Inferring Invariants in Separation Logic for Imperative List-processing Programs

Stephen Magill
Carnegie Mellon University
smagill@cs.cmu.edu

Aleksandar Nanevski
Harvard University
aleks@eecs.harvard.edu

Edmund Clarke,
Peter Lee
Carnegie Mellon University

ABSTRACT

An algorithm is presented for automatically inferring loop invariants in separation logic for imperative list-processing programs. A prototype implementation for a C-like language is shown to be successful in generating loop invariants for a variety of sample programs. The programs, while relatively small, iteratively perform destructive heap operations and hence pose problems more than challenging enough to demonstrate the utility of the approach. The invariants express information not only about the shape of the heap but also conventional properties of the program data. This combination makes it possible, in principle, to solve a wider range of verification problems and makes it easier to incorporate separation logic reasoning into static analysis systems, such as software model checkers. It also can provide a component of a separation-logic-based code certification system *a la* proof-carrying code.

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, particularly 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. Computing weakest preconditions and strongest postconditions becomes exponential in the number of program variables. This can be ameliorated 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. Any program analysis must also take into account the global context, since any two pointer variables, regardless of scope, are potential aliases.

Contrast this with separation logic [17], a program logic with connectives for expressing aliasing patterns and 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 all sorts of static analysis methods and tools.

In this paper we consider the problem of automatically inferring loop invariants in separation logic for imperative list-processing programs. Our primary motivation for doing this is to provide a key component of a general verification system for imperative programs that make use of pointers. Until recently, automated reasoning in separation logic has been largely unexplored. However, Berdine, et. al. [2] have presented a decision procedure for a fragment of separation logic that includes a list predicate. Weber [20] has developed an implementation of separation logic in Isabelle/HOL with support for tactics-based reasoning. These approaches allow for the automatic or semi-automatic verification of programs annotated with loop invariants and pre-/post-conditions. But what is missing is the generation of the loop invariants. Sims [19] has proposed an extension to separation logic that allows for the representation of fixed points in the logic, and thus the computation of strongest postconditions for while loops. But the issue still remains of how to compute these fixed points.

We present a heuristic method for inferring loop invariants in separation logic for iterative programs operating on integers and linked lists. The invariants are obtained by applying symbolic evaluation [5] to the loop body and then applying a *fold* operation, which weakens the result such that when this process is repeated, it eventually converges. Our invariants are capable of expressing both information about the structure of the graph of pointers (*shape* information), but also facts about program data, both on the stack and in the heap. As we will show, the ability to describe properties of data is desirable in certain programs. Once we have loop invariants, the way is paved for the adaptation of existing program analysis techniques (such as software model checking [1, 4, 12]) to the separation logic framework. Applications to code certification, for example via proof-carrying code [13, 14], would also be enabled by this.

We have implemented a prototype of our algorithm. Using it, we are able to extract loop invariants automatically for a number of pointer programs. These examples, while rather small, are iterative and perform destructive heap operations. Thus, they are more than challenging enough to demonstrate the utility of our algorithm.

There are many existing approaches to the treatment of shape information in iterative programs [9, 18]. It is not our intent here to try to beat these techniques. We do not attempt to quantify the relative strength of our inference

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

procedure versus others. We wish merely to present an alternate approach, as we believe that the development of a procedure based on separation logic has merit on its own. In particular, the compositionality of separation logic proofs is vital if such proofs are to be used in a proof-carrying code system. In such systems, the proof is written by the code producer, but the code runs on the client machine in a different and unpredictable context. As such, the proofs cannot depend on the global variables and aliasing patterns present on the client. Separation logic allows us to specify and check this independence.

In section 2 we present a brief introduction to separation logic. In section 3, we describe the algorithm, which consists of a symbolic evaluation routine and heuristics for finding fixed points. In section 4 we show the soundness of our approach, and in section 5, we discuss incompleteness and give an example on which our algorithm fails. Finally, in section 6 we give the results of our tests involving an SML implementation of the algorithm.

2. SEPARATION LOGIC

Here we present a brief overview of separation logic. For a full treatment, see [17]. The logic consists of all the connectives and quantifiers from classical logic, plus two new connectives: a *separating conjunction* (written $p * q$) and a *separating implication* ($p -\!\!* q$). Only separating conjunction will be used for the material in this paper. Figure 1 gives the domains involved in the semantics of the logic. There is a set of locations, $\mathcal{L}$, which is ordered and disjoint from the set of integers, $\mathbb{Z}$. The set of values consists of the union of $\mathcal{L}$ and $\mathbb{Z}$. There is a set of variables Ω, which contains at least the program variables. These are the variables that would reside on the stack in a standard implementation of an imperative language. The domain of stacks then is the function space from Ω to $\mathcal{V}$, which maps variables to values. Heaps map locations to values. Since we wish to track allocation, the heap is a partial function whose domain consists of only those locations that have been allocated. Thus, the domain of heaps is the set of partial functions from locations to values. There is a distinguished value **null** in $\mathcal{L}$ that is not in the domain of any heap. The models consist of pairs from $S \times H$.

The syntax is given in figure 2. New constructs not present in classical logic include the pointer expressions and heap predicates. Recall that the locations are ordered. $x.n$ stands for the n^{th} location after x in this ordering. **null** is the non-addressable element of $\mathcal{L}$. The heap predicate **emp** states that the heap is empty (its domain is the empty set). The predicate $e_1 \mapsto e_2$ indicates that the heap contains a single cell at location e_1, which contains the value given by e_2. Heaps with more than one cell are represented by combining these predicates with $*$. The formula $p * q$ holds if the heap can be split into two disjoint parts, one of which satisfies p while the other satisfies q. So for example, a heap containing two cells, one at location x and one at location y could be represented by the formula $(x \mapsto 3) * (y \mapsto 4)$. There is a special syntactic class of *pure formulas*, which are those formulas that do not contain heap predicates. Pure formulas are independent of the heap. We use the abbreviation $e_1 \mapsto (e_2, e_3)$ to stand for $(e_1 \mapsto e_2) * (e_1.1 \mapsto e_3)$. Thus, $e_1 \mapsto (e_2, e_3)$ can be read as saying that there is a pair of values (e_2, e_3) in the heap starting at location e_1.

We use a fragment of separation logic for our invariants

Locations	$\mathcal{L}$		
Addressable Locns	A	$=$	$\mathcal{L} - \{\mathbf{null}\}$
Integers	$\mathbb{Z}$		
Values	$\mathcal{V}$	$=$	$\mathcal{L} \cup \mathbb{Z}$
Variables	Ω		
Stacks	S	$=$	$\Omega \to \mathcal{V}$
Heaps	H	$=$	$\bigcup_{D \overset{fin}{\subseteq} A} (D \to \mathcal{V})$

Figure 1: Semantic domains

Variables	x, y	$\in$	Ω
Integers	i	$\in$	$\mathbb{Z}$
Integer Expns	i	$::=$	$x \mid n \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 \times i_2$
Pointer Expns	k	$::=$	$x \mid x.n \mid \mathbf{null}$
Boolean Expns	b	$::=$	$i_1 = i_2 \mid l_1 = l_2 \mid$ $i_1 < i_2 \mid \neg P \mid P_1 \wedge P_2 \mid$ $P_1 \vee P_2 \mid \top \mid \bot$
Expressions	e	$::=$	$i \mid b \mid k$
Heap Predicates	h	$::=$	$k \mapsto e \mid \mathbf{emp}$
Pure Formulas	p, q	$::=$	$b \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid \neg p \mid$ $p * q \mid p -\!\!* q \mid \forall x.\ p \mid \exists x.\ p$
Formulas	r, s	$::=$	$b \mid h \mid r \wedge s \mid r \vee s \mid$ $r \Rightarrow s \mid \neg r \mid r * s \mid r -\!\!* s \mid$ $\forall x.\ r \mid \exists x.\ r$

Figure 2: Separation logic syntax

and for reasoning during the computation of the invariants. The fragment we use for invariants consists of those formulas that have the form $p \wedge (h_1 * \ldots * h_n)$, where p is a pure formula and h_i is a heap predicate. This form allows us to separate to some extent the reasoning involving the heap from the classical logic reasoning. If we want to prove $p \wedge \phi \Rightarrow p' \wedge \phi'$, it is sufficient to show that $p \Rightarrow p'$ and $\phi \Rightarrow \phi'$. The first question can be settled by a standard classical logic theorem prover (such as Simplify [15] or Vampyre [3]) and we employ various heuristics, described later, to attempt to prove the second proposition. It should be noted that because pointer expressions only involve addition by a constant and because of the simplicity of ϕ and ϕ' (they are simply heap predicates joined by $*$), we can get a lot of leverage out of relatively simple checks. This method is by no means complete, but it has been sufficient to handle all of our examples. A full separation logic theorem prover would allow for a more robust approach, but we are not aware of the existence of such a system, and building a theorem prover was not the goal of this work. Were such a system to be developed, it could easily be used in place of our heuristics.

3. DESCRIPTION OF THE ALGORITHM

This section describes the operation of the invariant inference algorithm. We present here a summary of its operation and then go into detail in the following subsections. The algorithm infers loop invariants that describe not only the values of program variables, but also the shape and contents of list structures in the heap. Lists are described by an inductive list predicate (defined in section 3.1). The algorithm consists of two mutually recursive parts. In the first part, which applies to straight-line code, the program is evalu-

ated symbolically and the inductively-defined list predicates are expanded on demand via an "**unfold**" operation. Intuitively, **unfold** takes a non-empty list and separates it into its head element and the tail of the list. In the second part of the algorithm, which applies to while loops, a fixed point of the symbolic evaluation function is computed by repeatedly evaluating the loop body and then applying a "**fold**" operation, which applies the inductive definition of lists in the other direction, combining a list's head and tail into a single instance of the list predicate. We use a test we call the "name check" to determine when to apply **fold**. The name check is designed such that **fold** is only applied when it is necessary to make the sequence of formulas computed for the loop converge. Excessive application of **fold** can result in formulas that are too weak. The algorithm relies on its own procedure to handle reasoning in the arithmetically simple domain of pointer expressions and makes use of Vampyre [3] to handle reasoning involving integer expressions. We now present the algorithm in full detail.

3.1 Memory Descriptions

Both portions of the algorithm operate on sets of memories, defined in Figure 3. A memory is a triple $(H; S; P)$, where H is a list of heap entities (points-to and list predicates), S is a list of stack variable assignments, and P is a predicate in classical logic augmented with arithmetic. H keeps track of the values in the heap, S stores the symbolic values of the program's stack variables and P is used to record the conditional expressions that are true along the current path. Note that we separate pointer formulas from integer formulas. Such a separation is possible because our programming language separates integer formulas and pointer formulas into two different syntactic domains (see figure 4). As we will see, the formulas we accumulate as assumptions throughout our analysis come from the branch conditions in the program, so this syntactic differentiation in the programming language allows us to maintain that distinction in the logic. This enables us to use different decision procedures depending on whether we are trying to prove a pointer formula or integer formula. If Γ is a set of integer formulas and Δ is a set of pointer formulas with $FV(\Gamma) \cap FV(\Delta) = \emptyset$ and $\Gamma \not\vdash \mathbf{false}$ then $\Gamma, \Delta \vdash p$ iff $\Delta \vdash p$. Thus, in consistent memories, we can reason about pointers separately.

Depending on where we are in the algorithm, a memory may have some or all of its symbolic variables existentially quantified. Such quantifiers are collected at the head of the memory and have as their scope all of $(H; S; P)$. Memories, as we have presented them so far, are really just a convenient form in which to represent the data on which our symbolic execution algorithm operates. As such, they have no particular semantics. However, as we shall see, the memories correspond directly to separation logic formulas and thus gain an interpretation due to this correspondence. Therefore, free variables in memories have the same meaning as free variables in separation logic annotations.

We use * to separate heap entities in H to stress that each entity refers to a disjoint portion of the heap. However, we treat H as an unordered list and freely apply commutativity of * to reorder elements as necessary. The memory $\exists \bar{v}.\ (H; S; P)$ is equivalent to the separation logic formula $\exists \bar{v}.\ H \wedge \bigwedge S \wedge \bigwedge P$, where $\bigwedge S$ is the conjunction of all the equalities in S and $\bigwedge P$ is the conjunction of all the formulas

Program Vars	x, y	$\in$	Ω
Symbolic Vars	v	$\in$	Σ
Integers	n	$\in$	$\mathbb{Z}$
Integer Expns	i	$::=$	$v \mid n \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 \times i_2$
Pointer Expns	p, q, k	$::=$	$v \mid v.n \mid \mathbf{null}$
Integer Formulas	f_i	$::=$	$i_1 = i_2 \mid i_1 < i_2 \mid \neg P \mid$ $P_1 \wedge P_2 \mid P_1 \vee P_2 \mid$ $\mathtt{true} \mid \mathtt{false}$
Pointer Formulas	f_p	$::=$	$p_1 = p_2 \mid \neg P \mid P_1 \wedge P_2 \mid$ $P_1 \vee P_2 \mid \mathtt{true} \mid \mathtt{false}$
Formulas	b	$::=$	$f_i \mid f_p$
Symbolic Expns	σ	$::=$	$p \mid i$
Heap Entities	h	$::=$	$p \mapsto \sigma \mid \mathbf{ls}(p_1, p_2) \mid \mathbf{ls}^+(p_1, p_2)$
Heaps	H	$::=$	$\cdot \mid h * H$
Stacks	S	$::=$	$\cdot \mid x = \sigma, S$
Predicates	P	$::=$	$\cdot \mid b, P$
Memories	m	$::=$	$(H; S; P) \mid \exists v.\ m$
Memory Sets	M	$::=$	$\{m_1, \ldots, m_n\}$

Figure 3: Definition of memories and memory sets.

in P (we prove that the symbolic execution rules are sound when memories are given such an interpretation in section 4). When discussing this correspondence, we will sometimes use a memory in a place where a formula would normally be expected. In such cases, the memory should be interpreted as just given. For example, we would write $(H; S; P) \Rightarrow (H'; S'; P')$ to mean $H \wedge \bigwedge S \wedge \bigwedge P \Rightarrow H' \wedge \bigwedge S' \wedge \bigwedge P'$. Similarly, sets of memories are treated disjunctively, so if a set of memories $\{m_1, \ldots, m_n\}$ is used in a context where a formula would be expected, this should be interpreted as the formula $m_1 \vee \ldots \vee m_n$.

In fact, our language of memories constitutes a simplified logic of assertions for imperative pointer programs that manipulate lists. Memories correspond to a fragment of separation logic formulas and the symbolic evaluation rules that we present in section 3.3 are a specialization of the separation logic rules to formulas of this restricted form. We choose this form as it is sufficient to represent a large class of invariants and it simplifies the problem of deciding entailment between formulas. In particular, it allows the heap reasoning to be handled relatively independently of the classical reasoning. This enables us to use a classical logic theorem prover such as Simplify [15] or Vampyre [3] to decide implications in the classical domain and allows us to concentrate our efforts on reasoning about the heap.

There are two classes of variables: *symbolic variables* and *program variables*. Symbolic variables arise due to applications of the existential rule below.

$$\frac{\{P\}\ c\ \{Q\}}{\{\exists x.\ P\}\ c\ \{\exists x.\ Q\}}$$

Because they come from existentials, symbolic variables appear in assertions and invariants, but not in the program itself. We use a separate class of variables to track the origin of these variables and keep them separate from program variables, which are those that appear in the program.

We maintain our memories in a form such that program variables appear only in S. All variables in H and P are symbolic variables. Symbolic expressions are built from the standard connectives and symbolic variables and are de-

noted by σ. Pointer expressions consist of a variable or a variable plus a positive integer offset (denoted $v.n$).

Valid heap entities include the standard "points-to" relation from separation logic ($p \mapsto e$) along with the inductive list predicate "**ls**." We write $\mathbf{ls}(p, q)$ when there is a list segment starting at cell p and ending (via a chain of dereferencing of "next" pointers) at q. Each cell in the list is a pair (x, k), where x holds the data and k is a pointer to the next cell (or **null** if there is no next cell). The predicate **ls** is defined inductively as:

DEFINITION 1.

$$\mathbf{ls}(p_1, p_2) \equiv (\exists x, k.\ p_1 \mapsto (x, k) * \mathbf{ls}(k, p_2)) \vee (p_1 = p_2 \wedge \mathbf{emp})$$

This definition states that $\mathbf{ls}(p_1, p_2)$ either describes the empty heap, in which case $p_1 = p_2$ or it describes a heap which can be split into two portions: the head of the list (x, k) and the tail of the list ($\mathbf{ls}(k, p_2)$).

Note our **ls** predicate describes lists that may be cyclic. If we have a list $\mathbf{ls}(p, p)$, it may match either case in the definition above. That is, it may be either empty or cyclic. This is in contrast to Berdine et. al. [2] who adopt a non-cyclic version of **ls** for their automated reasoning. The two versions of $\mathbf{ls}(p_1, p_2)$ are the same when $p_2 = \mathbf{null}$, and are equally easy to work with when doing symbolic evaluation on straight-line code. But when processing a loop, the acyclic version of **ls** becomes problematic in certain cases. We give more details and an example in section 3.6.

It is also necessary to keep track of whether a list is non-empty. While **ls** describes a list that may or may not be empty, the predicate "$\mathbf{ls}^+$" is used to describe lists that are known to be non-empty and is defined as

$$\mathbf{ls}^+(p_1, p_2) \equiv \exists x, k.\ p_1 \mapsto (x, k) * \mathbf{ls}(k, p_2)$$

The reason this non-emptiness information is necessary is because it allows us to infer more inequalities from the heap description. Suppose our heap looks like this:

$$\mathbf{ls}(p_1, p_2) * (q \mapsto x)$$

If the list segment is empty, then $p_1 = p_2$ and p_1 and p_2 are otherwise unconstrained. Thus, it may be the case that $p_1 = q$. If, however, we know that the list segment is non-empty

$$\mathbf{ls}^+(p_1, p_2) * (q \mapsto x)$$

we can conclude that $p_1 \neq q$. This sort of reasoning was necessary in some of our test programs.

Our pointer expressions fall within Presburger arithmetic [8] (in fact, they are much simpler, because only addition of constants is allowed). Thus, the validity of entailments involving pointer expressions is decidable. We write $m \vdash f_p$, where f_p is a pointer formula, to mean that f_p follows from the pointer equalities and inequalities in m. Recall that m is a triple of the form (H;S;P). S is a list of equalities, some of which are equalities between pointer expressions and some of which involve integer expressions. Similarly, P contains some pointer formulas and some integer formulas. It is the portions of S and P that involve pointer formulas that we consider when deciding $m \vdash f_p$. We also include those inequalities that are implicit in the heap (H). For example, if the heap contains $(p \mapsto v_1) * (q \mapsto v_2)$, we can conclude that $p \neq q$. We write $m \vdash b$ to indicate that m entails a general formula b. In this case, all formulas from S and P are considered. If this entailment involves integer arithmetic, it is not generally decidable, but we can use incomplete heuristics, such as those in Simplify [15] and Vampyre [3], to try to find an answer.

Program Vars	x	$\in$	Ω
Integers	n	$\in$	$\mathbb{Z}$
Integer Expns	i	::=	$x \mid n \mid i_1 + i_2 \mid i_1 - i_2 \mid i_1 \times i_2$
Pointer Expns	p	::=	$x \mid x.n \mid$ `null`
Integer Formulas	f_i	::=	$i_1 = i_2 \mid i_1 < i_2 \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid$ `true` $\mid$ `false`
Pointer Formulas	f_p	::=	$p_1 = p_2 \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid$ `true` $\mid$ `false`
Formulas	b	::=	$f_i \mid f_p$
Expressions	e	::=	$p \mid i$
Commands	c	::=	x`:=`$e \mid x$`:=[`e`]` $\mid$ `[`p`]:=`$e \mid$ `skip` $\mid$ `cons` $(e_1, \ldots, e_n) \mid$ `dispose` $p \mid c_1$; $c_2 \mid$ `if` b `then` c_1 `else` $c_2 \mid$ `while` b `do` c `end`

Figure 4: Syntax for our simple imperative language

3.2 Programming Language

We present here a brief summary of the programming language under consideration. For a full description of the language and its semantics, see [17][1]. Figure 4 gives the syntax of the language. It includes pure expressions (e) as well as commands for assignment ($x := e$), mutation of heap cells ($[p] := e$), lookup ($x := [p]$), allocation ($x := cons(e_1, \ldots, e_n)$), which allocates and initializes n consecutive heap cells, and disposal (`dispose` p), which frees the heap cell at p. It also contains the standard conditional statement and while loops. Note that the condition of an "if" or "while" statement can involve pointers or integers, but not both. However, conditionals involving pointer and integer formulas joined with $\wedge$ or $\vee$ can be transformed into equivalent code which places each formula in a separate conditional, so this does not affect the expressiveness of our language. Square brackets are used to signify dereference, in the same manner that C uses the asterisk. Pointer expressions can contain an offset, and we use the same notation for this that we did in Figure 3. So $x := [y.1]$ means "assign to x the value in the heap cell at location $y + 1$."

The only values in our language are integers and pointers to lists of integers, although there is nothing preventing the addition of other value types. Boolean values and lists of Booleans could be easily added. Lists of lists of integers and the like require more thought, but it is our belief that they can also be added without substantially changing the symbolic evaluation and invariant inference frameworks presented here.

3.3 Symbolic Evaluation

The symbolic evaluator takes a memory and a command and returns the set of memories that can result from executing that command. The set of memories returned is treated disjunctively. That is, the command may terminate in a state satisfying any one of the returned memories.

[1] However, note that we use the version of the language without full address arithmetic (we allow only addition of a constant).

$$\frac{}{(H;S,x=\sigma;P)\ [x{=}e]\ \{(H;S,x=[\![e]\!]_{S,x=\sigma};P)\}}\ \textit{assign} \qquad \frac{(H,p'\mapsto\sigma;S;P)\vdash p=p'}{(H,p'\mapsto\sigma;S;P)\ [[p]:=e]\ \{(H,p'\mapsto[\![e]\!]_S;S;P)\}}\ \textit{mutate}$$

$$\frac{(H,p'\mapsto\sigma_2;S,x=\sigma_1;P)\vdash p=p'}{(H,p'\mapsto\sigma_2;S,x=\sigma_1;P)\ [x:=[p]]\ \{(H,p'\mapsto\sigma_2;S,x=\sigma_2;P)\}}\ \textit{lookup}$$

$$\frac{}{(H;S,x=\sigma_1;P)\ [x:=\texttt{cons}\ (e_0,\ldots,e_n)]\ \{\exists v.\ (H,v.0\mapsto[\![e_0]\!]_{S,x=\sigma_1},\ldots,v.n\mapsto[\![e_n]\!]_{S,x=\sigma_1};S,x=v;P)\}}\ \textit{alloc}\ (v\ \textit{fresh})$$

$$\frac{(H,p'\mapsto\sigma;S;P)\vdash p=p'}{(H,p'\mapsto\sigma;S;P)\ [\texttt{dispose}\ p]\ \{(H;S;P)\}}\ \textit{dispose} \qquad \frac{}{(S;H;P)\ [\texttt{skip}]\ \{(S;H;P)\}}\ \textit{skip}$$

$$\frac{(H;S;P)\vdash[\![b]\!]_S \quad (H;S;P)\ [c_1]\ M}{(H;S;P)\ [\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\ M}\ \textit{if}_t \qquad \frac{(H;S;P)\vdash\neg[\![b]\!]_S \quad (H;S;P)\ [c_2]\ M}{(H;S;P)\ [\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\ M}\ \textit{if}_f$$

$$\frac{(H;S;P,[\![b]\!]_S)\ [c_1]\ M_1 \quad (H;S;P,\neg[\![b]\!]_S)\ [c_2]\ M_2}{(H;S;P)\ [\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\ M_1\cup M_2}\ \textit{if}_n \qquad \frac{m\ [c_1]\ M' \quad M'\ [c_2]\ M}{m\ [c_1;c_2]\ M}\ \textit{seq}$$

$$\frac{m_1\ [c]\ M_1 \quad \ldots \quad m_n\ [c]\ M_n}{\{m_1,\ldots,m_n\}\ [c]\ \left(\bigcup_i M_i\right)}\ \textit{sets} \qquad \frac{m\ [c]\ \{m'_1,\ldots,m'_n\}}{\exists v.\ m\ [c]\ \{\exists v.\ m'_1,\ldots,\exists v.\ m'_n\}}\ \textit{exists}$$

$$\frac{(H,\mathbf{ls}(v_1,v_2);S;P)\vdash v_1\neq v_2 \quad \exists v_3,v_4.\ (H,v_1\mapsto(v_3,v_4),\mathbf{ls}(v_4,v_2);S;P)\ [c]\ M'}{(H,\mathbf{ls}(v_1,v_2);S;P)\ [c]\ M'}\ \textit{unfold}$$

Figure 5: Symbolic evaluation rules for straight-line code.

```
1:  {ls(old,null)}
2:  new := null;
3:  curr := old;

4:  while (curr <> null) do {
5:    old := [old.1];
6:    [curr.1] := new;
7:    new := curr;
8:    curr := old;
9:  }
```

Figure 6: In-place list reversal

Figure 6 gives a routine for in-place reversal of a linked list. We will use this as a running example to demonstrate the operational behavior of the algorithm.

Our symbolic evaluation routine starts with an annotation $H \wedge P$ provided by the programmer, describing the shape of the heap and any initial facts about the stack variables. This is given on line 1 in our example program. We convert this to an initial memory, $\exists \bar{v}.\ (H'; S; P')$, where S is a list of equalities of the form $x = v$, with x a program variable and v a new symbolic variable. H' is then H with each such x replaced by the corresponding v. The formula P' is the result of the same replacements applied to P. Thus, S becomes the only place where program variables occur, an important invariant that is key to keeping the evaluation rules simple. The annotation for our example is $\mathbf{ls}(\mathit{old}, \mathbf{null})$, so the initial memory would be $\exists v_1, v_2, v_3.\ (\mathbf{ls}(v_1, \mathit{null}); \mathit{old} = v_1, \mathit{new} = v_2, \mathit{curr} = v_3; \cdot)$. Note that the initial memory is equivalent to the original annotation since $F \Leftrightarrow \exists v.\ F[x/v] \wedge x = v$, where F is an arbitrary separation logic formula and v is not free in F. (we use $F[x/e]$ to mean F with x replaced by e)

After we have obtained the initial memory $\exists \bar{v}.\ m_0$, we symbolically evaluate the program c starting from m_0 and computing a postcondition M', such that the separation logic triple $\{m_0\}\ c\ \{M'\}$ holds[2]. Of course, if the program contains loops, part of computing this postcondition will involve inferring invariants for the loops, an issue we address in section 3.4.

We chose the given form for memories both to avoid theorem proving in full separation logic and also to simplify the evaluation rules. If the precondition matches the form of our memories, it eliminates the quantifiers in almost all of the separation logic rules (the exception is allocation, which is described later). A description of how this is done is included in the full paper.

Figure 5 gives the rules for symbolic evaluation. These are all derived by considering how the restricted form of our memories simplifies the separation logic rules. The only rule that retains a quantifier is allocation. The function $[\![e]\!]_S$ rewrites a program expression in terms of symbolic variables. A definition is given below ($e[x_i/\sigma_i]$ stands for the simultaneous substitution of each x_i by σ_i in e).

Definition 2.

$$[\![e]\!]_{x_1=\sigma_1,\ldots,x_n=\sigma_n} = e[x_i/\sigma_i]$$

Our judgments have two forms: $m\ [c]\ M'$ holds if execut-

[2] for a full description of the rules for separation logic triples, see [17]

ing the command c starting in memory m always yields a memory in M'. The form $M\ [c]\ M'$ is similar except that we start in the set of memories M. So for *every* $m \in M$, executing c starting from m must yield a memory in M'. Note that the *exists* rule would normally have the side-condition that $v \notin FV(c)$. In this case, however, since v is a symbolic variable and the set of symbolic variables is disjoint from the program variables, the condition is always satisfied.

We can now process the first commands in our example program (Figure 6). Recall that the initial memory was

$$(\mathbf{ls}(v_1, null);\ old = v_1, new = v_2, curr = v_3; \cdot)$$

After evaluating `new := null`, we get

$$(\mathbf{ls}(v_1, null);\ old = v_1, new = \mathbf{null}, curr = v_3; \cdot)$$

And after `curr := old` we have

$$(\mathbf{ls}(v_1, null);\ old = v_1, new = \mathbf{null}, curr = v_1; \cdot)$$

To continue with the example, we need to describe how loops are handled. This is the topic of the next section, so we will delay a full discussion of loops until then. At the moment we shall just state that the first step in processing a loop is to add the loop condition to P and process the body. This is enough to let us continue. So after the loop header at line 4, we have

$$(\mathbf{ls}(v_1, null);\ old = v_1, new = \mathbf{null}, curr = v_1; v_1 \neq \mathbf{null})$$

We then reach `old := [old.1]`, which looks up the value of `old.1` in the heap and assigns it to `old`. Our heap does not explicitly contain a cell corresponding to `old.1` (such a cell would have the form $old.1 \mapsto \sigma$). However, we know that $\mathbf{ls}(v_1, null)$ and $v_1 \neq null$, which allows us to unfold the recursively-defined **ls** predicate according to definition 1. This gives us $(v_1 \mapsto v_2) * (v_1.1 \mapsto v_3) * \mathbf{ls}(v_3, \mathbf{null})$ in the heap. Since $old.1 = v_1.1$, we now have $old.1$ explicitly in the heap and can look up its value (v_3).

The *unfold* rule handles the unrolling of inductive definitions (so far, this is just **ls**). This rule simply expands the **ls** predicate according to Definition 1. To optimize the algorithm we can choose to apply *unfold* only when no other rule applies and even then, only to the list we need to expand to process the current command. For example, if we have the memory $(H, \mathbf{ls}(v_1, v_2); S; P)$ and are trying to evaluate $x := [p_1]$, and $p_1 = v_1$, then we would expand $\mathbf{ls}(v_1, v_2)$ according to the *unfold* rule.

Proceeding with our example, we have unrolled

$$(\mathbf{ls}(v_1, null);\ old = v_1, new = \mathbf{null}, curr = v_1; v_1 \neq \mathbf{null})$$

to

$$(v_1 \mapsto (v_2, v_3) * \mathbf{ls}(v_3, null);\ old = v_1, new = \mathbf{null}, curr = v_1; v_1 \neq \mathbf{null})$$

and we can now finish processing `old := [old.1]` to get

$$(v_1 \mapsto (v_2, v_3) * \mathbf{ls}(v_3, null);\ old = v_3, new = \mathbf{null}, curr = v_1; v_1 \neq \mathbf{null})$$

We then evaluate `[curr.1] := new` yielding

$$(v_1 \mapsto (v_2, \mathbf{null}) * \mathbf{ls}(v_3, null);\ old = v_3, new = \mathbf{null}, curr = v_1; v_1 \neq \mathbf{null})$$

and finally the two assignments `new := curr`

$$(v_1 \mapsto (v_2, \mathbf{null}) * \mathbf{ls}(v_3, null);\ old = v_3, new = v_1, curr = v_1; v_1 \neq \mathbf{null})$$

and `curr := old`

$$(v_1 \mapsto (v_2, \mathbf{null}) * \mathbf{ls}(v_3, null);\ old = v_3, new = v_1, curr = v_3; v_1 \neq \mathbf{null})$$

3.4 Invariant Inference

The symbolic evaluation procedure described in the previous section allows us to get a postcondition from a supplied precondition for straight-line code. If c is a straight-line piece of code, we can start with memory m and find some set of memories M' such that $m\ [c]\ M'$. The postcondition for c is then the disjunction of the memories in M'. One approach to dealing with loops is to iterate this process.

Suppose we are trying to find an invariant for the loop `while` b `do` c `end`. To do this, we start with a set of memories M_{pre}. It is important for convergence that the only free variables in these memories be program variables. This is an easy requirement to satisfy since we can simply take each $m \in M_{pre}$ and quantify the symbolic variables to obtain $\exists \bar{v}.\ m$, where $\bar{v}$ is the list of free symbolic variables in m. This is sound since the symbolic variables are not modified by the program.

We add to each of these memories the loop condition b. The notation $M \wedge b$ means that we add $[\![b]\!]_S$ (defined in section 3.3) to each of the memories in M, resulting in the set $\{\exists \bar{v}.\ (H; S; P, [\![b]\!]_S) \mid \exists \bar{v}.\ (H; S; P) \in M\}$. This may seem questionable since $[\![b]\!]_S$ contains free variables that become bound when it is moved inside the quantifier. However, the process is sound as can be seen by the following progression. We start with

$$(\exists \bar{v}.\ (H; S; P)) \wedge b$$

Since b contains only program variables, we can move it inside the existential.

$$\exists \bar{v}.\ (H; S; P) \wedge b$$

And since $S \wedge b \Leftrightarrow S \wedge [\![b]\!]_S$, the above formula is equivalent to

$$\exists \bar{v}.\ (H; S; P) \wedge [\![b]\!]_S$$

which is equivalent to

$$\exists \bar{v}.\ (H; S; P, [\![b]\!]_S)$$

This gives us the set of reachable memories at the start of the loop. We then symbolically execute the loop body c. We then weaken the result and repeat this process. The full algorithm is given below.

1. Let $M'_0 = M_{pre}$
2. Given $M'_0, \ldots, M'_i$, compute M_{i+1} such that $M'_i \wedge b\ [c]\ M_{i+1}$
3. Compute M'_{i+1} such that $M_{i+1} \Rightarrow M'_{i+1}$
4. If $\bigcup_i M'_i$ is an invariant, then stop. Otherwise return to step 2.

The process terminates when the set of memories obtained reaches a fixed point. Whether or not we reach this fixed

$$
\begin{array}{rcl}
p \mapsto (v,k) * \mathbf{ls}(k,q) & \Rightarrow & \mathbf{ls}^+(p,q) \\
\mathbf{ls}(p,k) * k \mapsto (v,q) & \Rightarrow & \mathbf{ls}^+(p,q) \\
p \mapsto (v_1,k) * k \mapsto (v_2,q) & \Rightarrow & \mathbf{ls}^+(p,q)
\end{array}
$$

Figure 7: Rewrite rules for fold. In order to apply them to a memory $(H;S;P)$, it must be the case that $\neg hasname(S,k)$.

point depends crucially on the weakening that is performed in step 3. We describe how M'_{i+1} is computed in the next section. The test in step 4 is performed by checking whether there is an instance of the following rule where $M = \bigcup_i M_i$.

$$
\frac{M \wedge b \ [c] \ M' \quad M' \Rightarrow M}{M \ [\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}] \ M \wedge \neg b} \ \textit{while}
$$

This is just the standard Hoare rule for while combined with the rule of consequence. Once we find an instance of this rule, we can continue symbolically evaluating the rest of the code starting from the post-condition $M \wedge \neg b$. We discuss how to decide the weakening in the second premise in section 3.4.2. In the next section we describe a procedure for performing the weakening in step 3.

3.4.1 Fold

In this section, we describe how we perform the weakening in step 3 of the invariant inference algorithm. The core of this transformation is a function *fold*, which is the inverse of the *unfold* rule used by the symbolic evaluation routine. *fold* performs a weakening of the heap that helps the search for a fixed point converge. It does this by examining the heap and trying to extend existing lists and create new lists using the rewrite rules given in Figure 7. Additionally, we allow the procedure to weaken $\mathbf{ls}^+$ predicates to **ls** predicates if this is necessary to apply one of the above rules. Note however, that we cannot simply apply these rules in an unrestricted manner or we will end up with a heap that is too weak to continue processing the loop. Consider a loop that iterates through a list:

```
while(curr <> null) do {
  curr := [curr.1];
}
```

We would start with a memory like

$$(\mathbf{ls}(v_1, \mathbf{null}); l = v_1, curr = v_1; \cdot)$$

and after one iteration would produce the following memory

$$(v_1 \mapsto (v_2, v_3) * \mathbf{ls}(v_3, \mathbf{null}); l = v_1, curr = v_3; \cdot)$$

If we apply the rewrites in Figure 7 indiscriminately, we obtain $\mathbf{ls}(v_1, \mathbf{null})$ for the heap and have lost track of where in the list *curr* points. The next attempt to evaluate `curr := [curr.1]` will cause the symbolic evaluation routine to get stuck (no symbolic evaluation rule applies).

So we need a restriction on when to apply *fold*. Applying it too often results in weak descriptions of the heap that cause evaluation to get stuck. Applying it too little keeps the fixed point computation from terminating. The restriction that we have adopted is to fold up a list only when the intermediate pointer does not correspond to a variable in the program. Using the values of the program variables to guide the selection of which heap cells to fold seems natural in the sense that if a memory cell is important, the program probably maintains a pointer to it. It is certainly the case that refusing to fold any cell to which a program variable still points will prevent us from getting immediately stuck. That these are the only cells we need to keep separate is not as clear and, in fact, is not always the case. However, for loop invariants that are expressible solely in terms of the program variables this heuristic has proven successful for typical programs (though it is by no means complete).

We introduce a new function *hasname* to perform this check. It takes as arguments the list of equalities S and the symbolic variable to check. It returns true if there is a program variable equal to the symbolic variable provided.

$$
\begin{array}{l}
hasname((H;S;P),v) \text{ iff there is some program} \\
\qquad\qquad \text{variable } x \text{ such that } (H;S;P) \vdash x = v
\end{array}
$$

This can be decided by repeatedly querying our decision procedure for pointer expressions, although there are also more efficient approaches. We then only fold a memory cell v when $\neg hasname(S,v)$. So, for example, the memory

$$(ls(v_1,v_2) * v_2 \mapsto (v_3,v_4) * ls(v_4, \mathbf{null}); l = v_1, curr = v_4; \cdot)$$

would be folded to

$$(ls(v_1,v_4) * ls(v_4, \mathbf{null}); l = v_1, curr = v_4;)$$

because there is no program variable corresponding to v_2. However, the memory

$$
\begin{array}{l}
(ls(v_1,v_2) * v_2 \mapsto (v_3,v_4) * ls(v_4, \mathbf{null}); \\
\qquad\qquad l = v_1, curr = v_4, prev = v_2; \cdot)
\end{array}
$$

which is the same as above except for the presence of $prev = v_2$, would not change since v_2 now has a name.

We compute $fold(m)$ by repeatedly applying the rewrite rules in Figure 7, subject to the *hasname* check, until no more rules are applicable. Note that these rules do produce a valid weakening of the input memory. If we have

$$(H, v_1 \mapsto (v_2, v_3), ls(v_3, v_4); S; P)$$

This can be weakened to

$$\exists v_2, v_3.\ (H, v_1 \mapsto (v_2, v_3), ls(v_3, v_4); S; P)$$

which, by Definition 1 is equivalent to

$$(H, ls(v_1, v_4); S; P)$$

3.4.2 Deciding Weakening

The previous section described how to compute $fold(m)$, a memory which is, by construction, a weakening of m. This operation is used in the invariant inference algorithm to compute candidate invariants. In this section, we address the weakening present in the invariant check (the *while* rule from section 3.4). In this case, we are given two sets of memories M and M' and must decide whether $M \Rightarrow M'$. It is sufficient for our purposes (though not complete in general) to check this by checking that for each $m \in M$ there is some $m' \in M'$ such that $m \Rightarrow m'$. To check this last condition would be easy if we had access to a separation logic theorem prover, as we could simply ask the prover to settle this question. But since we lack such a prover (and in fact are unaware of the existence of such a system), we must do our

own reasoning about the heap. We adopt a rather coarse approach in this case, essentially requiring the heap portions of the memories to be equal. We then use a classical prover to decide entailment between the classical portions of the memories. We now present this approach in detail.

We check that $\exists \bar{v}.\ (H; S; P)$ implies $\exists \bar{v}'.\ (H'; S'; P')$ by searching for a formula H_c, which contains only program variables, such that $\exists \bar{v}.\ (H; S; P) = \exists \bar{v}.\ (H_c; S; P)$ and $\exists \bar{v}'.\ (H'; S'; P') = \exists \bar{v}'.\ (H_c; S'; P')$. We then must show

$$\exists \bar{v}.\ (H_c; S; P) \Rightarrow \exists \bar{v}'.\ (H_c; S'; P')$$

which, since H_c contains only program variables, is equivalent to

$$H_c \wedge (\exists \bar{v}.\ S \wedge P) \Rightarrow H_c \wedge (\exists \bar{v}'.\ S' \wedge P')$$

which is true if

$$\exists \bar{v}.\ S \wedge P \Rightarrow \exists \bar{v}'.\ S' \wedge P'$$

This formula can then be checked by a classical theorem prover. In general, it may contain integer arithmetic and thus be undecidable. However, in section 3.4.4 we present a technique for separating out the portions of the memory that refer to data, which then ensures that we can decide this implication.

We find the above-mentioned heap H_c by rewriting H according to the pointer equalities in S. For each pointer equality $x = v.n$ in S, we solve for v, obtaining $v = x - n$. We then substitute $x - n$ for v throughout H and S. We consider $x = v$ to be shorthand for $x = v.0$. Note that this does force us to add $v - n$ as an allowable pointer expression. However, this causes no issues with the decidability of pointer equalities and inequalities. Equalities in S involving integer expressions are ignored. We call the result of this substitution $pv(m)$ (because it rewrites H in terms of *program variables*) and present a full definition in Figure 8.

As an example, consider the following memories

$$m_1 \equiv \exists v_1, v_2.\ (\mathbf{ls}(v_1, v_2) * \mathbf{ls}(v_2, \mathbf{null}); \\ l = v_1, curr = v_2; v_1 <> v_2)$$

$$m_2 \equiv \exists v_1, v_2.\ (\mathbf{ls}(v_2, v_1) * \mathbf{ls}(v_1, \mathbf{null}); \\ l = v_2, curr = v_1; \cdot)$$

Applying pv to either memory gives us

$$\mathbf{ls}(l, curr) * \mathbf{ls}(curr, \mathbf{null})$$

Since the heaps match, we go on to test whether

$$\exists v_1, v_2.\ l = v_1 \wedge curr = v_2 \wedge v_1 <> v_2 \\ \Rightarrow \exists v_1, v_2.\ l = v_2 \wedge curr = v_1$$

As this is true, we conclude that $m_1 \Rightarrow m_2$.

3.4.3 Fixed Points

We now return to our example of in-place list reversal. We start with the memory

$$\exists v_1.\ (\mathbf{ls}(v_1, \mathbf{null}); curr = v_1, new = \mathbf{null}, old = v_1); \cdot) \quad (1)$$

After one iteration through the loop, we have

$$\exists v_1, v_2, v_3.\ (v_1 \mapsto (v_2, \mathbf{null}), \mathbf{ls}(v_3, \mathbf{null}); \\ curr = v_3, new = v_1, old = v_3; v_1 \neq \mathbf{null}) \quad (2)$$

$$pv(\exists \bar{v}.\ (H; S; P)) = pv(H; S; P)$$
$$pv(H; S, x = v.n; P) = \\ pv(H[v/x - n]; S; P)$$
$$pv(H; S, x = i; P) = pv(H; S; P)$$
$$pv(H; \cdot; P) = H$$

Figure 8: Definition of pv, which rewrites the heap in terms of *program variables*

Applying *fold* at this point has no effect, so we continue with the memory above. After iteration #2, we obtain

$$\exists v_1, v_2, v_3, v_4.\ (v_3 \mapsto (v_4, v_1) * v_1 \mapsto (v_2, \mathbf{null}) * \mathbf{ls}(v_5, \mathbf{null}); \\ curr = v_5, new = v_3, old = v_5; \\ v_3 \neq \mathbf{null} \wedge v_1 \neq \mathbf{null})$$

Since there is no program variable corresponding to v_1, this gets folded to

$$\exists v_1, v_3, v_5.\ (\mathbf{ls}^+(v_3, \mathbf{null}) * \mathbf{ls}(v_5, \mathbf{null}); \\ curr = v_5, new = v_3, old = v_5; \\ v_3 \neq \mathbf{null} \wedge v_1 \neq \mathbf{null}) \quad (3)$$

And this is a fixed point, as can be verified by evaluating the loop body one more time, yielding

$$\exists v_1, v_3, v_5, v_7.\ (\mathbf{ls}^+(v_5, \mathbf{null}) * \mathbf{ls}(v_7, \mathbf{null}); \\ curr = v_7, new = v_5, old = v_7; \\ v_5 \neq \mathbf{null} \wedge v_3 \neq \mathbf{null} \wedge v_1 \neq \mathbf{null}) \quad (4)$$

Let $(H; S; P) = (4)$ and $(H'; S'; P') = (3)$. To verify that we have reached a fixed point, we must show the following

$$\exists v_1, v_3, v_5, v_7.\ H \wedge S \wedge P \Rightarrow \exists v_1, v_3, v_5.\ H' \wedge S' \wedge P'$$

To check this, we compute $pv(H; S; P)$, which is $\mathbf{ls}(new, \mathbf{null}) * \mathbf{ls}(old, \mathbf{null})$. This is the same as $pv(H'; S'; P')$. Thus,

$$\exists v_1, v_3, v_5, v_7.\ H \wedge S \wedge P = \\ \mathbf{ls}^+(new, \mathbf{null}) * \mathbf{ls}(old, \mathbf{null}) \wedge \exists v_1, v_3, v_5, v_7.\ S \wedge P$$

and

$$\exists v_1, v_3, v_5.\ H' \wedge S' \wedge P' = \\ \mathbf{ls}^+(new, \mathbf{null}) * \mathbf{ls}(old, \mathbf{null}) \wedge \exists v_1, v_3, v_5.\ S' \wedge P'$$

Since the heaps are now clearly equal, all that remains is to check that

$$(\exists v_1, v_3, v_5, v_7.\ curr = v_7 \wedge new = v_5 \wedge old = v_7 \wedge \\ v_5 \neq \mathbf{null} \wedge v_3 \neq \mathbf{null} \wedge v_1 \neq \mathbf{null}) \Rightarrow \\ (\exists v_1, v_3, v_5.\ curr = v_5, new = v_3, old = v_5 \wedge \\ v_3 \neq \mathbf{null} \wedge v_1 \neq \mathbf{null})$$

This is easily proved since first-order formulas involving pointer expressions are decidable (using, for example, the decision procedure for Presburger arithmetic [8]).

Finally, recall that the actual loop invariant is the disjunction of every memory leading up to the fixed point. Thus, the full invariant is (1) ∨ (2) ∨ (3).

3.4.4 Integer Arithmetic

So far, we have described how to compare memories in our quest for a fixed point. We also mentioned that in order for $\exists \bar{v}.\ (H; S; P)$ to imply $\exists \bar{v}'.\ (H'; S'; P')$, the implication $\exists \bar{v}.\ S \wedge P \Rightarrow \exists \bar{v}'.\ S' \wedge P'$ must hold. In our example, this implication contained only pointer expressions and so was decidable. In general, S and P may contain information about integer variables, whose arithmetic is sufficiently complex that this question becomes undecidable. To keep our inability to decide this implication from affecting convergence of the algorithm, we weaken the memory after each evaluation of the loop body in such a way that we force S and P to converge. One such approach is to simply drop all information about integer data. After each iteration we replace integer expressions in S with new symbolic variables, so $x = v_3 \times v_2 + v_7$ would become simply $x = v_9$. Similarly, we can drop from P any predicates involving integer expressions. This eliminates the arithmetic and returns our formulas to the realm of decidability. However, it requires us to forget all facts about integers after every iteration. In section 3.5 we describe the use of *predicate abstraction* to carry some of this information over.

The same issue arises with heaps. Consider the following program, which adds the elements in a list using a heap cell to store the intermediate values.

```
[accum] := 0
curr := hd;
while(curr <> null) do {
  s := [curr];
  t := [accum];
  [accum] := s + t;
  curr := [curr.1];
}
```

The memories computed for this program (after applying *fold* and *pv*) will follow the progression:

$$\exists v_1.\ \mathbf{ls}(hd, curr) * \mathbf{ls}(curr, \mathbf{null}) * accum \mapsto v_1$$
$$\exists v_1, v_2.\ \mathbf{ls}(hd, curr) * \mathbf{ls}(curr, \mathbf{null}) * accum \mapsto v_1 + v_2$$
$$\exists v_1, v_2, v_3.\ \mathbf{ls}(hd, curr) * \mathbf{ls}(curr, \mathbf{null}) * accum \mapsto v_1 + v_2 + v_3 \ldots$$

We will never converge if we keep track of the exact value of *accum*. Since we are primarily interested in shape information, we can simply abstract out the data, just as we did for S. We can "forget" what *accum* points to after each iteration by replacing its contents with a fresh symbolic variable. This is equivalent to using the following formula as our candidate invariant

$$\exists v.\ \mathbf{ls}(hd, curr) * \mathbf{ls}(curr, \mathbf{null}) * accum \mapsto v$$

Again, we present a more sophisticated approach in section 3.5.

3.4.5 Invariant Inference in Brief

To summarize, we compute a loop invariant by searching for a fixed point of the symbolic evaluation function plus weakening. We find this fixed point by repeated evaluation of the loop body, starting from the precondition of the loop and weakening the resulting postcondition by applying the *fold* operation and eliminating integer expressions. Convergence is detected by checking that if $M\ [c]\ M'$ then every memory in M' can be weakened to some memory in M. This check involves 1) comparing the heaps for equality after transforming them according to *pv* and 2) checking that $\exists \bar{v}.\ S \wedge P$ for the stronger memory implies $\exists \bar{v}'.\ S' \wedge P'$ for the weaker memory. Once a fixed point is found, the loop invariant is the disjunction of all the memories computed during the search.

3.5 Predicate Abstraction

In the previous section, we presented a method, *fold*, for weakening the heap in order to help guide toward convergence the heaps obtained by repeated symbolic evaluation of a loop body. This did nothing to help the classical portion of the memory converge though, and we ended up just removing all integer formulas from P and S. However, we would like to infer post-conditions that record properties of integer data and doing so requires a better method of approximating P that still assures convergence. One such approach is *predicate abstraction* [10].

Predicate abstraction is an abstract interpretation [7] procedure for the verification of software. The idea is that a set of predicates $P_1, \ldots, P_n$ are provided and the abstraction function finds the conjunction of (possibly negated) predicates that most closely matches the program state. For example, if the predicates are $\{x > 5, y = x\}$ and the state is $x = 3 \wedge y = 3 \wedge z = 2$ then the combination would be

$$\neg(x > 5) \wedge y = x$$

We lose information about z and about the exact values of x and y, but if we provide the right set of predicates, we can often maintain whatever information is important for the verification of the program. Also, the predicates and their negations, together with $\wedge$ and $\vee$, form a finite lattice. So if we have a series of abstractions $A_1, A_2, A_3 \ldots$, which we have obtained from a loop, then the sequence of loop invariant approximations $A_1, A_1 \vee A_2, A_1 \vee A_2 \vee A_3, \ldots$ is guaranteed to converge.

Computing an abstraction A_P of a classical logic formula P is accomplished by asking a theorem prover whether $P \Rightarrow P_i$ for each predicate P_i. If this is true, we include P_i in the conjunction. If, on the other hand, the theorem prover can prove $P \Rightarrow \neg P_i$, then we include $\neg P_i$ in the conjunction. If neither is provable (a possibility since classical logic plus arithmetic is undecidable), then neither P_i nor its negation appear in A_P.

Now that we can compute A_P, we can describe a refinement of the loop invariant inference procedure that maintains more information about the state of integer values in the program. We start with a set of predicates provided by the programmer. These are statements about program variables, such as $x > 0$ or $y = x$. It is this predicate set that forms the basis for our abstraction function. At each step, we take a set of memories M and find the set M' such that $M\ [c]\ M'$, where c is the loop body. For each memory $m'_i \in M'$, with components H'_i, S'_i and P'_i, we perform the *fold* operation and replace symbolic pointer variables in H'_i according to *pv*, as described in the previous section. However, now before dropping integer formulas from P'_i and S'_i, we compute the abstraction $A_{S'_i \wedge P'_i}$. We add this to the candidate invariant so that it has the form $\bar{v}.\ (H; S; P, A)$, where A is the abstraction obtained from S and P.

To detect convergence, we have to be able to decide implication between these memories. To decide whether $\exists \bar{v}.\ (H; S; P, A)$ implies $\exists \bar{v}'.\ (H'; S'; P', A')$, we compare H and

H' for equality as before (by rewriting according to pv). S and P are still free of integer expressions, so the implication $(\exists \bar{v}.\ S \wedge P) \Rightarrow (\exists \bar{v}'.\ S' \wedge P')$ is decidable. Finally, since A and A' contain only program variables, they can be moved outside the existential and checked separately. And since they form a finite lattice, it is easy to check whether $A \Rightarrow A'$, for example by checking that $A \wedge A' = A$.

So now that we can check implication between memories of this form, we can ask whether for each $m'_i \in M'$ there is some $m_i \in M$ such that m'_i implies m_i. If this holds, we have reached a fixed point and are done. If this is not the case, we merge candidate invariants and continue processing the loop body. The merge operation searches for pairs of candidate invariants $\exists \bar{v}.\ (H; S; P, A)$ and $\exists \bar{v}'.\ (H'; S'; P', A')$ such that $H = H'$ and $(\exists \bar{v}.\ S \wedge P) \Rightarrow (\exists \bar{v}'.\ S' \wedge P')$. It then merges these memories into the single memory $\exists \bar{v}.\ (H; S; P, A \vee A')$. This has the effect that once the shape information is fixed, the formulas obtained via predicate abstraction get progressively weaker, ensuring that they will not keep the algorithm from terminating. However, other factors can still impede termination, as we shall see in section 5.

As an example, consider the program below, which adds up the positive elements of a list.

```
curr := hd;
x := 0;
while(curr <> null) {
    t := [curr];
    if( t > 0 ) then x := x + t;
        else skip;
    curr := [curr.1];
}
```

We will take our set of predicates to be $\{x > 0, x = 0, x < 0\}$. We start with the memory

$$(\mathbf{ls}(v_1, \mathbf{null}); hd = v_1, curr = v_2, x = v_3, t = v_4; \cdot)$$

When we reach the "if" statement, we have the memory

$$(v_1 \mapsto v_5 * v_1.1 \mapsto v_6 * \mathbf{ls}(v_6, \mathbf{null}); \\ hd = v_1, curr = v_1, x = 0, t = v_5; \cdot)$$

We can't decide the branch condition, so we evaluate both branches, resulting in two memories at the end of the loop

$$(v_1 \mapsto v_5 * v_1.1 \mapsto v_6 * \mathbf{ls}(v_6, \mathbf{null}); \\ hd = v_1, curr = v_6, x = 0, t = v_5; \neg(v_5 > 0))$$

and

$$(v_1 \mapsto v_5 * v_1.1 \mapsto v_6 * \mathbf{ls}(v_6, \mathbf{null}); \\ hd = v_1, curr = v_6, x = 0 + v_5, t = v_5; v_5 > 0)$$

We then find the conjunction of predicates implied by these memories. In this case, each memory only implies a single predicate. The first memory implies $x = 0$, while the second implies $x > 0$. We then erase the integer data from the memories and keep only these predicates. For example, the memory in the second case becomes

$$(v_1 \mapsto v_5 * v_1.1 \mapsto v_6 * \mathbf{ls}(v_6, \mathbf{null}); \\ hd = v_1, curr = v_6, x = v_7, t = v_5; v_7 > 0)$$

while the first is the same, except that $v_7 = 0$ appears in the P portion. Since we have not reached a fixed point yet, and the heap portion of these two memories are equivalent, we merge them:

$$(v_1 \mapsto v_5 * v_1.1 \mapsto v_6 * \mathbf{ls}(v_6, \mathbf{null}); \\ hd = v_1, curr = v_6, x = v_7, t = v_5; v_7 = 0 \vee v_7 > 0)$$

Another pass through the loop results in two memories, which, after being *fold*ed are again merged to form

$$(\mathbf{ls}(v_1, v_9) * \mathbf{ls}(v_9, \mathbf{null}); \\ hd = v_1, curr = v_9, x = v_{10}, t = v_9; v_{10} = 0 \vee v_{10} > 0)$$

This is a fixed point and because of the information we are maintaining about x, the corresponding loop invariant is strong enough to let us conclude the following postcondition for the program.

$$\mathbf{ls}(hd, \mathbf{null}) \wedge x \geq 0$$

3.5.1 Data in the Heap

The technique just presented allows us to preserve information about stack variables between iterations of the symbolic evaluation loop. However, there is also data in the heap that we might like to say something about. To enable this, we introduce a new integer expression $c(p)$. The function c returns the contents of memory cell p and can be used by the programmer when he specifies the set of predicates for predicate abstraction. We then alter what we do when producing candidate invariants. Rather than replacing integer expressions in the heap with arbitrary fresh variables, we replace them with the appropriate instances of c, and record the substitution as follows. If we start with the memory

$$(v_1 \mapsto 5 * \mathbf{ls}(v_2, \mathbf{null}); accum = v_1, curr = v_2;)$$

Then when we abstract out the data, rather than obtaining

$$\exists v_3.\ (v_1 \mapsto v_3 * \mathbf{ls}(v_2, \mathbf{null}); accum = v_1, curr = v_2;)$$

as we previously would, we instead obtain

$$(v_1 \mapsto c(v_1) * \mathbf{ls}(v_2, \mathbf{null}); accum = v_1, curr = v_2; c(v_1) = 5)$$

If one of our predicates of interest is $c(accum) > 0$, we can ask any theorem prover that can handle uninterpreted functions whether $accum = v_1 \wedge curr = v_2 \wedge c(v_1) = 5 \Rightarrow c(accum) > 0$.

In general, for every heap statement of the form $p \mapsto i$, where i is an integer expression, we replace i by $c(p)$ and record the fact that $c(p) = i$. That is, we apply the following transformation to the memory until it fails to match.

$$(H, p \mapsto i; S; P) \Longrightarrow (H, p \mapsto c(p); S; P, c(p) = i)$$

We then perform predicate abstraction exactly as outlined in the previous section.

3.6 Cycles

We mentioned in section 3.1 that our lists may be cyclic. Here, we explain the reason for this decision. In order to enforce acyclicity, we must insist that the final pointer in a list segment be dangling. That is, whatever segment of the heap satisfies $\mathbf{ls}(p, q)$ cannot have q in its domain. To maintain this property of list segments, we must check that whenever we perform the fold operation, we do not create cycles. Unfortunately, we do not usually have enough information about the heap to guarantee this. In the full paper, we will present examples here of exactly why this is the case.

4. SOUNDNESS

We have proved soundness and this proof will be included in the final paper.

5. COMPLETENESS

While our algorithm works on many interesting examples, it is not complete. A full discussion of completeness, including counter-examples, will be included in the final paper.

6. RESULTS

We have implemented a prototype of our algorithm in about 4,000 lines of SML code. It does it own reasoning about pointers and uses calls to Vampyre [3] when it needs to prove a fact about integers. It implements everything described up to, but not including, the section on predicate abstraction (3.5). We have used this implementation to test the algorithm on several examples, including routines for computing the length of a list, summing the elements in a list, destructively concatenating two lists, deleting a list and freeing its storage, destructive reversal of a list, and destructive partition. The implementation was successful in generating loop invariants fully automatically for all of these examples. We have also worked by hand a number of examples involving predicate abstraction. In this section, we give an example of the invariants produced by our implementation and comment on the issues that arose during testing.

The most difficult program to handle was partition. This routine takes a threshold value v and a list pointer hd. It operates by scanning through the list at hd, passing over elements that are $\geq v$ and shuffling elements that are less than v over to the list at new. The program text is given below.

```
curr := hd;
prev := null;

while (curr <> null) do {
    nextCurr := [curr.1];
    t := [curr];

    if (t < v) {
        if (prev <> null) [prev.1] := nextCurr;
            else skip;
        if (curr = hd) hd := nextCurr; else skip;
        [curr.1] := newl;
        newl := curr;
    }
    else prev := curr;

    curr := nextCurr;
}
```

The difficulty in handling this example comes from the many branches inside the loop body and the interplay between them. For example, note that when $prev = \mathbf{null}$, then $curr = hd$. Thus, there is a relationship between the two innermost "if" statements. Being able to decide branch conditions involving pointers and avoid executing impossible branches (the if_t and if_f rules) were crucial in allowing us to handle this example without generating an invariant containing impossible states.

This example also highlights the importance of keeping track of which lists are known to be non-empty (the $\mathbf{ls}^+$ predicate). When evaluating the loop, after several iterations we arrive at a memory equivalent to the following separation logic formula

$$\exists k.\, \mathbf{ls}(hd, prev) * (prev \mapsto nextCurr) * curr \mapsto (t, nextCurr) \\ * \mathbf{ls}(nextCurr, \mathbf{null}) * \mathbf{ls}(new, \mathbf{null})$$

This is what holds immediately before executing `if (curr = hd)`. Since $\mathbf{prev} \neq \mathbf{null}$ it should be the case that $curr \neq hd$, but this fact does not follow from the formula above. However, if we track the non-emptiness of the list between hd and $prev$, we get a formula of the form

$$\mathbf{ls}^+(hd, prev) * \ldots * curr \mapsto (t, nextCurr) * \ldots$$

Since $\mathbf{ls}^+(hd, prev)$ is non-empty, the portion of the heap that satisfies this list predicate has hd in its domain. And since $*$ separates disjoint pieces of heap, we can conclude that $curr \neq hd$. If we fail to recognize this, we end up erroneously advancing hd, which results in a state in which we have lost the pointer to the head of the list (hd now points somewhere in the middle). Since the program cannot actually reach such a state and it would be quite disturbing to see such a state in an invariant, it is important that we can rule this out.

In the end, the loop invariant inferred for this program is equivalent to the following separation logic formula

$$\begin{aligned} &(\mathbf{ls}(hd, \mathbf{null}) \wedge new = \mathbf{null} \wedge prev = \mathbf{null}) \\ &\vee (\mathbf{ls}(hd, \mathbf{null}) * \mathbf{ls}(new, \mathbf{null}) \wedge prev = \mathbf{null}) \\ &\vee (\exists v_1.\ hd \mapsto (v_1, curr) * \mathbf{ls}(curr, \mathbf{null}) * \mathbf{ls}(new, \mathbf{null})) \\ &\vee (\exists v_1.\ \mathbf{ls}(hd, prev) * prev \mapsto (v_1, curr) \\ &\qquad * \mathbf{ls}(curr, \mathbf{null}) * \mathbf{ls}(new, \mathbf{null})) \end{aligned}$$

The first case in this disjunction corresponds to the loop entry point. The second case is the state after we have put some elements in new, but have not kept any in the list at hd. In the third case, we have kept one element in hd. And in the fourth case, we are in the middle of iterating through the list at hd, with different $prev$ and $curr$ pointers.

We also confirmed a result of Colby et al. [6], which is that failure of the symbolic evaluator can often be helpful in finding bugs. When our symbolic execution algorithm gets stuck, it usually indicates a pointer error of some sort. In such cases, the program path leading up to the failure, combined with the symbolic state at that point, can be a great debugging aid.

7. CONCLUSION

In this paper, we have presented a technique for inferring loop invariants in separation logic [17] for imperative list-processing programs. These invariants are capable of describing both the shape of the heap and its contents. The invariants can also express information about data values both on the heap and in the stack. We have implemented the method and run it on interesting examples.

The examples we have been able to handle are quite encouraging. Still, we are aware of a number of important limitations, some of which have been highlighted in Sections 3.6 and 5. Chief among them is the inability to reason effectively about acyclic lists. Acyclic lists, as discussed in

[2], have the desirable property that they describe a unique piece of the heap. However, as we explain in section 3.6, we cannot apply our *fold* operation to them. In the future, we would like to find better approximations of lists that capture properties such as acyclicity but still allow automation. We would also like to move beyond lists and allow other inductive predicates, ideally allowing for programmer-defined recursive predicates.

Ultimately, it is our intention that such an inference procedure form the foundation for further program verification, using techniques such as software model checking [1, 4, 12] and other static analyses. For example, we would like to incorporate this invariant inference into a software model checker to enable checking of temporal safety and liveness properties of pointer programs.

This framework also makes an ideal starting point for a proof-carrying code system [13, 14]. Since it is based on separation logic, the proofs corresponding to the inference procedure presented here are compositional. A certificate can be generated for code in isolation and it remains a valid proof when the code is run in a different context. However, generation of such certificates requires a proof theory for separation logic, something we are currently lacking. While a proof theory exists for the logic of bunched implications [16], we are not aware of such a system for the special case of separation logic. We would also like to explore the combination of model checking and certification in this framework, as described in [11].

Since detecting convergence of our invariant inference procedure requires checking separation logic implications, we can benefit from any work in separation logic theorem proving and decision procedures for fragments of the logic, such as that given in [2]. It is our hope that the recent surge of interest in separation logic will lead to advances in these areas.

8. ADDITIONAL AUTHORS

9. REFERENCES

[1] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[2] Berdine, Calcagno, and O'Hearn. A decidable fragment of separation logic. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 24, 2004.

[3] David Blei, George Necula, Ranjit Jhala, Rupak Majumdar, et al. Vampyre. http://www-cad.eecs.berkeley.edu/ rupak/Vampyre/.

[4] Sagar Chaki, Edmund Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.

[5] L. A. Clarke and D. J. Richardson. *Symbolic evaluation methods for program analysis*. Prentice-Hall, 1981.

[6] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107, New York, NY, USA, 2000. ACM Press.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[8] Jeanne Ferrante and Charles W. Rackoff. *The Computational Complexity of Logical Theories*. Lecture Notes in Mathematics. Springer, 1979.

[9] Rakesh Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 1–15, 1996.

[10] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *9th International Conference on Computer Aided Verification (CAV)*, pages 72–83, London, UK, 1997. Springer-Verlag.

[11] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *14th International Conference on Computer Aided Verification (CAV)*, pages 526–538. Springer-Verlag, 2002.

[12] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *ACM Symposium on Principles of programming languages (POPL)*, pages 58–70, New York, NY, USA, 2002. ACM Press.

[13] George C. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Langauges (POPL)*, pages 106–119, Paris, January 1997.

[14] George C. Necula and Peter Lee. Safe Kernel extensions without run-time checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 229–244, Berkeley, October 28–31 1996. USENIX Association.

[15] Charles Gregory Nelson. *Techniques for program verification*. PhD thesis, Stanford University, 1980.

[16] D.J. Pym. *The Semantics and Proof Theory of the Logic of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[17] John C. Reynolds. Separation logic: A logic for shared mutable data structures. *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[18] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Transactions on Programming Languages and Systems*, 2002.

[19] Élodie-Jane Sims. Extending separation logic with fixpoints and postponed substitution. In *AMAST: Algebraic Methodology and Software Technology, 10th International Conference*, pages 475–490. Springer, 2004.

[20] T. Weber. Towards mechanized program verification with separation logic. In *CSL: 18th Workshop on Computer Science Logic*. LNCS, Springer-Verlag, 2004.