads a value from a heap cell, and $x.f := e$ writes a value into a heap cell. The command $x := \mathtt{alloc}(f_1, \ldots, f_n)$ allocates a new uninitialized heap cell with fields $f_1, \ldots, f_n$. The command $\mathtt{free}\; x$ disposes of the heap cell at x.

Figure 8 gives the transition semantics of continuations. There are three types of states: intermediate states, final states, and goto states. Intermediate states have the form $\langle k, (s,h)\rangle$ where k is the current continuation and (s,h) is the current store and heap. Final states are either $\mathbf{error}$, which indicates that an error has occurred, or have the form $\mathbf{final}(s,h)$, which indicates that the program has terminated in the state (s,h). Goto states have the form $\mathbf{goto}(l,(s,h))$ and indicate that execution should continue from label l in state (s,h).

Note that at branch points, a non-determinstic choice is made among all branches whose condition is satisfied. There are no execution steps possible from a branch whose conditions are all false. This makes a single-condition branch equivalent to an "assume" statement, and we will write "$\mathtt{assume}(e);k$" as short-hand for "$\mathtt{branch}\; e \Rightarrow k\; \mathtt{end}$." Since our source programs are written in C, we will have the property that all branches in the original program are total (that is, the disjunction of the branch conditions is equivalent to *true*). This ensures that all non-final states in the original program can transition.

Figure 9 gives the semantics of programs. The program transitions whenever the current continuation can transition. If a state of the form $\mathbf{goto}(l,(s,h))$ is reached, then execution proceeds with the continuation at l.

SEMANTICS OF CONTINUATIONS

$$
\frac{(s',h') \in [\![c]\!]\,(s,h)}{\langle (c;k),(s,h)\rangle \rightsquigarrow \langle k,(s',h')\rangle}
\qquad
\frac{\mathbf{error} \in [\![c]\!]\,(s,h)}{\langle (c;k),(s,h)\rangle \rightsquigarrow \mathbf{error}}
$$

$$
\frac{[\![e_i]\!]\,s = \mathbf{true}}{\langle \mathtt{branch}\; \ldots, e_i \Rightarrow k_i, \ldots\; \mathtt{end}, (s,h)\rangle \rightsquigarrow \langle k_i,(s,h)\rangle}
$$

$$
\frac{}{\langle \mathtt{goto}\; l,(s,h)\rangle \rightsquigarrow \mathbf{goto}(l,(s,h))}
\qquad
\frac{}{\langle \mathtt{halt},(s,h)\rangle \rightsquigarrow \mathbf{final}(s,h)}
$$

$$
\frac{}{\langle \mathtt{abort},(s,h)\rangle \rightsquigarrow \mathbf{error}}
$$

Figure 8. Semantics of continuations.

$$
\frac{\langle k,(s,h)\rangle \rightsquigarrow \gamma}{\langle k,(s,h)\rangle \;{-P\!\!\to}\; \gamma}
\qquad
\frac{}{\mathbf{goto}(l,(s,h)) \;{-P\!\!\to}\; \langle P(l),(s,h)\rangle}
$$

Figure 9. Transition relation for program P. The variables γ represents an arbitrary execution state.

4.3 Separation Logic

Our method for producing numeric programs will involve reasoning about separation logic formulae. The syntax for formulae is given in Figure 10. The syntax of expressions is the same as in Figure 6 and is not repeated here. $\mathcal{D}$ is a set of identifiers that are used to refer to inductively-defined predicates. We discuss inductive predicate definitions in Section 4.4.

The semantics of formulae is also given in Figure 10. The spatial formula $\mathbf{emp}$ describes an empty heap. The formula $x \mapsto [\mathrm{f}_1 : e_1,\ \ldots,\ \mathrm{f}_n : e_n]$ describes a singleton heap where x points to a record whose f_1 field contains e_1 and so on. A heap satisfies $Q_1 * Q_2$ if it is a disjoint union of heaps h_1 and h_2 (denoted by $h_1 \uplus h_2$) such that h_1 satisfies Q_1 and h_2 satisfies Q_2. The formula $d(\vec{e})$, where $\vec{e}$ is a list of expressions, describes an instance of the inductive predicate d. We leave more details of inductive predicates (including their semantics) to Section 4.4.

SYNTAX

Inductive Predicates	d	$\in$	$\mathcal{D}$
Records	ρ	$::=$	$\epsilon \mid f^{\tau} : e^{\tau},\ \rho$
Spatial Predicates	Ξ	$::=$	$\mathbf{emp} \mid x \mapsto [\rho] \mid d(\vec{e})$
Separation Logic Formulae	Q	$::=$	$e^{\mathrm{b}} \mid \Xi \mid Q * Q \mid Q \wedge Q \mid Q \vee Q \mid \exists x.\, Q$

SEMANTICS

$(s,h) \models e^{\mathrm{b}}$	$\Leftrightarrow$	$[\![e^{\mathrm{b}}]\!]\, s = \mathbf{true}$
$(s,h) \models \mathbf{emp}$	$\Leftrightarrow$	$dom(h) = \{\}$
$(s,h) \models x \mapsto [\rho]$	$\Leftrightarrow$	$h = \{(s(x), R)\}$ where $R = \{(f_i, [\![e_i]\!]\, s) \mid f_i : e_i \in \rho\}$
$(s,h) \models Q_1 \wedge Q_2$	$\Leftrightarrow$	$(s,h) \models Q_1$ and $(s,h) \models Q_2$
$(s,h) \models Q_1 \vee Q_2$	$\Leftrightarrow$	$(s,h) \models Q_1$ or $(s,h) \models Q_2$
$(s,h) \models Q_1 * Q_2$	$\Leftrightarrow$	$h = h_1 \uplus h_2$ and $(s,h_1) \models Q_1$ and $(s,h_2) \models Q_2$
$(s,h) \models \exists x.\, Q$	$\Leftrightarrow$	there exists $v \in$ Values such that $(s[x \to v], h) \models Q$

Figure 10. Syntax and semantics of formulae. The relation $h = h_1 \uplus h_2$ holds iff $dom(h_1) \cap dom(h_2) = \emptyset$ and $h = h_1 \cup h_2$. The notation $f_i : e_i \in \rho$ indicates that $\rho = \ldots, f_i : e_i, \ldots$

4.4 Defining Inductive Pointer Structures

Unbounded pointer structures in our system are described inductively using definitions of the following form. We use vector notation (e.g. $\vec{v}$) to denote a list of variables and refer to each $Q_{i,j}$ as a *case* of predicate d_i.

$$\begin{array}{lcl} d_1(\vec{v_1}) & \equiv & Q_{1,1} \vee \ldots \vee Q_{1,m_1} \\ \ldots & & \\ d_n(\vec{v_n}) & \equiv & Q_{n,1} \vee \ldots \vee Q_{n,m_n} \end{array}$$

We require that all variables in $\vec{v_i}$ are distinct and that if $d \in \mathcal{D}$ appears in any $Q_{i,j}$ then $d \in \{d_1, \ldots, d_n\}$. Also, $fv(Q_{i,j}) \subseteq \vec{v_i}$. This specifies a set of mutually inductive predicates. We write $\mathrm{unroll}^n(Q)$ for the bounded expansion of formula Q. This is defined such that $\mathrm{unroll}^n(Q)$ is Q with each instance of $d_i(\vec{e})$ replaced by

$$(\mathrm{unroll}^{(n-1)}(Q_{i,1}[\vec{e}/\vec{v_i}])) \vee \ldots \vee (\mathrm{unroll}^{(n-1)}(Q_{i,m_i}[\vec{e}/\vec{v_i}]))$$

where $Q[\vec{e}/\vec{v_i}]$ is the simultaneous substitution of each element of $\vec{e}$ for the corresponding element of $\vec{v_i}$. The semantics of a formula Q containing inductive predicates is then defined such that $(s,h) \models Q$ iff there exists an n such that $(s,h) \models \mathrm{unroll}^n(Q)$.

As an example, consider the following definition of a doubly-linked list segment with length n starting at heap cell *first* and ending at *last*.

$$\begin{array}{l} \mathrm{dll}(n, prev, first, last, next) \equiv \\ \qquad \mathbf{emp} \wedge n = 0 \wedge first = next \wedge last = prev \\ \quad \vee\ (\exists z.\ (first \mapsto [\mathrm{prev} : prev, \mathrm{next} : z]) * \\ \qquad\qquad\qquad \mathrm{dll}(n-1, first, z, last, next)) \wedge n > 0 \end{array}$$

This states that there are two possible cases for a segment of length n. Either $n = 0$, in which case the list is empty, or $n > 0$, in which case there is a cell at the head of the list and a tail of length $n - 1$.

5. Instrumented Programs and Numeric Abstractions

The translation from heap-manipulating programs to numeric programs proceeds via an intermediate form that we call *instrumented programs*. These have the same structure as the original program and include the original program commands, as well as new commands that update a class of *instrumentation variables*. These instrumentation variables track changes to the values of numeric counts, such as the size of a data structure, during execution of the program. The new commands are added to the instrumented program as a proof of memory safety is constructed and make use of the intermediate results of this safety analysis. Once the instrumented program has been constructed, the numeric abstraction is extracted from it by a simple syntax-directed translation. The end result is that the numeric program simulates the original program and thus is a sound abstraction for both safety and liveness properties. In this section, we discuss the theory of instrumented programs. In Section 7 we describe how we can automate the generation of instrumented programs and thus numeric abstractions.

5.1 Instrumented Programs

Figures 11 and 12 give the rules for producing an instrumented version $\widehat{P}$ of an original program P. The instrumented program may include commands that reference variables that are not present in the original program. We refer to such variables as *instrumentation variables* and they play a role similar to that of auxiliary variables in program logics for concurrency [27].

Figure 12 defines a judgment $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ which states that under invariants Γ, $\widehat{P}$ is an instrumented version of P with instrumentation variables in V. The *invariant function* Γ is a mapping from labels to separation logic formulae that specifies the invariants associated with program labels. The notation $\{Q\}\ c\ \{Q'\}$ specifies a partial correctness triple and indicates that if $(s,h) \models Q$ and $((s',h') \in [\![c]\!]\,(s,h))$ then $(s',h') \models Q'$. Types are omitted to avoid cluttering the rules, but instrumented programs and continuations must satisfy the typing rules given in Figure 6.

Note that the property of being a valid instrumentation is defined with respect to program invariants Γ and, in the case of continuations, with respect to a precondition Q. If we view the construction of a proof in the system given in Figure 11 as proceeding in a bottom-up manner, then instrumentation proceeds in lock-step with the derivation of a partial correctness proof of the program. The rules COMMAND and BRANCH tell us how to update the precondition to reflect the results of executing an existing command, and rules INST-ASSIGN, INST-ASSUME, INST-DISJ and INST-EXISTS tell us which new commands may be added. The invariants in Γ and those produced during the proving of the continuations that make up the program ensure that the instrumentation commands are consistent with the semantics of the program and provide a link between the original program and the numeric program that we eventually produce. More details on this connection can be found in the statement of soundness in Section 6.

A key difference between this approach to command insertion and the auxiliary variable approach lies with the EXISTS rule. This rule tells us that if we insert an assignment $x := ?$, then we can remove an existential quantifier on x. This may seem odd, since $\{\exists x.\ Q\}\ x := ?\ \{Q\}$ is not a valid partial correctness triple. However, inserting such a command and reasoning from the unquantified formula is sound because our soundness result is based on simulation. To maintain soundness, we must show that if the original program can take a step, then there exists a step in the instrumented program that takes us to a related state. The fact that the semantics of $x := ?$ includes all possible updates to x allows us to find such a state. More details are given in Section 6.

HALT
$$\frac{}{\Gamma \vdash \{Q\}\,\texttt{halt} \blacktriangleright_V \texttt{halt}}$$

ABORT
$$\frac{}{\Gamma \vdash \{Q\}\,\texttt{abort} \blacktriangleright_V \texttt{abort}}$$

GOTO
$$\frac{\Gamma(l) = Q}{\Gamma \vdash \{Q\}\,\texttt{goto}\; l \blacktriangleright_V \texttt{goto}\; l}$$

COMMAND
$$\frac{\{Q\}\; c\; \{Q'\} \qquad \Gamma \vdash \{Q'\}\, \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\; (c; \widehat{k}) \blacktriangleright_V c; k}$$

BRANCH
$$\frac{\forall i.\ (\Gamma \vdash \{Q \wedge e_i\}\, \widehat{k_i} \blacktriangleright_V k_i)}{\Gamma \vdash \{Q\}\,\texttt{branch}\; \ldots, e_i \Rightarrow \widehat{k_i}, \ldots\; \texttt{end} \blacktriangleright_V \texttt{branch}\; \ldots, e_i \Rightarrow k_i, \ldots\; \texttt{end}}$$

FALSE
$$\frac{}{\Gamma \vdash \{\mathit{false}\}\,\texttt{halt} \blacktriangleright_V k}$$

INST-ASSIGN
$$\frac{\{Q\}\; x := e\; \{Q'\} \qquad \Gamma \vdash \{Q'\}\, \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\; (x := e; \widehat{k}) \blacktriangleright_V k}\; x \in V$$

INST-DISJ
$$\frac{\Gamma \vdash \{Q_1\}\, \widehat{k}_1 \blacktriangleright_V k \qquad \Gamma \vdash \{Q_2\}\, \widehat{k}_2 \blacktriangleright_V k}{\Gamma \vdash \{Q_1 \vee Q_2\}\,\texttt{branch}\; \mathit{true} \Rightarrow \widehat{k}_1, \mathit{true} \Rightarrow \widehat{k}_2\; \texttt{end} \blacktriangleright_V k}$$

INST-ASSUME
$$\frac{Q \Rightarrow e^{\mathrm{b}} \qquad \Gamma \vdash \{Q\}\, \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\,\texttt{branch}\; e^{\mathrm{b}} \Rightarrow \widehat{k}\; \texttt{end} \blacktriangleright_V k}$$

INST-EXISTS
$$\frac{\Gamma \vdash \{Q\}\, \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{\exists x.\ Q\}\; (x := ?; \widehat{k}) \blacktriangleright_V k}\; x \in V$$

STRENGTHENING
$$\frac{Q \Rightarrow Q' \qquad \Gamma \vdash \{Q'\}\, \widehat{k} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\, \widehat{k} \blacktriangleright_V k}$$

Figure 11. Rules for establishing that $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$, read "under precondition Q, with label invariants Γ, the continuation $\widehat{k}$ is an instrumented version of k with instrumentation variables V."

$$\frac{\mathit{dom}(\widehat{P}) = \mathit{dom}(P) \qquad \mathit{fv}(P) \cap V = \emptyset \\ \forall l \in \mathit{dom}(P).\ (\Gamma \vdash \{\Gamma(l)\}\, \widehat{P}(l) \blacktriangleright_V P(l))}{\Gamma \vdash \widehat{P} \blacktriangleright_V P}$$

Figure 12. Rule for proving that $\widehat{P}$ is an instrumented version of P.

$$
\begin{array}{l}
\mathrm{numabs}_V(\widehat{k}) = \widehat{k} \qquad \text{if } \widehat{k} \in \{\texttt{abort}, \texttt{halt}, \texttt{goto}\; l\} \\
\mathrm{numabs}_V(c; \widehat{k}) = c; (\mathrm{numabs}_V(\widehat{k})) \\
\qquad \text{if } c \text{ is } x := e \text{ or } x := ? \text{ and } \mathit{fv}(x, e) \subseteq V \\
\mathrm{numabs}_V(c; \widehat{k}) = x := ?; (\mathrm{numabs}_V(\widehat{k})) \\
\qquad \text{if } c \text{ is } x := e \text{ and } x \in V \text{ and } \mathit{fv}(e) \not\subseteq V \\
\qquad \text{or } c \text{ is } x := y.f \text{ and } x \in V \\
\mathrm{numabs}_V(c; \widehat{k}) = \mathrm{numabs}_V(\widehat{k}) \\
\qquad \text{otherwise} \\
\mathrm{numabs}_V(\texttt{branch}\; e_1 \Rightarrow \widehat{k_1}, \ldots, e_n \Rightarrow \widehat{k_n}\; \texttt{end}) = \\
\qquad \texttt{branch}\; \mathrm{numexp}_V(e_1) \Rightarrow \mathrm{numabs}_V(\widehat{k_1}), \ldots, \\
\qquad\qquad \mathrm{numexp}_V(e_n) \Rightarrow \mathrm{numabs}_V(\widehat{k_n})\; \texttt{end} \\
\qquad \text{where } \mathrm{numexp}_V(e) = \begin{cases} e & \text{if } \mathit{fv}(e) \subseteq V \\ \mathit{true} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 13. Definition of the function $\mathrm{numabs}_V(\widehat{k})$, which converts an instrumented program to a numeric abstraction with free variables contained in V.

5.2 Numeric Programs

Numeric programs are the projection of the instrumented program onto a subset of the integer-valued variables. In Figure 13 we define a function $\mathrm{numabs}_V(\widehat{k})$ that, given an instrumented program $\widehat{k}$ and a set of variables V, produces the numeric abstraction of $\widehat{k}$ over variables in V (which must all be of integer type). Because they involve only integer-valued variables, numeric abstractions can be analyzed by a tool without support for the heap and, as shown in Section 6, a proof of safety or termination from the non-heap-aware tool guarantees safety or termination of the original program.

5.3 Common Reasoning Patterns

We now demonstrate how the rules in Figure 11 may be used to insert code that expresses various facts about the behavior of numeric properties of data structures.

Deterministic Size Changes Suppose we have the following definition of singly-linked list segments.

$$
\begin{array}{l}
\mathit{ls}(n, \mathit{first}, \mathit{tail}) \equiv \\
\qquad \mathbf{emp} \wedge n = 0 \wedge \mathit{first} = \mathit{tail} \\
\qquad \vee\, (\exists z.\ (\mathit{first} \mapsto [\mathrm{next} : z]) * \mathit{ls}(n-1, z, \mathit{tail})) \wedge n > 0
\end{array}
$$

and execute the code given below.

$$
\begin{array}{l}
1 : \texttt{branch}\; x \neq \mathrm{nil} \Rightarrow x := x.\mathrm{next};\ \texttt{goto}\; 1, \\
\qquad\qquad x = \mathrm{nil} \Rightarrow \texttt{halt}\; \texttt{end}
\end{array}
$$

An invariant of this code at label 1 is $\exists n_1, n_2, x'.\ \mathit{ls}(n_1, x', x) * \mathit{ls}(n_2, x, \mathrm{nil})$. In order to track how the sizes of the segments are changing, we can generate an instrumented program for the code above. Let $\Gamma(1) = \exists x'.\ \mathit{ls}(n_1, x', x) * \mathit{ls}(n_2, x, \mathrm{nil})$. Then the code below is an instrumented version of the code above with instrumentation variables n_1, n_2 (the assignments to n_1 and n_2 are added with the INST-ASSIGN rule).

$$
\begin{array}{l}
1 : \texttt{branch}\; x \neq \mathrm{nil} \Rightarrow x := x.\mathrm{next};\ n_1 := n_1 + 1; \\
\qquad\qquad\qquad n_2 := n_2 - 1;\ \texttt{goto}\; 1, \\
\qquad x = \mathrm{nil} \Rightarrow \texttt{halt}\; \texttt{end}
\end{array}
$$

Note that the existential quantification is dropped in the invariant used for the instrumented program. This is possible because we are now updating n_1 and n_2 in the body of the loop. Viewed another way, it is by committing to an invariant in which n_1 and n_2 are unquantified that we are forced to write the appropriate updates to n_1 and n_2 in the body (if we update n_1 or n_2 incorrectly, we will not be able to reestablish $\Gamma(1)$ following execution of the code).

Non-deterministic Size Changes Suppose we have the following definition of a binary tree.

$$
\begin{array}{l}
\mathit{tree}(h, r) \equiv (h = 0 \wedge r = \mathrm{nil}) \\
\qquad \vee\, (h > 0 \wedge \exists h_1, h_2.\ (h_1 < h) \wedge (h_2 < h) \wedge \\
\qquad\qquad \exists \mathit{lc}, \mathit{rc}.\ r \mapsto [\mathrm{left} : \mathit{lc}, \mathrm{right} : \mathit{rc}] \\
\qquad\qquad\qquad * \mathit{tree}(h_1, \mathit{lc}) * \mathit{tree}(h_2, \mathit{rc}))
\end{array}
$$

If we now consider code for descending through the tree, we can obtain update commands similar to those obtained for the linked list example above. However, since h here is a bound on the height of the tree, and the height of a subtree is not a deterministic function of the height of the containing tree, we generate non-deterministic update commands. The code for performing the descent is given below. We have marked with $\{Q\}$ a location of interest during creation of the instrumented program.

```
1 : branch $r \neq \mathrm{nil} \Rightarrow \{Q\}$branch $true \Rightarrow r := r.\mathrm{left}$;
                                        goto 1,
                               $true \Rightarrow r := r.\mathrm{right}$;
                                        goto 1 end
        $r = \mathrm{nil} \Rightarrow$ halt end
```

Let $\Gamma(1) = true * (tree(h, r))$ (where $true$ is used to capture the part of the heap no longer below r in the tree) and let Q' be the formula $(h > 0 \wedge (h_1 < h) \wedge (h_2 < h) \ \wedge \exists lc, rc.\ r \mapsto [\mathrm{left} : lc, \mathrm{right} : rc] * tree(h_1, lc) * tree(h_2, rc))$. Then setting $Q = \exists h_1, h_2.\ Q'$ gives us a valid instance of the BRANCH rule (we would also need to generate a corresponding precondition for `halt`). Applying the EXISTS rule then lets us insert the command $h_1 := ?$ and begin working from the state $\exists h_2.\ Q'$. We then do the same for h_2, obtaining $h_2 := ?$ and Q'. Finally, we can use the INST-ASSUME rule to insert $\texttt{assume}(h_1 < h \wedge h_2 < h)$.[1] Putting this all together with an update command for h (added via the INST-ASSIGN rule), we get the instrumented program given below.

```
1 : branch $r \neq \mathrm{nil} \Rightarrow h_1 := ?;\ h_2 := ?;$
                    assume$(h_1 < h \wedge h_2 < h)$;
                    branch $true \Rightarrow r := r.\mathrm{left}$;
                                     $h = h_1$; goto 1,
                           $true \Rightarrow r := r.\mathrm{right}$;
                                     $h = h_2$; goto 1 end
        $r = \mathrm{nil} \Rightarrow$ halt end
```

Non-deterministic assignment and "assume" statements are not generally present in standard C source code, but they are implemented in most program analysis tools (for example, BLAST [20] has the `__BLAST_NONDET` construct). We can use these facilities to format the numeric code we generate for particular tools.

Branch Condition Translation Let us return to the linked-list example from above. The instrumented code that we generated summarized how n_1 and n_2 were changing during each iteration. This is sufficient, for example, to prove that the quantity $n_1 + n_2$ is invariant at location 1. However, we did not add any commands to indicate how n_1 and n_2 influence the truth of the branch conditions. Thus, when we extract the numeric program, we will get the following, which we cannot prove terminates.

```
1 : branch $true \Rightarrow x := x.next;\ n_1 := n_1 + 1;$
                 $n_2 := n_2 - 1$; goto 1,
        $true \Rightarrow$ halt end
```

To obtain a more precise numeric abstraction, we need to replace the branch on $x = \mathrm{nil}$ with a branch involving n_1 and n_2. To accomplish this, we can use the INST-ASSUME rule to insert an assumption on n_2.[1] The final instrumented program then becomes.

```
1 : branch $x \neq \mathrm{nil} \Rightarrow$ assume$(n_2 > 0)$; $x := x.next$;
                 $n_1 := n_1 + 1$; $n_2 := n_2 - 1$; goto 1,
        $x = \mathrm{nil} \Rightarrow$ assume$(n_2 = 0)$; halt end
```

[1] Recall that $\texttt{assume}(e^{\mathrm{b}})\,;k$ is an abbreviation for $\texttt{branch}\ e^{\mathrm{b}} \Rightarrow k\ \texttt{end}$.

In this case, we have an exact translation of the branch condition into an assume statement involving instrumentation variables. In general, however, this approach let us records assumptions that over-approximate the original program branch.

Inserting Branches In addition to adding assumptions after existing branch cases, we can also insert branches at arbitrary points in the instrumented program. If we can show that our precondition Q implies $(Q_1 \wedge e_1) \vee (Q_2 \wedge e_2)$ then we can use INST-DISJ and INST-ASSUME to insert a branch of the form $\texttt{branch}\ true \Rightarrow (\texttt{assume}(e_1)\,;\widehat{k_1}),\ true \Rightarrow (\texttt{assume}(e_2)\,;\widehat{k_2})\ \texttt{end}$. By our semantics, this is equivalent to $\texttt{branch}\ e_1 \Rightarrow \widehat{k_1}, e_2 \Rightarrow \widehat{k_2}\ \texttt{end}$ (ignoring repeated states). Using this equivalence, we can codify the previous instrumentation pattern as the following derived rule.

INST-BRANCH

$$\frac{Q \Rightarrow (Q_1 \wedge e_1) \vee (Q_2 \wedge e_2) \qquad \Gamma \vdash \{Q_1 \wedge e_1\}\ \widehat{k_1} \blacktriangleright_V k \qquad \Gamma \vdash \{Q_2 \wedge e_2\}\ \widehat{k_2} \blacktriangleright_V k}{\Gamma \vdash \{Q\}\ \texttt{branch}\ e_1 \Rightarrow \widehat{k_1}, e_2 \Rightarrow \widehat{k_2}\ \texttt{end} \blacktriangleright_V k}$$

As branches with multiple cases can be encoded using binary branches, we can also derive an n-ary version of this rule.

Since $Q \Rightarrow (Q \wedge e) \vee (Q \wedge \neg e)$ is always derivable, we can use this to insert arbitrary case splits into the instrumented program. We can also use it to insert splits based on which case of an inductive definition holds. For example, suppose we have the precondition $ls(n, x, y)$ and a branch on whether $x = y$ or $x \neq y$. If $n = 0$ then we know $x = y$. If $n > 0$ we cannot conclude anything about x and y as they could still be equal if the list is cyclic. We can make this relationship between $n = 0$ and $x = y$ apparent by first case splitting on n using the derived rule above and then using FALSE to prune the branch where $n = 0$ and $x \neq y$. We obtain the program below, where we have written $\{false\}$ to highlight the location of the inconsistent case.

```
branch $n_2 > 0 \Rightarrow$
           branch $x \neq y \Rightarrow \ldots$,
                  $x = y \Rightarrow \ldots$ end,
       $n_2 = 0 \Rightarrow$
           branch $x \neq y \Rightarrow \{false\}$ halt,
                  $x = y \Rightarrow \ldots$ end
   end
```

6. Soundness

In this section, we prove the main soundness result, which is that the numeric program simulates the original program. We then state several consequences of this theorem as it relates to particular classes of program properties. We use the notation $s = s' \mod V$ to indicate that $\forall x.\ x \notin V \Rightarrow s(x) = s'(x)$.

Definition 1. *Let $R_1^{V,\Gamma}$ be the least relation on execution states satisfying the following.*

$$\begin{array}{rcll}
\langle k, (s, h)\rangle & R_1^{V,\Gamma} & \langle \widehat{k}, (\widehat{s}, h)\rangle & \textit{iff}\ \ s = \widehat{s} \mod V \\
& & & \textit{and}\ \exists Q.\ \big(\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k\big) \\
& & & \wedge\ \big((\widehat{s}, h) \models Q\big) \\
\mathbf{goto}(l, (s, h)) & R_1^{V,\Gamma} & \mathbf{goto}(l, (\widehat{s}, h)) & \textit{iff}\ \ s = \widehat{s} \mod V \\
& & & \textit{and}\ (\widehat{s}, h) \models \Gamma(l) \\
\mathbf{final}(s, h) & R_1^{V,\Gamma} & \mathbf{final}(\widehat{s}, h) & \textit{iff}\ \ s = \widehat{s} \mod V \\
\mathbf{error} & R_1^{V,\Gamma} & \mathbf{error} &
\end{array}$$

Theorem 1. *Suppose $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ and choose any l_0 such that $l_0 \in dom(P)$. Let $(s_0, h_0) \models \Gamma(l_0)$. Then we have that $\widehat{P}$ with initial state $\langle \widehat{P}(l_0), (s_0, h_0)\rangle$ stuttering simulates P with* *initial state $\langle P(l_0), (s_0, h_0)\rangle$ and $R_1^{V,\Gamma}$ is the simulation relation that witnesses this.*

Proof. We use the framework of well-founded simulations from [25] for the proof. We first show that initial states are related. We have $s_0 = s_0 \mod V$ and $(s_0, h_0) \models \Gamma(l_0)$. Since we have $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ and there is only one rule for establishing this, we have $\Gamma \vdash \{\Gamma(l_0)\}\ \widehat{P}(l_0) \blacktriangleright_V P(l_0)$ from the premises of that rule (see Figure 12). This completes the proof that $\langle P(l_0), (s_0, h_0)\rangle$ $R_1^{V,\Gamma} \langle \widehat{P}(l_0), (\widehat{s}_0, h_0)\rangle$.

For non-initial states, suppose $\gamma R_1^{V,\Gamma} \widehat{\gamma}$ and $\gamma \xrightarrow{P} \gamma'$. We will show that one of the following holds.

1. $\widehat{\gamma} \xrightarrow{\widehat{P}} \widehat{\gamma}'$ and $\gamma' R_1^{V,\Gamma} \widehat{\gamma}'$
2. $\widehat{\gamma} \xrightarrow{\widehat{P}} \widehat{\gamma}'$ and $\gamma R_1^{V,\Gamma} \widehat{\gamma}'$

There are additional well-foundedness conditions associated with condition 2 which we address when we discuss those cases.

The only states that can transition are intermediate states and goto states. We address goto states first. Suppose $\gamma = \textbf{goto}(l, (s, h))$. Then, since $\gamma R_1^{V,\Gamma} \widehat{\gamma}$, we have $\widehat{\gamma} = \textbf{goto}(l, (\widehat{s}, h))$ and $s = \widehat{s} \mod V$ and $(\widehat{s}, h) \models \Gamma(l)$. Our program semantics gives us $\gamma \xrightarrow{P} \langle P(l), (s, h)\rangle$ and $\widehat{\gamma} \xrightarrow{\widehat{P}} \langle \widehat{P}(l), (\widehat{s}, h)\rangle$. Since we have $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ and there is only one rule for establishing this, we have $\Gamma \vdash \{\Gamma(l)\}\ \widehat{P}(l) \blacktriangleright_V P(l)$ from the premises of that rule. These are all the conditions required to establish that the target states are related.

We now consider the intermediate states. Let $\gamma = \langle k, (s, h)\rangle$. Since $\gamma R_1^{V,\Gamma} \widehat{\gamma}$ we then have $\widehat{\gamma} = \langle \widehat{k}, (\widehat{s}, h)\rangle$ for some $\widehat{k}, \widehat{s}$ and that there exists a Q satisfying the following

$$s = \widehat{s} \mod V \tag{1}$$

$$(\widehat{s}, h) \models Q \tag{2}$$

$$\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k \tag{3}$$

The proof for these cases will proceed by induction on the derivation of $\Gamma \vdash \{Q\}\ \widehat{k} \blacktriangleright_V k$. The HALT and ABORT cases are straightforward. For GOTO we have $\gamma' = \textbf{goto}(l, (s, h))$ and $\widehat{\gamma}' = \textbf{goto}(l, (\widehat{s}, h))$. To show that these states are related, we must show $(\widehat{s}, h) \models \Gamma(l)$, which we have as a premise of the rule.

For COMMAND we have $k = (c; k')$ and $\widehat{k} = (c; \widehat{k}')$. By virtue of the fact that the same command is being executed, and that $s = \widehat{s} \mod V$ and c does not contain variables in V, we have that the heaps in the post-states are the same and the stores are equal mod V. That $(\widehat{s}', h) \models Q'$ follows from our premise $\{Q\}\ c\ \{Q'\}$. For BRANCH we have from $\gamma \xrightarrow{P} \gamma'$ that $[\![e_i]\!]\, s = \textbf{true}$ for some e_i. Since $s = \widehat{s} \mod V$ and $fv(e_i) \cap V = \emptyset$, we have $(\widehat{s}, h) \models e_i$ which implies that $\widehat{P}$ can match the step.

The FALSE case holds vacuously, since our assumption (2) becomes $(\widehat{s}, h) \models \mathit{false}$, which cannot hold. For STRENGTHENING, we have that $Q \Rightarrow Q'$ so $(\widehat{s}, h) \models Q$ implies $(\widehat{s}, h) \models Q'$, allowing us to apply the inductive hypothesis to $\Gamma \vdash \{Q'\}\ \widehat{k} \blacktriangleright_V k$ and thus obtain the result.

We now turn to the cases in category 2, where $\widehat{\gamma} \xrightarrow{\widehat{P}} \widehat{\gamma}'$ and $\gamma R_1^{V,\Gamma} \widehat{\gamma}'$. These are INST-ASSIGN, INST-DISJ, INST-ASSUME, and INST-EXISTS. In addition to our usual proof obligations, in these cases we must also show that some well-founded measure decreases following the transition. If we take the size of $\widehat{\gamma}'$ to be the size of the term representing the current continuation, then this will be an appropriate measure.

We begin with INST-ASSUME. We must first show that $\widehat{k} = \texttt{branch}\ e^{\mathrm{b}} \Rightarrow \widehat{k}'\ \texttt{end}$ allows $\widehat{\gamma}$ to transition. This follows from the premise $Q \Rightarrow e^{\mathrm{b}}$, which implies $(\widehat{s}, h) \models e^{\mathrm{b}}$ and thus $[\![e^{\mathrm{b}}]\!]\, \widehat{s} = \textbf{true}$, which is the required condition to establish $\langle \texttt{branch}\ e^{\mathrm{b}} \Rightarrow \widehat{k}'\ \texttt{end}, (\widehat{s}, h)\rangle \xrightarrow{\widehat{P}} \langle \widehat{k}', (\widehat{s}, h)\rangle$. That $\gamma R_1^{V,\Gamma} \langle \widehat{k}', (\widehat{s}, h)\rangle$ is then straightforward.

For the other cases, it is easy to see that a transition $\widehat{\gamma} \xrightarrow{\widehat{P}} \widehat{\gamma}'$ exists. Let $\widehat{\gamma}' = \langle \widehat{k}', (\widehat{s}', h)\rangle$ (as it will have this form in all remaining cases). For INST-ASSIGN, we have as a premise $\{Q\}\ x := e\ \{Q'\}$, which implies $(\widehat{s}', h) \models Q'$. From (1) and the side condition $x \in V$ we have $s = \widehat{s}' \mod V$. Our second premise then provides the last condition required to establish $\gamma R_1^{V,\Gamma} \widehat{\gamma}'$.

For INST-DISJ, we must show that one of the branches allows the post-state to be related to γ. Since $\widehat{s}$ and h are unchanged by branch execution, we need only show that there is a Q satisfying conditions (2) and (3). We have from $\gamma R_1^{V,\Gamma} \widehat{\gamma}$ that $(\widehat{s}, h) \models Q_1 \vee Q_2$, which implies $(\widehat{s}, h) \models Q_1$ or $(\widehat{s}, h) \models Q_2$. In either case we have $\Gamma \vdash \{Q_i\}\ \widehat{k}_i \blacktriangleright_V k$ and $\widehat{\gamma} \xrightarrow{\widehat{P}} \langle \widehat{k}_i, (\widehat{s}, h)\rangle$, which implies $\gamma R_1^{V,\Gamma} \widehat{\gamma}'$.

The final case is INST-EXISTS. In this case, we must show that there is some $\widehat{s}$ such that $\langle (x := ?\,; \widehat{k}'), (\widehat{s}, h)\rangle \rightsquigarrow \langle \widehat{k}', (\widehat{s}', h)\rangle$ and $(\widehat{s}', h) \models Q$. We have that $(\widehat{s}, h) \models \exists x.\ Q$ from our assumptions. This implies that $(\widehat{s}', h) \models Q$ for some $\widehat{s}'$ that is the same as $\widehat{s}$ except at x. The semantics of $x := ?$ ensures that $(\widehat{s}', h)$ is in the set $[\![x := ?]\!]\, (\widehat{s}, h)$ and thus we have our result. □

Since the numeric program does not involve heap access commands, its semantics can be given entirely in terms of commands' effect on the store. The interpretation of commands is then as given in Figure 7 but with h, which is invariant for non-heap commands, omitted. Intermediate states of the original program and instrumented program are then tuples $\langle k, (s, h)\rangle$ while concrete states of the numeric program have the form $\langle k, s\rangle$.

Definition 2. *Let $s =_{V'} \widehat{s}$ hold iff $\forall x \in V'.\ s(x) = \widehat{s}(x)$. Let $R_2^{V'}$ be the least relation on states satisfying the following*

$$\begin{array}{rcll}
\langle k, (s, h)\rangle & R_2^{V'} & \langle k', s'\rangle & \textit{iff} \quad s =_{V'} s' \\
\textbf{goto}(l, (s, h)) & R_2^{V'} & \textbf{goto}(l, s') & \textit{iff} \quad s =_{V'} s' \\
\textbf{final}(s, h) & R_2^{V'} & \textbf{final}(s') & \textit{iff} \quad s =_{V'} s' \\
\textbf{error} & R_2^{V'} & \textbf{error} &
\end{array}$$

Theorem 2. *Suppose $P' = numabs_{V'}(\widehat{P})$ and choose any l_0 such that $l_0 \in dom(\widehat{P})$. Let $(\widehat{s}_0, \widehat{h}_0) \models \Gamma(l_0)$. Then we have that P' with initial state $\langle P'(l_0), \widehat{s}_0\rangle$ simulates $\widehat{P}$ with initial state $\langle \widehat{P}(l_0), (\widehat{s}_0, \widehat{h}_0)\rangle$ and $R_2^{V'}$ is the simulation relation that witnesses this.*

Proof. The proof proceeds by induction on the structure of $\widehat{P}$. There is a case for each branch of the definition of $\mathrm{numabs}_{V'}(\widehat{k})$. We need only consider commands c that modify variables in V'. We have that $s_1 =_{V'} s_1'$ and must show that after executing c we have $s_2 =_{V'} s_2'$ where s_2 and s_2' are the stores in the post-states. For $c = (x := y.f)$ we have a numeric command of $x := ?$, the semantics of which includes all possible writes into x. For $c = (x := e)$ we have either $x := ?$ or $x := e$ as the numeric command. We have $x := e$ only when $fv(e) \subseteq V'$, in which case $s_1 =_{V'} s_1'$ implies $[\![e]\!]\, s_1 = [\![e]\!]\, s_1'$, which ensures that the same value is written into x in each case.

For branches, the reasoning is similar in that the branch condition is either *true*, which is an over-approximation of any branch, or the condition is e with $fv(e) \subseteq V'$ which implies that the condition evaluates to the same value in both the instrumented and numeric states. The condition that labels in the goto states are the same follows from the fact that `goto` statements are carried over to the numeric program unchanged. □

Corollary 1. *If $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ and $P' = numabs_{V'}(\widehat{P})$ and P' terminates, then P terminates when started from any state $\langle P(l_0), (s,h)\rangle$ such that $(s,h) \models \Gamma(l_0)$.*

Proof. This follows directly from the simulation results above. Transitivity of simulation gives us that P' simulates P with precondition $\Gamma(l_0)$, which implies that if P' contains only finite traces then P must contain only finite traces. □

The following corollary relates safety properties of the numeric program to safety properties of the original program by relating the program invariants. We use the notation $\Gamma \vdash P$ to mean that Γ specifies invariants of P. That is, $dom(\Gamma) = dom(P)$ and for all $l \in dom(P)$ and for all s, h, if $(s,h) \models \Gamma(l)$ and $\langle P(l), (s,h)\rangle \multimap_P\!\!\to^* \mathbf{goto}(l', (s', h'))$ then $(s', h') \models \Gamma(l')$. We will also write $\exists V.\ (\Gamma \wedge \Gamma')$ to denote the function Γ'' with the same domain as Γ but with $\Gamma''(l) = \big(\exists V.\ (\Gamma(l) \wedge \Gamma'(l)\big)$.

Corollary 2. *If $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ and $P' = numabs_{V'}(\widehat{P})$ and $\Gamma' \vdash P'$ and $fv(\Gamma') \subseteq V'$, then $\exists V.\ (\Gamma \wedge \Gamma') \vdash P$.*

Proof. This follows from the details of the simulation relations involved. Let $\widehat{P}$ be the instrumented program connecting P and P'. The simulation relation tells us that if a state $\mathbf{goto}(l, (s,h))$ is reachable in a trace of P starting from some $\langle P(l_0), (s_0, h_0)\rangle$ such that $(s_0, h_0) \models \exists V.\ \Gamma(l_0)$, then the state is $R_1^{V,\Gamma}$-related to a state $\mathbf{goto}(l, (\widehat{s}, h))$ in a trace of $\widehat{P}$ which is then $R_2^{V'}$-related to a state $\mathbf{goto}(l, s')$ in a trace of P'. That $\mathbf{goto}(l, s')$ is reachable in P' and $\Gamma' \vdash P'$ implies that $s' \models \Gamma'(l)$. The conditions of the relation $R_2^{V'}$ and $fv(\Gamma') \subseteq V'$ then gives us that $(\widehat{s}, h) \models \Gamma'(l)$ since s' and $\widehat{s}$ agree on the values of the variables in V'. Finally, the $R_1^{V,\Gamma}$ relation tells us that $(\widehat{s}, h) \models \Gamma(l)$ which implies $(\widehat{s}, h) \models \Gamma(l) \wedge \Gamma'(l)$. That $s = \widehat{s} \mod V$ then implies that $(s,h) \models \exists V.\ \Gamma(l) \wedge \Gamma'(l)$. □

In fact, the simulation relations involved ensure that if $\Gamma \vdash \widehat{P} \blacktriangleright_V P$ and $P' = \mathrm{numabs}_{V'}(\widehat{P})$ then each trace of P starting from $\langle P(l_0), (s,h)\rangle$ with $(s,h) \models \Gamma(l_0)$ is stuttering equivalent to a trace of P', where the equivalence identifies states that agree on $V' - V$. The theory of stuttering equivalence developed in [8] then gives us that P' is a sound abstraction of these traces of P for all LTL$_{-\mathrm{X}}$ properties over variables in $V' - V$. In our implementation, V' is chosen such that this difference includes integer variables of interest from the original program, such as variables whose values we would like to bound, or for which we want to compute invariants.

7. Instrumentation Analysis

We will now describe how we have automated the production of instrumented programs and thus numeric abstractions. As partial correctness invariants describing the heap are a required part of the proof that one program is an instrumentation of another, a shape analysis will be at the center of our approach and we will only be able to automatically produce instrumentations of programs for which the shape analysis succeeds. The shape analysis we use is of the form described in [16] and [23]. It has been implemented as an extension of the THOR [24] shape analysis tool.

Using the terminology of [16], shape analyses of this type generally have four primary operations: 1) rearrangement, 2) abstraction, 3) symbolic execution, 4) entailment. In order to produce numeric abstractions, we need to make slight changes to steps 1, 2, and 4. We also need a method of translating branches involving pointer variables into branches over instrumentation variables. We now describe how to modify each of the shape analysis steps in order to obtain an instrumentation. The modifications are such that they do not interfere with the accuracy or convergence properties of the underlying shape analysis.

Rearrangement In this phase, inductively-defined predicates are expanded with the goal of exposing some specific heap formula (often a formula stating that a certain heap cell exists). We will use the notation $Q(\vec{x})$ to denote a formula with free variables $\vec{x}$ and then $Q(\vec{e})$ to denote the simultaneous substitution of $\vec{e}$ for $\vec{x}$. Given an expansion rule of the form

$$d(\vec{x}) \Rightarrow \big(Q_1(\vec{x}) \vee \ldots \vee Q_n(\vec{x})\big)$$

and a formula $Q * d(\vec{e})$ containing an instance of d, the analysis proceeds by reasoning from each case $Q * Q_i(\vec{e})$.

In order to obtain an instrumentation in such a case, we require that the implication have the form below, where the underlined variables represent instrumentation variables. The expressions e_i' record the relation between the instrumentation variables $\underline{\vec{v}}$, which occur as parameters to the predicate d and the new instrumentation variables $\underline{\vec{z_i}}$, which appear in the Q_i.

$$\begin{aligned} d(\underline{\vec{v}}; \vec{w}) \Rightarrow & (e_1 \wedge \exists \underline{\vec{z_1}}.\ e_1' \wedge Q_1') \vee \ldots \vee \\ & (e_n \wedge \exists \underline{\vec{z_n}}.\ e_n' \wedge Q_n') \end{aligned}$$

As a concrete example, consider our list predicate, which can be written in the following form (where $\underline{\vec{z_1}}$ is empty and $e_1' = true$ in the first case).

$$\begin{aligned} & ls(\underline{n}; first, tail) \Rightarrow \\ & \quad \underline{n} = 0 \wedge true \wedge (\mathbf{emp} \wedge first = tail) \\ & \quad \vee (\underline{n} > 0) \wedge \exists \underline{n}'.\ (\underline{n}' = \underline{n} - 1) \wedge \\ & \qquad (\exists z.\ (first \mapsto [\mathrm{next}: z]) * ls(\underline{n}'; z, tail)) \end{aligned}$$

We can either provide such implications directly or, as is the case with our tool, generate them from user-provided inductive definitions.

We can represent this reasoning by cases in the instrumented program by inserting an n-way non-deterministic branch (which can be encoded as nested binary branches and justified with INST-DISJ). We now have the precondition $Q * Q_i(\vec{e})$ in each case and, based on the restricted syntax above, we know that $Q_i(\vec{e})$ has the form $e_i \wedge \exists \underline{\vec{z_i}}.\ e_i' \wedge Q_i'$. Continuing our insertion of instrumentation commands, we can use INST-ASSUME to insert `assume`(e_i) then INST-EXISTS to insert $\underline{\vec{z_i}} := ?$. We now have a precondition of the form $Q * (e_i \wedge e_i' \wedge Q_i')$ and can use INST-ASSUME again to insert `assume`(e_i').

Putting this together and simplifying the resulting program, we get the following, where $\widehat{k}_i$ is the result of instrumenting the current continuation k starting from the precondition $Q * (e_i \wedge e_i' \wedge Q_i')$.

$$\texttt{branch}\ \ldots, e_i \Rightarrow \underline{\vec{z_i}}' := ?; \texttt{assume}(e_i'); \widehat{k}_i, \ldots\ \texttt{end}$$

Symbolic Execution During symbolic execution, we are given the precondition Q of some command c and must compute the postcondition. This step is generally specified assuming that rearrangement has already revealed any heap cells necessary to process c. That is, if c is $x.\mathrm{next} = \mathrm{nil}$ then Q has the form $Q * (x \mapsto [\mathrm{next}: e])$. This step is unchanged, as original program commands are simply copied into the instrumentation, a process justified by the COMMAND rule.

Abstraction Abstraction corresponds to applying an implication whose right-hand side contains a single instance of an inductive definition (there can be other abstraction rules as well, but these are the kind we will want to instrument). Again, we require that the implication in question have the following specific form, where e relates $\underline{\vec{z}}$ to $\underline{\vec{v}}$.

$$Q(\underline{\vec{z}}) \Rightarrow \exists \underline{\vec{v}}.\ e \wedge d(\underline{\vec{v}}; \vec{w})$$

As a concrete example, consider the following abstraction rule for lists.

$$(x \mapsto [\text{next} : y]) * ls(\underline{n}'; y, z) \Rightarrow$$
$$\exists \underline{n}.\ \underline{n} > 0 \wedge (\underline{n} = \underline{n}' + 1) \wedge ls(\underline{n}; x, z)$$

Abstraction rules for all the data structures considered in our experiments can be written in this form.

Suppose we have such an abstraction rule and a precondition Q_0 to which it is applicable. That is, $Q_0 = Q_0' * Q(\vec{e})$ for some $\vec{e}$. Then by STRENGTHENING followed by INST-EXISTS and INST-ASSUME, we obtain the following instrumentation commands

$$\underline{\vec{v}} := ?;\texttt{assume}(e)$$

and can proceed to reason from the precondition $Q_0' * (d(\underline{\vec{v}}; \vec{w}) \wedge e)$. In our list example, for the precondition $(x \mapsto [\text{next} : y]) * ls(\underline{n}'; y, z)$ we would obtain the commands $\underline{n} := ?;$ $\texttt{assume}(\underline{n} > 0 \wedge \underline{n} = \underline{n}' + 1)$.

Entailment In order to determine whether the shape analysis process has converged, we need to check whether the current precondition entails an invariant that we have seen before. This occurs, for example, when we process a `goto` l command. In this case, we want to check whether the current formula implies another formula that is already known to be in $\Gamma(l)$. In our implementation, when this holds, the spatial portion of the formulas will always be syntactically equal up to renaming of existentially quantified variables and instrumentation variables. Thus, we have some $\exists \vec{x}.\ Q_1$ and some previously discovered formula $\exists \vec{y}.\ Q_2$ such that $Q_1 \equiv \sigma(Q_2)$, where σ is a substitution whose domain consists of $\vec{y}$ and the instrumentation variables in Q_2. Let σ_i be the portion of σ that affects instrumentation variables. Then, for all $\underline{x} \in dom(\sigma_i)$ we add the command $\underline{x} := \sigma_i(\widehat{x})$. The commands are justified using the INST-ASSIGN rule.

To again return to our list example, if we have $ls(\underline{n}_2; x, \text{nil})$ and are processing `goto` l and have previously discovered that $ls(\underline{n}; x, \text{nil})$ is an invariant at l, then we can establish this invariant from the current state by executing the instrumentation command $\underline{n} := \underline{n}_2$. The post-condition of this command is $ls(\underline{n}; x, \text{nil})$, allowing us to show that we have properly instrumented `goto` l using the GOTO rule.

Branch Condition Translation Branches in the original program that only involve integer-valued variables can be carried over unchanged to the numeric program. However, branches that involve pointer variables will be abstracted by the $\text{numabs}_V(k)$ function. In many cases, we can obtain good approximations of these branches in terms of instrumentation variables. For example, given the state $ls(n; x, \text{nil})$ and the condition $x = \text{nil}$, we can prove that this condition is equivalent to $n = 0$.

In the general case, given a state Q and a continuation

$$\texttt{branch}\ \ldots, e \Rightarrow k, \ldots\ \texttt{end}$$

we want to find some e' involving only integer variables such that $Q \wedge e \Rightarrow e'$. We can then use the INST-ASSUME rule to add an $\texttt{assume}(e')$ command to the branch case, obtaining

$$\texttt{branch}\ \ldots, e \Rightarrow \texttt{assume}(e'); k, \ldots\ \texttt{end}$$

This $\texttt{assume}(e')$ command will then become part of the numeric program we produce.

The search for such an e' is fairly undirected, however, as the right-hand side of the implication is entirely unknown. In our tool, we have found it is easier to instead consider the equivalent formula $Q \wedge \neg e' \Rightarrow \neg e$. We can then search for an assumption e_a such that $Q \wedge e_a \Rightarrow \neg e$. This similar to the process of abduction described in [9], except that we are searching for a pure, rather than spatial, assumption. This ensures that we have something known on both the left and right sides of the implication, which helps to structure the proof search. Our translated branch condition is then $\neg e_a$.

Consider the example of $ls(a; x, y) * ls(b; y, x)$ and the condition $x \neq y$. This describes a state in which we have a cyclic list that x and y both point into. We would ask our prover to find an e_a such that $(ls(a; x, y) * ls(b; y, x)) \wedge e_a \Rightarrow x = y$. The discovered e_a is $(a = 0) \vee (b = 0)$. We then negate this to obtain $(a \neq 0) \wedge (b \neq 0)$, which is an over-approximation of $x \neq y$ in the given state.

In practice, we find e_a by expanding inductive predicates on the left and recording in which cases a proof of $Q \Rightarrow \neg e$ succeeds and in which it fails. The condition returned is then the condition that rules out all the failure cases.

8. Experimental Results

We have implemented the techniques described here in the tool THOR [24]. Table 1 summarizes the experimental results of verifying safety and termination of programs that manipulate different inductive data structures. For each program, we use THOR to produce a numeric abstraction of the original program. Then we use BLAST and ARMC to verify safety and ARMC-LIVE to check termination of the numeric abstraction. The results of BLAST, ARMC, and ARMC-LIVE are all consistent with the expected results and thus we only list the timing information.

In these results, safety refers to memory safety of the program. We can use the numeric abstraction produced by THOR to show memory safety in cases where memory safety involves tracking arithmetic information. For example, the copy_zip example involves copying a list and then combining it with the original list to produce a list of pairs. The code for combining the lists assumes that they have equal length and will produce a memory fault if not. Verification for this example then begins by adding assert statements at each memory access checking that the variable to be accessed is non-nil. For example, we might translate $y := x.\text{next};\ k$ to the continuation $\texttt{branch}\ x = \text{nil} \Rightarrow \texttt{abort}, x \neq \text{nil} \Rightarrow y := x.\text{next};\ k\ \texttt{end}$. We then run THOR on this code to produce a numeric abstraction. Some of the `abort` statements will have been shown to be unreachable by the shape analysis, but others will remain and be present in the numeric abstraction. This numeric program is then fed to a separate safety tool, such as BLAST or ARMC, which can prove that the remaining `abort` statements are all unreachable. This implies that they are also unreachable in the original program and thus the original program is memory safe.

As can be seen in the *lift* and *dfs* examples, the analysis time for the numeric program is sometimes quite large. This is due to the number of branches and instrumentation variables that are inserted by our current implementation. Many of these are redundant, and simplifying the resulting numeric programs, either by changes to the algorithm or post-processing steps will be important in order to allow such an approach to scale.

9. Conclusion

We have presented a formal system for producing numeric abstractions of heap-manipulating programs. The numeric abstractions describe how integer properties of data structures change during program execution and can be used to reason effectively about safety, termination, and variable bounds of the original program. In fact, the numeric abstraction is a sound abstraction for all $\text{LTL}_{-\text{X}}$ properties specified in terms of integer variables shared by the original program and numeric abstraction.

We have implemented this abstraction technique as part of a shape analysis tool based on separation logic that includes support for user-defined inductive predicates. We applied the implementation to a collection of test programs and used a variety of existing numeric analysis tools to investigate safety and termination prop-

Program	Expected Result	T_{NA}	T_{BLAST}	T_{ARMC}	$T_{\text{ARMC-LIVE}}$
Doubly Linked Lists					
reverse_bad	unsafe	0.076	0.151	0.037	0.044
copy_zip	safe / terminates	4.862	0.238	7.674	31.683
iter_sum	safe / terminates	1.204	0.342	8.036	9.589
Circular Doubly-Linked Lists					
traverse	safe / terminates	1.526	0.046	0.908	1.383
delete	safe / terminates	2.245	0.068	11.138	20.204
meet	safe / diverges	0.760	0.126	1.734	0.180
Circular Linked Lists					
sum	safe / terminates	0.827	0.065	1.621	2.582
add_after	safe / terminates	1.072	0.061	4.846	12.342
add_after_loop	safe / diverges	0.997	0.065	1.945	3.364
Skip Lists					
create	safe / terminates	9.651	0.122	10.546	34.960
lift	unsafe	10.464	0.356	5.814	971.090
find_loop	safe / diverges	4.431	0.106	36.860	45.709
Binary Search Trees					
insert	safe / terminates	1.550	0.046	0.458	0.895
mem	safe / terminates	0.573	0.042	0.387	2.690
Binary Trees					
dfs	safe / terminates	0.780	0.042	0.444	3503.065

Table 1. The experimental results. Time is in seconds. T_{NA} represents the time of producing numeric abstraction. T_{BLAST}, T_{ARMC}, and $T_{\text{ARMC-LIVE}}$ represent the time of verifying the numeric abstraction by BLAST, ARMC, and ARMC-LIVE respectively.

erties of these programs. The results demonstrate that generating numeric abstractions is a flexible and effective means of extending the precision of shape analysis tools based on separation logic. Such an approach allows reasoning about complex numeric safety and termination properties without complicating the shape analysis tool or the abstract domain on which it is based.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213. ACM Press, 2001.

[2] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25. ACM Press, 2004.

[3] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, LNCS 4590, pages 178–192. Springer, 2007.

[4] J. Berdine, B. Cook, D. Distefano, P. W. O'hearn, and Q. Mary. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, pages 386–400. Springer, 2006.

[5] D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *CAV*, LNCS 4144, pages 532–546. Springer, 2006.

[6] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, LNCS 4144, pages 517–531. Springer, 2006.

[7] J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, pages 101–112. ACM Press, 2008.

[8] M. C. Browne, E. M. Clarke, and O. Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1-2):115–131, 1988.

[9] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, New York, NY, USA, 2009. ACM.

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

[11] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, New York, NY, USA, 2008. ACM.

[12] B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In *SAS*, LNCS 4634, pages 384–401. Springer, 2007.

[13] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, and V. Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD'09*, 2009.

[14] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, New York, NY, USA, 2006. ACM.

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP*, pages 21–30, 2005.

[16] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, LNCS 3920, pages 287–302. Springer, 2006.

[17] D. Distefano and M. J. Parkinson. jStar: towards practical verification for java. In *OOPSLA*, pages 213–226. ACM, 2008.

[18] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, New York, NY, USA, 2009. ACM.

[19] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving termination of tree manipulating programs. In *ATVA*, LNCS 4762, pages 145–161. Springer, 2007.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM Press, 2002.

[21] A. Loginov, T. W. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS*, LNCS 4134, pages 261–279. Springer, 2006.

[22] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, LNCS 4634, pages 419–436. Springer, 2007.

[23] S. Magill, A. Nanevski, and E. M. Clarke. Inferring invariants in separation logic for imperative list-processing programs. In *SPACE*, 2006.

[24] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV*, LNCS 5123, pages 428–432. Springer, 2008.

[25] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, 2001.

[26] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.

[27] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[28] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE Computer Society, 2004.

[29] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, LNCS 4354, pages 245–259. Springer, 2007.

[30] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV 2008*, LNCS 5123, pages 314–327. Springer-Verlag, 2008.

[31] R. Rugina. Quantitative shape analysis. In *SAS*, pages 228–245, 2004.

[32] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[33] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.