Shape Analysis with Inductive Recursion Synthesis

Bolei Guo Neil Vachharajani David I. August
Department of Computer Science
Princeton University {bguo,nvachhar,august}@princeton.edu

Abstract

Separation logic with recursively defined predicates allows for concise yet precise description of the shapes of data structures. However, most uses of separation logic for program analysis rely on pre-defined recursive predicates, limiting the class of programs analyzable to those that manipulate only a priori data structures. This paper describes a general algorithm based on *inductive program synthesis* that automatically infers recursive shape invariants, yielding a shape analysis based on separation logic that can be applied to any program.

A key strength of separation logic is that it facilitates, via explicit expression of structural separation, local reasoning about heap where the effects of altering one part of a data structure are analyzed in isolation from the rest. The interaction between local reasoning and the global invariants given by recursive predicates is a difficult area, especially in the presence of complex internal sharing in the data structures. Existing approaches, using logic rules specifically designed for the list predicate to unfold and fold linked-lists, again require a priori knowledge about the shapes of the data structures and do not easily generalize to more complex data structures. We introduce a notion of "truncation points" in a recursive predicate, which gives rise to generic algorithms for unfolding and folding arbitrary data structures.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages, Theory

Keywords Shape analysis, separation logic, loop invariant inference, inductive recursion synthesis, artificial intelligence

1. Introduction

Shape analysis aims at an accurate description of the program heap layout, which can enable aggressive optimizations, program verification, and program understanding tools. With the prevalent use of dynamic memory allocation, a heap abstraction must have some way of describing infinite number of concrete heaps with a finite representation. Among such techniques are *summary node* [1] which groups elements of potentially unbounded data structures into a finite number of abstract heap nodes, and *k-limiting* [2] which only distinguishes elements of a linked data structure up to depth k. In both cases, the approximation of memory states leads to loss of information about the shapes of recursive data structures. Inductively defined predicates such as those used in separation logic [3] allow for concise yet precise description of recursive data structures. For example, an acyclic linked-list is captured by $\mathit{list}(x) \doteq (x = \mathit{null} \wedge \mathbf{emp}) \vee (x \rightarrow \alpha * \mathit{list}(\alpha))$. Although expressive, a problem with these predicates is that it is difficult to infer them from the program. As a result, current uses of separation logic usually have a handful of pre-defined predicates hardwired into the logic and are limited to program verification where the logic engine is supplied with user specifications that a predicate holds at certain program point. In the case of linked-lists, logic rules can be designed to recognize certain patterns in the logic formulae and rewrite them to synthesize the list predicate. Two analyses of list-processing programs are proposed [4, 5], both containing a rule that says if x points to y and y points to z then there is a list segment between x and z. It is difficult to generalize this to arbitrary data structures. Lee et al. propose a grammar-based shape analysis [6] that automatically discovers grammars which can be translated to recursive predicates. However, their grammars can have only one explicit parameter, limiting the class of data structures describable.

We propose a shape analysis that performs *inductive recursion synthesis* to automatically infer arbitrary recursive predicates, effectively reverse-engineering the data types in the program. This technique leverages an existing method in artificial intelligence called *inductive program synthesis*, originally developed for constructing recursive logic programs from sample input/output pairs [7, 8]. It allows the analysis to extract a loop invariant from a constant number of symbolically executed loop iterations. Soundness is guaranteed by verifying that the invariant derives itself over the loop body. If so, then it allows the analysis to converge over the loop and proceed, but unlike many widening operations used to reach fixed points, there is no approximation involved and hence no loss of precision. Otherwise, the analysis will halt and report failure. This technique can infer any data type with a tree-like backbone and some other pointer fields that point in the backbone, possibly producing dags and cycles. This gives our analysis the same descriptive power as the Pointer Assertion Logic [9]. However, in their framework shape invariants are already given by non-traditional data type declarations and the logic engine relies on user specifications including procedure pre- and post-conditions, and loop invariants. our analysis starts with zero knowledge and infers everything, the data types, the procedure summaries, and the loop invariants from scratch. Inductive recursion synthesis is also used to converge over recursive procedures.

Another difficulty in using recursive predicates lies in the fact that while they express global properties that hold over entire data structures, most programs perform many local alterations (insertions, deletions, rotations, etc.) to the data structures and re-establish global properties afterwards. Ideally, the analysis should be able to zoom in on a small part of a data structure, reason about it ignoring the rest, and then zoom back out. The spatial conjunction operator of separation logic is designed to facilitate this kind of local reasoning via explicit expression of structure separation

PLDI'07 June 11–13, 2007, San Diego, California, USA.

and aliasing. However, for data structures with complex internal sharing, isolating a sub-structure separate from the rest is difficult. In [4, 5], logic rules tailored to the list predicate can unfold a list to expose a list element and fold it back. But, again, this does not easily generalize to other data structures. To enable smooth transitioning between local reasoning and global invariants, we introduce the notion of "truncation points" in a recursive predicate, which helps the analysis to cut corners out of a data structure. Generic algorithms based on truncation points are then designed to unfold and fold arbitrary recursive data structures.

The shape analysis presented in this paper is interprocedural. Like the analysis by Gotsman et al. [10], at each procedure entry, it extracts the region of heap accessed by the procedure, called *local heap*, from the rest of the heap and, upon return, re-incorporates this updated local heap using the Frame rule of separation logic. *Cutpoints* [11], the nodes that separate the local heap from the frame, are preserved so that upon return the callee's effects can be properly propagated to the caller. In the presence of recursive procedures, the number of cutpoints can be infinite. [10] bounds the number of cutpoints at the cost of potential precision loss. In our case, inductive recursion synthesis allows cutpoints to be described inductively in the entry/exit invariants of recursive procedures, hence there is no need to bound them.

Our goal is to handle real-life C programs like those in the SPEC benchmarks, whose data structures are complex and cannot be easily taken apart into independent pieces. We will use as a running example the benchmark 181.mcf from SPEC2000, which builds and manipulates a left-child right-sibling tree with two kinds of backward links – a parent link and a left-sibling link. As shown in Figure 1, there is a great degree of internal sharing which makes both inferring its shape and reasoning about its shape challenging. Additionally, many applications perform their own memory management using arrays. To model this correctly, our analysis tracks aliasing that arises from pointer arithmetic. Finally, the algorithm performs a pre-pass including a fast pointer analysis and program slicing to preserve only code that may affect the result of shape analysis. This effectively reduces the overhead of being flow-sensitive on realistic programs, which is important because flow-sensitivity allows strong updates, key to shape analysis. This also reduces noises that may confuse the inductive recursion synthesis algorithm.

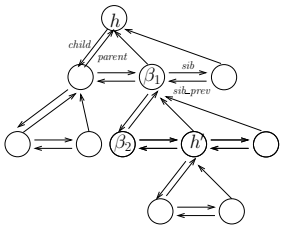


Figure 1. A specimen of the tree used in 181.mcf

In summary, the contributions of this work are:

- An algorithm for inferring recursive shape invariants based on inductive program synthesis.
- A general algorithm for unrolling and rolling back arbitrary recursive data structures, even those with internal sharing.
- Techniques to handle real-life applications.

Section 2 defines the abstract semantics. Section 3 describes inductive recursion synthesis, and Section 4 presents the algorithm for unrolling and rolling recursive predicates. Section 5 describes the implementation of our analysis. Preliminary test results are reported in Section 6. Section 7 discusses related work and finally, Section 8 concludes.

2. Abstract States and Abstract Operational Semantics

Labels	l	$\in$	$Label$
Globals	g	$\in$	$Global$
Registers	r	$\in$	Reg
Exprs	e	$::=$	**null** $\mid g \mid r$
Insts	s	$::=$	$r = e \mid r =$ **malloc**$() \mid$ **free**$(r) \mid r = f(\vec{x}) \mid$ $[r_1] = r_2 \mid r_1 = [r_2] \mid$ **goto** $l \mid$ **if** c **goto** l
Branch Conds	c	$::=$	$r_1 = r_2 \mid r_1 \neq r_2$
Vars	α	$\in$	Var
Recursion vars	A	$\in$	Rec
Heap names	h	$::=$	$g \mid \alpha \mid h.n$
Symbolic vals	v	$::=$	$null \mid h \mid h + n$
Pure assertions	P	$::=$	$v_1 = v_2 \mid v_1 \neq v_2$
Heap assertions	H	$::=$	$h_1.n \rightarrow h_2 \mid A(h_1, ..., h_n[; h'_1, ..., h'_m])$
Register values	Π	$::=$	$\oslash \mid \Pi, r = v$
Heap formulae	Σ	$::=$	**emp** $\mid H \mid \Sigma * \Sigma$
Pure formulae	Φ	$::=$	$P \mid \Phi \wedge \Phi$
States	S	$::=$	$\Pi \mid \Sigma \mid \Phi$
Predicate defs	Θ	$::=$	$\oslash \mid \Theta, A \doteq P \vee \Sigma$
Proc summaries	Γ	$::=$	$\oslash \mid \Gamma, (f, \Pi_{entry} \mid \Sigma_{entry} \mid \Phi_{entry}, \Pi_{exit} \mid \Sigma_{exit} \mid \Phi_{exit})$

Table 1. Target Language and Abstract States

The target language of our analysis is an assembly like low-level intermediate language used by our optimizing C compiler. The syntax of the language is shown in Table 1. Globals are names of heap locations allocated for global variables. Our analysis also handles instructions such as $r_1 = r_2 * n$ and $r_1 = r_2 + n$, which perform pointer arithmetic. For simplicity of presentation, they are not included in the discussion here. Detailed treatment of arrays and pointer arithmetic is based on the low-level pointer analysis by Guo et al. [12]. The symbolic values computed in this analysis contain offset information. Indistinguishable array elements are collapsed into one element.

The rest of this section describes the abstract representation of states and an abstract operational semantics of the instructions tailored to the unstructured control flow of machine-level code.

2.1 Abstract States

An *abstract state* $\Pi \mid \Sigma \mid \Phi$ consists of a mapping Π from registers to their symbolic values, a separation logic formula Σ that is the conjunction of a finite number of atomic heap assertions, and a pure formula Φ that records true branch conditions along the execution paths with which the state is associated and also records aliasing between pointer arithmetic and heap names. Two global environments are maintained: Θ for recording the definitions of recursive predicates and Γ for tabulating procedure summaries. Table 1 gives the definition of the state.

Unlike the "symbolic heaps" defined in [13], which use program variables (the high-level counterpart of registers) to name heap locations and record alias relationships between program variables, our analysis takes the "points-to" approach, assigning unique names to heap locations and recording the target of each register explicitly. Aside from the benefit that there is no need for "rearrangement" rules which set the pre-state in a suitable form by going through the alias pairs, this facilitates the inductive recursion synthesis algorithm. As will be explained in Section 3, the access-path- like heap names encode important patterns of spatial relationships between heap locations that can be recognized and then generalized via inductive reasoning. The heap names can be simply thought of as logic variables with long names.

Recursive predicates are parameterized so that they are expressive enough to describe data structures with internal sharing via backward links. The first parameter represents the top of the data structure and the rest represent the targets of backward links. For example, the left-child right-sibling tree with parent and left-sibling

links from 181.mcf can be written as:

$$\begin{aligned}\textit{mcf\_tree}(x_1, x_2, x_3) \doteq (x_1 = \textit{null} \wedge \mathbf{emp}) \vee \\ (x_1.\textit{parent} \rightarrow x_2 * x_1.\textit{child} \rightarrow \alpha * \textit{mcf\_tree}(\alpha, x_1, \textit{null}) * \\ x_1.\textit{sib\_prev} \rightarrow x_3 * x_2.\textit{sib} \rightarrow \beta * \textit{mcf\_tree}(\beta, x_2, x_1)).\end{aligned}$$

An instance of such a tree where the root h has null *parent* and *sib_prev* links is described by instantiating the predicate: *mcf_tree*$(h, \textit{null}, \textit{null})$.

We also introduce a new type of recursive predicates called *truncated recursive predicates*, $A(h_1, ..., h_n; h'_1, ..., h'_m)$ where n is the arity of A. This is designed for handling modifications to a data structure when the analysis needs to isolate relevant parts of the data structure so as to reason about the modifications in a localized fashion. The second set of parameters $\{h'_1, ..., h'_m\}$ is of variable length and is what we call the set of *truncation points* in the data structure rooted at h_1. This predicate is syntactic sugar for $(*_{i=1..m} \exists \beta_{i,1}, ..., \beta_{i,n-1}.A(h'_i, \beta_{i,1}, ..., \beta_{i,n-1})) -\!\!* A(h_1, ..., h_n)$ (both the iterated spatial conjunction operator and the "magic wand" operator $-\!\!*$ are defined in [3]). It identifies a heap that, when combined with m heaps rooted at $h'_1, ..., h'_m$ on each of which A holds, yields a heap rooted at h_1 on which A holds. In other words, this is the data structure reachable from h_1, with all sub-graphs rooted at $h'_1, ..., h'_m$ cut out from it. Because the predicates are "precise" [14] in that each unambiguously identifies a piece of heap, when $A(h_1, ..., h_n)$ holds on the combined piece of heap, it must go through all the nodes in it. Hence it is impossible to have a situation where the truncation points do not truncate the data structure at all, ensuring the correctness of our definition. The definition also specifies that the sub-graphs are mutually disjoint, i.e. no truncation point can be in the sub-graph of another truncation point. This invariant is crucial for unrolling predicates as it constrains the number of possible outcomes (details are in Section 4). In Figure 1, suppose that at some program point, there is a pointer to an interior node h' of the tree, then the heap is described as *mcf_tree*$(h, \textit{null}, \textit{null}; h') * \textit{mcf\_tree}(h', \beta_1, \beta_2)$. By the definition of *mcf_tree*, we know that the dangling points β_1 and β_2 of the heap *mcf_tree*(h', β_1, β_2) are backward links and therefore reside in the other half of the heap.

A linked-list fragment between x and y can be described by *list*$(x; y)$, which looks similar to the "list segment" predicate *list*$(x, y) \doteq (x = y \wedge \mathbf{emp}) \vee (x \rightarrow \alpha * \textit{list}(\alpha, y))$ defined in [15]. However, this predicate is defined by specifying a path by which y is reached from x and is therefore hard to generalize to more complex data structures, whereas we avoid this complication entirely by working not from the top of the data structure, but from the bottom, and hiding the reaching path information with the "magic wand". Not only is our approach completely general and capable of handling messy backward links, it is also more flexible by allowing a variable number of truncation points. This is important because unlike lists, other data structures may have more than one end. The ability to model this comes in handy, for example, when cutting and grafting sub-trees.

Let $[\![]\!]_{\Pi,\Phi}$ be a function that evaluates each expression e to a heap name or *null*.

$$[\![\mathbf{null}]\!]_{\Pi,\Phi} = \textit{null} \qquad [\![g]\!]_{\Pi,\Phi} = g$$

$$[\![r]\!]_{\Pi,\Phi} = \begin{cases} h' & \text{if } \Pi(r) = h + n \text{ and } \Phi \text{ records the alias } h + n = h' \\ \alpha & \text{if } \Pi(r) = h + n \text{ and } \Phi \text{ records no alias of } h + n, \\ & \alpha \text{ is a fresh variable} \\ \Pi(r) & \text{otherwise} \end{cases}$$

The partial order $\sqsubseteq$ over the set of abstract states is defined as follows: $\Pi_1 \mid \Sigma_1 \mid \Phi_1 \sqsubseteq \Pi_2 \mid \Sigma_2 \mid \Phi_2$ if there exists a mapping f between the heap names in the two states such that *(i)* for each $r \in \textit{Domain}(\Pi_1)$, if $[\![r]\!]_{\Pi_1,\Phi_1} = \textit{null}$, then $[\![r]\!]_{\Pi_2,\Phi_2} = \textit{null}$; otherwise $f([\![r]\!]_{\Pi_1,\Phi_1}) = [\![r]\!]_{\Pi_2,\Phi_2}$, and *(ii)* for each atomic H in Σ_1, $f^\dagger(H)$ is in Σ_2, $f^\dagger$ replaces each h appearing in H with $f(h)$, and *(iii)* for each atomic P in Φ_1, $f^\ddagger(P)$ is in Φ_2, $f^\ddagger$ replaces each h in P with $f(h)$. Obviously there could be infinitely increasingly chains of abstract states. Termination of the analysis is achieved via inductive recursion synthesis.

2.2 Abstract Operational Semantics

We give the abstract operational semantics for the target language in the style of $\mathcal{L}_c$, a compositional logic for control flow [16], with some modifications. Most noticeably, our logic rules are written for forward analysis while those in [16] are for backward analysis. The judgment we use is: $\Psi, F \vdash \Psi'$. F is a set of program fragments $l(s)l'$, with label l identifying the entry of instruction s and l' identifying the exit. Ψ and Ψ' are sets of labeled states: $\Psi = \{l_1 : S_1, ..., l_n : S_n\}, \Psi' = \{l'_1 : S'_1, ..., l'_m : S'_m\}$. Labels $l_1, ..., l_n$ are where the control flow may enter F and labels $l'_1, ..., l'_m$ are where it may leave F. The judgment is read as: if for $i = 1..n$, the state at entry l_i is S_i, and the execution of F does not get stuck, then for $j = 1..m$, the state at exit l'_j is S'_j.

Table 2 lists the operational rules that transform entry states of a program fragment to exit states. It includes one rule for each primitive instruction, composition rules **COMBINE**, **DISCHARGE** and **WEAKEN** for combining individual instructions, and the rule **UNFOLD** for unrolling a recursive predicate to reveal a points-to fact. Note, the rules shown in the table perform strong updates to the abstract state. In the case of aliasing due to array elements collapsed into a single heap element, the analysis would have to use an alternate set of rules which perform weak updates.

In the rule **MALLOC**, $\alpha.? \rightarrow ?$ simply registers α as an allocated heap node whose content is unknown.

MUTATE invokes an important sub-routine *rearrange_names*, shown in Figure 2, to encode access-path info in heap names. The recursion synthesis algorithm relies on this to identify the basic structure of a recursion. *rearrange_names* assumes that the current heap satisfies $h_1.n \rightarrow h_2$ and that v is to be written to $h_1.n$. The appropriate name for v is determined based on its form:

- If it is a simple variable, then we assign $h_1.n$ as its new name. If the old content stored in field n of h_1 has already claimed this name, then the old content is renamed to a fresh variable.
- If it is a heap name plus an offset, then it points to the middle of a structure, most likely an array element. As in the first case, $h_1.n$ is assigned as its new name. Additionally, the analysis records in Φ that the pointer arithmetic aliases with $h_1.n$ so that if later the location is visited via pointer arithmetic instead of access path, the analysis will recognize it as well.
- Otherwise, v points to a heap location that has already been linked to a parent, and no special action is necessary.

The intuition behind this is: While a heap location may be reachable via multiple access paths (one data structure may contain cross pointers to another data structure; or, within a single data structure, a node may be internally shared in the presence of dags and cycles), the algorithm chooses the access path that reveals the acyclic backbone of the recursive data structure to which the location belongs. Our heuristic is to inherit the access path of first location it is linked to, taking advantage of the fact that such a link is usually created when adding a new expansion to a recursive data structure.

The rule **PROC_CALL** is the same as the one given in [10]. It exploits the Frame rule by breaking the heap at a call site into the "local heap" accessed by the callee and a frame. σ is a mapping between the formal parameters and the actuals, and between the return value and destination register of the call instruction. **PROC_CALL** is understood as follows: If there exists in Γ a recorded summary of the callee, $(f, \Pi_{\textit{entry}} \mid \Sigma_{\textit{entry}} \mid \Phi_{\textit{entry}}, \Pi_{\textit{exit}} \mid$

rearrange_names(h_1, n, h_2, v)
if $v = a$ **then**
  if $h_2 = h_1.n$ **then**
    replace h_2 everywhere with a fresh variable
  replace v everywhere with $h_1.n$
  return $h_1.n$
else
  if $v = h + n$ **then**
    if $h_2 = h_1.n$ **then**
      replace h_2 everywhere with a fresh variable
    record alias $\langle h + n, h_1.n \rangle$
    return $h_1.n$
  return v

Figure 2. Algorithm of *rearrange_names*

$\Sigma_{exit} \mid \Phi_{exit}$), and the current heap Σ can be separated into disjoint pieces $\sigma(\Sigma_{entry})$ and R, then the heap after the call instruction is a conjunction of $\sigma(\Sigma_{exit})$ and R; and r is assigned the return value translated by σ (*ret* is special register for holding return values). Since we are not concerned with bounding the number of cutpoints, they are simply treated as dangling points from the frame.

The rule responsible for termination of the analysis is **normalize**. There are two kinds of normalization operations. From a subheap, **synthesis** infers a recursive description that is guaranteed to be more general. **fold** reduces the size of the state by folding surrounding heap nodes into a recursive predicate.

3. Inferring Recursive Predicates

This section describes the algorithm for automatically inferring recursive predicates. It enables the analysis to arrive at loop invariants without introducing unnecessary approximation. Loop invariant inference proceeds in the following steps:

1. Symbolically execute the loop body up to a fixed number of times (2 suffices in the experimentation).
2. If the analysis does not converge over the loop at this point, then invoke inductive recursion synthesis, which returns a hypothesized loop invariant.
3. Verify the soundness of the loop invariant by assuming that it holds on loop entry and checking that for each control flow path in the loop, if Σ' is the heap at the end of the path, then $fold_{\Theta}(\Sigma') = \Sigma_{invariant}$. If the analysis diverges, then halt and report failure.
4. Otherwise, the loop invariant is valid. By the algorithm of recursion synthesis, the states associated with the loop entry in the initial number of iterations must be derivable from the invariant by unrolling it. Hence they are eliminated using the **WEAKEN** rule.

3.1 The Inductive Recursion Synthesis Algorithm

Inductive program synthesis, the problem of automatic synthesis of recursive programs from input/output samples, studied in AI research, resembles loop invariant inference in the sense that the input/output samples are provided by finite executions of the loop and the invariant can be seen as a highly abstracted encoding of the loop. The approach introduced by Summers [7] consists of two steps. First, the input/output samples are rewritten as finite program traces, then a recurrence relation is identified by inspecting the traces. In our case, the program trace is readily available as the heap formula after execution of the loop, with crucial information encoded in the logic variable names by *rearrange_names* in Section 2. The logic formula is translated into a *term*, the form of inputs on which the recurrence detection algorithm operates, using the domain knowledge about heap semantics. Such global inspection of states is only conducted when converging over loops. The rest of the analysis updates states locally.

3.1.1 Translating Heap Formulae into Terms

The set of *terms* is defined in Figure 3. A term can be viewed as a tree where each symbol is a node.

Terms	t	$::=$	x	variables
		$\mid$	c	constants
		$\mid$	$f(t_1, .., .t_n)$	functions

Figure 3. Set of Terms

The idea is to map each heap location to a term that describes the data structure reachable from it, referred to as a "heap" term. We start by assigning a function symbol to each logic operator, $*$ for spatial conjunctions, $\xrightarrow{n}$ for points-to assertions with n being the field, and the predicate name for predicate instantiations. As the heap locations are interconnected with each other, naturally some terms will be sub-trees of other terms. While the heap may contain dags and cycles, the term tree structure must remain acyclic (consistent with the fact that the backbones of inductive definitions are acyclic). To achieve this, each heap location is also associated with a "name" term. For all appearances of a heap location on the right hand side of a points-to assertion, only one will result in the corresponding heap term being linked as a sub-tree of the left hand side. All others are translated into name terms by the rewrite function $[]$, cutting the points-to link in a sense.

$$[\mathit{null}] = \text{NULL} \qquad [g] = g \qquad [\alpha] = \alpha \qquad [h.n] = n([h]).$$

The translation process maintains a mapping γ from heap locations to heap terms. $[]$ is overloaded to translate heap formulae to terms.

$$\begin{aligned}
&[A(h_1, ..., h_n[; h'_1, ..., h'_m])] = \gamma(h_1) = \\
&\qquad A([h_1], ..., [h_n][; [h'_1]..., [h'_m])) \\
&[h.n_1 \rightarrow h_1 * ... * h.n_r \rightarrow h_r] = \gamma(h) = \\
&\qquad *(\xrightarrow{n_1} ([h], \mathit{get_term}(h, h_1)), ..., \xrightarrow{n_r} ([h], \mathit{get_term}(h, h_r))), \\
&\qquad \mathit{get_term}(h_1, h_2) = \begin{cases} \gamma(h_2) & \text{if } h_2 = h_1.n \\ [h_2] & \text{otherwise} \end{cases}
\end{aligned}$$

In a depth-first traversal of the abstract heap, every predicate instantiation is translated into the heap term of the first parameter; all points-to assertions with the same location on the left hand side are translated together into the heap term of that location. The choice between a heap term and a name term for the right hand side is guided by the access paths encoded in the names of the heap locations. The final result is a forest of top-level term trees and because of the heuristic adopted in *rearrange_names*, each of these trees roughly corresponds to a different data structure in the program.

Figure 4(a) contains a loop from 181.mcf that builds the left-child right-sibling tree with backward links. `nodes` is an array of tree nodes. All new tree nodes are subsequently requested from it. Figure 4(b) shows the term tree after two iterations of the loop. Each $*$ term represents a distinct node in the data structure, whose name is given in the parenthesis next to it. For each field n in the node, the corresponding $*$ term contains $\xrightarrow{n}$ term whose left sub-term is the name of the source location and the right sub-term is either the name of the target location or the $*$ term representing the target location. In the latter case, the expansion of the data structure is continued from below the $*$ term. In the former case, the data structure reachable from the target location will be expanded along some other access path reaching that target. The term in Figure 4(b) completely captures the effect of the loop on heap at this execution point and presents it in such a way that exposes the underlying recursive pattern to the recurrence detection algorithm.

$$\frac{}{\{l:\Pi \mid \Sigma \mid \Phi\}, \{l\ (r=e)\ l'\} \vdash \{l':\Pi[r=[\![e]\!]_{\Pi,\Phi}] \mid \Sigma \mid \Phi\}}\ \textbf{ASSIGN}$$

$$\frac{}{\{l:\Pi \mid \Sigma \mid \Phi\}, \{l\ (r=\textbf{malloc}())\ l'\} \vdash \{l':\Pi[r=\alpha] \mid \Sigma * \alpha.? \rightarrow ? \mid \Phi\}}\ ,\alpha \text{ fresh } \textbf{MALLOC}$$

$$\frac{}{\{l:\Pi, r=h \mid \Sigma * H(h) \mid \Phi\}, \{l\ (free(r))\ l'\} \vdash \{l':\Pi, r=h \mid \Sigma \mid \Phi\}}\ ,H(h) ::= h.n \rightarrow h' \mid A(h,...)\ \textbf{FREE}$$

$$\frac{}{\{l:\Pi, r_1=h_1+n \mid \Sigma * h_1.n \rightarrow h_2 \mid \Phi\}, \{l\ (r_2=[r_1])\ l'\} \vdash \{l':\Pi, r_1=h_1+n\ [r_2=h_2] \mid \Sigma * h_1.n \rightarrow h_2 \mid \Phi\}}\ \textbf{LOOKUP}$$

$$\frac{h_2' = \mathit{rearrange\_names}(h_1, n, h_2, v)}{\{l:\Pi, r_1=h_1+n, r_2=v \mid \Sigma * h_1.n \rightarrow h_2 \mid \Phi\}, \{l\ ([r_1]=r_2)\ l'\} \vdash \{l':\Pi, r_1=h_1+n, r_2=h_2' \mid \Sigma * h_1.n \rightarrow h_2' \mid \Phi\}}\ \textbf{MUTATE}$$

$$\frac{}{\{l:S\}, \{l\ (\textbf{goto}\ l_1)\ l'\} \vdash \{l_1:S\}}\ \textbf{JUMP} \qquad \frac{}{\{l:S\}, \{l\ (\textbf{if}\ c\ \textbf{goto}\ l_1)\ l'\} \vdash \mathit{filter}(c)(l_1:S) \cup \mathit{filter}(\neg c)(l':S)\}}\ \textbf{BRANCH}$$

$$\frac{\Gamma \vdash (f, \Pi_{\mathit{entry}} \mid \Sigma_{\mathit{entry}} \mid \Phi_{\mathit{entry}}, \Pi_{\mathit{exit}} \mid \Sigma_{\mathit{exit}} \mid \Phi_{\mathit{exit}}) \quad \Sigma \models \sigma(\Sigma_{\mathit{entry}}) * R}{\{l:\Pi \mid \Sigma \mid \Phi\}, \{l\ (r=f(\vec{x}))\ l'\} \vdash \{l':\Pi[r=\sigma(\Pi_{\mathit{exit}}(\mathit{ret}))] \mid \sigma(\Sigma_{\mathit{exit}}) * R \mid \Phi\}}\ \textbf{PROC\_CALL}$$

$$\frac{\mathit{unfold}_{\Theta}(l:S,h), F \vdash \Psi'}{\{l:S\}, F \vdash \Psi'}\ \textbf{UNFOLD}$$

$$\frac{\Psi_1, F_1 \vdash \Psi_1' \quad \Psi_2, F_2 \vdash \Psi_2'}{\Psi_1 \cup \Psi_2, F_1 \cup F_2 \vdash \Psi_1' \cup \Psi_2'}\ \textbf{COMBINE} \qquad \frac{\Psi \cup \{l:S\}, F \vdash \Psi' \cup \{l:S\}}{\Psi, F \vdash \Psi' \cup \{l:S\}}\ \textbf{DISCHARGE} \qquad \frac{\Psi, F \vdash \Psi_2' \quad \Psi_2' \Rightarrow \Psi_1'}{\Psi, F \vdash \Psi_1'}\ \textbf{WEAKEN}$$

$$\frac{\Psi_1 \supseteq \Psi_2}{\Psi_1 \Rightarrow \Psi_2}\ \textbf{superset} \qquad \frac{\Sigma' \rightsquigarrow \Sigma}{\Psi \cup \{l:\Pi \mid \Sigma \mid \Phi\} \Rightarrow \Psi \cup \{l:\Pi \mid \Sigma' \mid \Phi\}}\ \textbf{normalize}$$

$$\frac{\mathit{recursion\_synthesis}(\Sigma_1) = A(h_1,...,h_n[;\alpha_1,...,\alpha_m])}{\Sigma * \Sigma_1 \rightsquigarrow \Sigma * A(h_1,...,h_n[;\alpha_1,...,\alpha_m])}\ \textbf{synthesis} \qquad \frac{\mathit{fold}_{\Theta}(\Sigma_1) = A(h_1,...,h_n[;\alpha_1,...,\alpha_m])}{\Sigma * \Sigma_1 \rightsquigarrow \Sigma * A(h_1,...,h_n[;\alpha_1,...,\alpha_m])}\ \textbf{fold}$$

$$\mathit{filter}(r_1=r2)(l:\Pi \mid \Sigma \mid \Phi) = \begin{cases} \{l:\Pi \mid \Sigma \mid \Phi \wedge [\![r1]\!]_{\Pi,\Phi} = [\![r2]\!]_{\Pi,\Phi}\} & \text{if } \Phi \not\vdash [\![r1]\!]_{\Pi,\Phi} \neq [\![r2]\!]_{\Pi,\Phi} \\ \oslash & \text{otherwise} \end{cases}$$

$$\mathit{filter}(r_1 \neq r2)(l:\Pi \mid \Sigma \mid \Phi) = \begin{cases} \{l:\Pi \mid \Sigma \mid \Phi \wedge [\![r1]\!]_{\Pi,\Phi} \neq [\![r2]\!]_{\Pi,\Phi}\} & \text{if } \Phi \not\vdash [\![r1]\!]_{\Pi,\Phi} = [\![r2]\!]_{\Pi,\Phi} \\ \oslash & \text{otherwise} \end{cases}$$

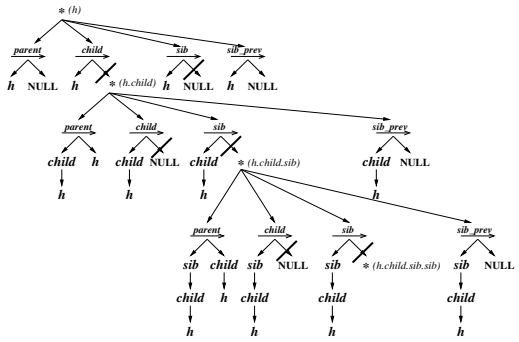
Table 2. Abstract Operational Semantics

```
    nodes = malloc(MAX_NODES);

    root = nodes;
    node = nodes+1;
    root->parent = null;
s1: root->child = node;
    root->sib = null;
    root->sib_prev = null;

    for (...) {
        node->parent = root;
        node->child = null;
s2:     node->sib = node+1;
s3:     node->sib_prev = node-1;
        node++;
    }
```

(a) Code



(b) Term tree

$$g(x_1,x_2,x_3) = *(\xrightarrow{\mathit{parent}}(x_1,x_2), \xrightarrow{\mathit{child}}(x_1,g), \xrightarrow{\mathit{sib}}(x_1,g), \xrightarrow{\mathit{sib\_prev}}(x_1,x_3))$$

$$g(x_1,x_2,x_3) = *(\xrightarrow{\mathit{parent}}(x_1,x_2), \xrightarrow{\mathit{child}}(x_1, g(\mathit{child}(x_1), x_1, \text{NULL})), \xrightarrow{\mathit{sib}}(x_1, g(\mathit{sib}(x_1), x_2, x_1)), \xrightarrow{\mathit{sib\_prev}}(x_1,x_3))$$

(c) Recurrence

Figure 4. A loop in 181.mcf that builds its tree

This cannot be achieved by ordinary separation logic formulae without the enhancement of access-path-based heap names or the domain-specific translation into terms.

3.1.2 Recurrence Detection

For convergence over loops, the recurrence detection algorithm is applied to each top-level term (a loop may touch multiple data structures). The algorithm we build on is by Schmid [8]. The high-level intuition is that if there is a recurrence relation that explains a term, then the term can be obtained from the recurrence relation by unfolding the recursion body up to a finite length. So a term can be folded into a recursion by finding a segmentation of the term corresponding to the unfolding points, together with parameter substitution rules. As pointed out by Summers [7], this can be viewed as the converse of using fixed points to give the semantics of a recursive function. The algorithm proceeds in three steps:

1. Search for a valid segmentation of the input term. This can be quite complex as the recurrence relation can be of arbitrary form, not just simple linear recursions such as linked-lists. In

Figure 4(b), bold lines cutting across tree edges segment the term such that the target node of each edge cut is an unfolding point. Three unfolding points are NULL nodes, which correspond to the base case of the recursion (modifications are made to Schmid's algorithm to determine when NULL nodes are not unfolding points). The unfolding point *h.child.sib.sib* is where the symbolic execution of the loop and hence the expansion of the term tree stop. An unfolding point like this is a single $*$ term with no children. We refer to them as the "un-expanded" nodes.

The basic algorithm for finding a valid segmentation is given in Figure 5. Formally, it searches for the set R of *recursion points* – places in the recurrence body where it invokes itself. The unfolding points in the term can be derived by repeated unrolling of the recurrence at its recursion points. In Figure 4(b), the recursion points coincide with the top two unfolding points, closest to the root of the tree. The other four are results of unrolling twice starting at the recursion point on the left. To ensure that the algorithm returns the minimal recurrence relation that explains the input term, the search for the next recursion point proceeds from left to right and from top to bottom, backtracking when R does not induce a valid segmentation. Validity of segmentation is checked by first computing a skeleton t_{skel} of the hypothetical recurrence body. t_{skel} is the minimal term tree that contains all paths in t leading to the recursion points, with the recursion points replaced by a special symbol 0. All other paths are replaced by fresh variables at the highest points. R induces a valid segmentation if for each unfolding point u derived from R, the relation $t_{skel} \leq u$ holds. $\leq$ is defined inductively as

- $0 \leq t'$ if t' contains NULL or un-expanded nodes,
- $x \leq t'$ if t' does not contain NULL or un-expanded nodes,
- $f(t_1, ..., t_n) \leq f(t'_1, ..., t'_n)$ if $t_i \leq t'_i$ for $i = 1..n$.

find_valid_segmentation(t)
$R = \{\}$ // The set of recursion points
$x =$ leftmost child of t
while $x \neq$ **null do**
  if *is_potential_recursion_point*(x, t) **then**
    R += x
    if *is_valid_segmentation*(R, t) **then**
      $x =$ *next_pos_right*(x)
      continue
    else
      R −= x
  $x =$ leftmost child of x, if any or *next_pos_right(x)*
return the set of unfolding points induced by R

is_potential_recursion_point(x, t)
return x is a NULL node $\vee$ (x has the same symbol as $t \wedge$
  x contains NULL or un-expanded nodes in its term tree)

is_valid_segmentation(R, t)
t_{skel} = the minimal pattern of t reaching all nodes in R
return $\forall$ unfolding point $u . t_{skel} \leq u$

next_pos_right(x)
if x has no parent **then**
  return null
if x has a right sibling y **then**
  return y
return *next_pos_right*(parent of x)

Figure 5. Algorithm to find valid segmentations

2. Compute the body of the recurrence, which is the maximal overlapping portion of all segments. This is done by *anti-unifying* ($\sqcap$) the segments:
   - $f(t_1, ..., t_n) \sqcap f'(t'_1, ..., t'_m) = \varphi(f(t_1, ..., t_n), f'(t'_1, ..., t'_m))$,
   - $f(t_1, ..., t_n) \sqcap f(t'_1, ..., t'_n) = f(t_1 \sqcap t'_1, ..., t_n \sqcap t'_n)$.

   φ is a one-to-one mapping between pairs of terms and variables which guarantees that identical sub-term pairs are replaced by the same variable throughout the whole term.
3. Find parameter substitutions. The sub-terms where the segments differ are instantiations of the parameters in the recurrence. Parameter substitutions are computed by identifying regularities in these terms. In our case, the parameters are precisely those terms translated from the names of heap locations. The access- paths, now encoded in the prefix form, provide the excellent opportunity for identifying interrelationships between the parameters. We define the notion of *positions* in a term tree t: *(i)* λ is the position of the root *(ii)* if the node at position u, denoted as $t|_u$, is a function, then its i-th child has position $u.i$. Within each segment s, let $\beta_{s,x_j,r}$ denote the sub-term that is the instantiation of parameter x_j at recursion point r. The substitution term for parameter x_j recursion point r is computed as *sub*(x_j, r, λ).

$$\textit{sub}(x_j, r, u) = \begin{cases} x_k & \text{if } \textit{is\_recurrent}(x_j, x_k, r, u) \\ f(\textit{sub}(x_j, r, u.1), ..., \textit{sub}(x_j, r, u.n)) & \text{if } \forall \text{ segment } s,\ \beta_{s,x_j,r}|_u = f(...) \text{ with } \textit{arity}(f) = n, \end{cases}$$

$$\textit{is\_recurrent}(x_j, x_k, r, u) = \forall \text{ successive segments } s \text{ and } s',\ \beta_{s',x_j,r}|_u = \beta_{s,x_k,r}|_\lambda.$$

sub is defined by structural induction on term trees. The leafs of the substitution term are parameter variables. They are determined through comparison of successive segment pairs in *is_recurrent* to see if a general pattern emerges. For an internal position u in the substitution term, it must hold that the corresponding parameter instantiations in all segments share the same function node at u.

For the term in Figure 4(b), the recurrence body is shown in Figure 4(c) on the top, and the final recurrence relation with parameter substitutions is shown at the bottom, which translates to the predicate *mcf_tree*(x_1, x_2, x_3) defined in Section 2.

3.2 What Recursion Synthesis Can and Cannot Do

The complete algorithm given in [8] handles the case where the recursion does not start at the root of the term tree, which happens when a recursive data structure is conjoined with some extra data. It can handle mutual recursions and nested recursions, which allows the analysis to support nested data structures, e.g. trees of linked-lists. It can also handle interdependencies between parameter instantiations and incomplete program traces. We believe the algorithm is powerful enough to decipher most recursive data types. However, our technique relies on the loop that constructs the data structure to reveal the data structure's recursive backbone. It will fail, for example, if the code reads a table that specifies the data structure or copies a data structure by keeping a map between pointers in the original and those in the duplicate.

4. Local Reasoning under Global Invariants

This section discusses two functions used by the symbolic execution rules, *unfold*$_\Theta(l{:}S, h)$ and *fold*$_\Theta(\Sigma)$. They concern reasoning about local changes to recursive data structures described by global shape invariants, yielding a general algorithm for unfolding and folding recursive predicates.

unfold$_\Theta$ takes a state S and a heap location h located either at the root of a recursive data structure or at the bottom sitting between the data structure and a truncation point, unrolls the predicate describing the data structure so that S contains explicit points-to assertions with h on the left hand side. It returns a set of states because case analysis is needed in the presence of truncation points.

Peeling the data structure from the top is conceptually easy, simply replace the recursive predicate with its inductive definition, substituting arguments passed to the predicate for parameters in the definition. Complication arises when the predicate contains truncation points. We do not know their exact positions relative to the root. They could be sitting right below the root, in which case they

alias with the newly exposed targets of h, or they could be farther way from h so that they become the truncation points in the sub data structures below h. Because spatial conjunction does not allow implicit aliasing, we have to enumerate all possible scenarios of relative positioning between h and the truncation points, constrained by the invariant that the sub data structures rooted at truncation points are mutually disjoint. Let n be the number of recursion points (Section 3.1.2) in the definition of the recursive predicate and m be the number of truncation points in the predicate. The total number of possibilities is exponential in $n \times m$. However, n is a small constant (1 for linked-lists, 2 for binary trees), m is also small because local updates only involve a few nodes and once done, the global invariant is restored and these truncation points are eliminated by $fold_{\ominus}$. Consider the heap: $mcf\_tree(h, null, null; \alpha) * mcf\_tree(\alpha, \beta_1, \beta_2)$. Unfolding h yields four heaps:

- $h.parent \rightarrow null * h.child \rightarrow \alpha * mcf\_tree(\alpha, h, null) *$ $h.sib\_prev \rightarrow null * h.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, null, h)$
- $h.parent \rightarrow null * h.child \rightarrow \beta_3 * mcf\_tree(\beta_3, h, null) *$ $h.sib\_prev \rightarrow null * h.sib \rightarrow \alpha * mcf\_tree(\alpha, null, h)$
- $h.parent \rightarrow null * h.child \rightarrow \beta_3 * mcf\_tree(\beta_3, h, null; \alpha) *$ $h.sib\_prev \rightarrow null * h.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, null, h)$
- $h.parent \rightarrow null * h.child \rightarrow \beta_3 * mcf\_tree(\beta_3, h, null) *$ $h.sib\_prev \rightarrow null * h.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, null, h; \alpha)$

Unrolling a recursive predicate from the bottom up makes h a new truncation point, causing some old truncation points to be removed to maintain mutual-disjointness of truncation points. Let T be the set of the original truncation points that point to h. Again, we do not know the exact access path from h to a $t \in T$, so case splitting is required as well. In this case the link from t to h also limits the possible places where t may alias with a node under h, according to the definition of the recursive predicate. Consider again the heap $mcf\_tree(h, null, null; \alpha) * mcf\_tree(\alpha, \beta_1, \beta_2)$. To unroll β_2, because the link $\alpha \rightarrow \beta_2$ is a *sib_prev* link, by definition of *mcf_tree*, α must be the target of the *sib* link originating from β_2. Hence after unrolling β_2, we have $mcf\_tree(h, null, null; \beta_2) * \beta_2.parent \rightarrow \beta_1 * \beta_2.child \rightarrow \beta_3 * mcf\_tree(\beta_3, \beta_2, null) * \beta_2.sib \rightarrow \alpha * \beta_2.sib\_prev \rightarrow \beta_4 * mcf\_tree(\alpha, \beta_1, \beta_2)$. If we were to consider α as the target of the *child* link of β_2, then α would be described as $mcf\_tree(\alpha, \beta_2, null)$, which is inconsistent with its description before the unrolling. Similar inconsistency also arises if we do not consider α as any target of β_2. Our algorithm checks each combination to rule out inconsistencies. In the case of unrolling β_1, there are two possibilities, either α is the *child* of β_1 or it is a truncation point in the *child* sub-tree of β_1.

Figure 6 contains the algorithm that determines all possible spatial relationships between an unfolded node h, associated with a recursive predicate A, and a set of truncation points T. Each possibility is represented by a function π that maps every t in T to either r or $\underline{r}$, where r is one of the recursion points in the definition of A. r means the t is located at the recursion point and $\underline{r}$ means that t is further below r. The algorithm assumes that neighboring recursive elements in the data structure are one pointer traversal away. This assumption can be removed by generalizing the algorithm. Details are omitted due to space constraint.

The case analysis performed in $unfold_{\ominus}$ closely mimics the way a programmer may reason informally about local updates – "If it is the case that x points y, then ...", but it does so exhaustively to ensure correctness. In comparison, folding a heap formula is straightforward because we do not need to worry about accidentally creating implicit aliasing, hence no case analysis is needed. It cleans up unused truncation points left behind by $unfold_{\ominus}$ in an attempt to incorporate cut-out pieces of the original data structure back into it, thereby restoring the global invariant. $fold_{\ominus}$ takes a

valid_possibilities $= \{\}$
for all π **do**
  if $\forall r, \exists t.\pi(t) = r \Rightarrow \nexists t'.t' \neq t \wedge \pi(t) = r$ or $\underline{r}$ **then**
    ok $=$ **true**
    for all $t \in T$ **do**
      if $\exists$ backward link $t.n \rightarrow h$ **then**
        let x_j be the parameter in A's definition s.t. $x_1.n \rightarrow x_j$
        let r be the recursion point s.t. $\pi(t) = r$ or $\underline{r}$
        if $\pi(t) = r$ **then**
          if the recursive call at r substitutes x_1 for x_j **then**
            continue
        else
          if the recursive call at r substitutes x_1 for some $x_k \wedge$ $\exists r'$.the recursive call at r' substitutes x_k for x_j **then**
            continue
        ok $=$ **false**
        break
    if *ok* **then**
      valid_possibilities += π

Figure 6. Algorithm for case analysis in $unfold_{\ominus}$

heap, looks for locations not pointed to by any live register, and tries to merge it into a neighboring data structure. Like $unfold_{\ominus}$, it also works from two directions, starting either with locations sitting directly atop a recursive data structure and working its way upwards, or with truncation points and working its way downwards. It achieves similar effect of the rewrite rules for list in [4, 5], $p \rightarrow k * list(k, q) \rightsquigarrow list(p, q)$ and $list(p, k) * k \rightarrow q \rightsquigarrow list(p, q)$. However, we handle arbitrary predicates by crawling the abstract heap such that each time, instead of a single heap cell, a whole chunk of heap fitting the definition of the recursive predicate (including backward links) are absorbed.

To illustrate unfolding and folding, we will again turn to 181.mcf. Figure 7 contains a code fragment which cuts a sub-tree from under its parent and connects it to a new parent. At entry `l0`, q and t are two truncation points in the *mcf_tree* whose root is R. The *parent* link of t points to p. The code fragment removes the sub-tree rooted at t from under p, moving the right sibling of t, if any, towards the left to be the new *child* of p. t is added as the *child* of q, shifting the old *child* of q, if any, toward the right. The heap formulae associated with each program label are listed in Table 3. For each label, parts of the heap formulae that are different from the previous label are underlined. The unfold and fold actions taken at each step are also listed. Formula $\Sigma_{1,2}$ corresponds to the case where the branch at `l1` is not taken. We omit subsequent formulae derived from it. They are similar to those listed here. The same is done for $\Sigma_{3,2}$. The registers that are live at the end of this code fragment are `t` and `q`. In the last step, we fold all other nodes back into the tree. The final heap $\Sigma_{6,2}$, where *t.sib* points to *null*, is subsumed by $\Sigma_{6,1}$ (based on the definition of $\sqsubseteq$ in Section 2.1).

```
l0:
    if (t->sib)
        t->sib->sib_prev = t->sib_prev;
l1:
    if (t->sib_prev)
        t->sib_prev->sib = t->sib;
    else
        p->child = t->sib;
l2:
    t->parent = q;
    t->sib = q->child;
l3:
    if (t->sib)
        t->sib->sib_prev = t;
l4:
    q->child = t;
    t->sib_prev = 0;
l5:
```

Figure 7. Local modification to a tree in 181.mcf

l0	$\mathbf{\Sigma_0}$: *mcf_tree*$(r, null, null; q, t) * mcf\_tree(q, \beta_1, \beta_2)*$ $t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ $t.sib\_prev \rightarrow \alpha_1 * t.sib \rightarrow \alpha_3 * mcf\_tree(\alpha_3, p, t)$	
l1	$\mathbf{\Sigma_{1,1}}$: Unfold α_3 *mcf_tree*$(r, null, null; q, t) * mcf\_tree(q, \beta_1, \beta_2)*$ $t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ $t.sib\_prev \rightarrow \alpha_1 * t.sib \rightarrow \alpha_3*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ $\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$	$\mathbf{\Sigma_{1,2}}$: *mcf_tree*$(r, null, null; q, t) * mcf\_tree(q, \beta_1, \beta_2)*$ $t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ $t.sib\_prev \rightarrow \alpha_1 *$ *t.sib → null*
l2	$\mathbf{\Sigma_{2,1}}$: Unfold α_1 *mcf_tree*$(r, null, null; q, \alpha_1) * mcf\_tree(q, \beta_1, \beta_2)*$ $\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$ $\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$ $t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ $t.sib\_prev \rightarrow \alpha_1 * t.sib \rightarrow \alpha_3*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ $\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$	$\mathbf{\Sigma_{2,2}}$: Unfold p *mcf_tree*$(r, null, null; q,$ *p*$) * mcf\_tree(q, \beta_1, \beta_2)*$ *p.parent* $\rightarrow \alpha_8 *$ *p.child* $\rightarrow \alpha_3*$ *p.sib_prev* $\rightarrow \alpha_9 *$ *p.sib* $\rightarrow \alpha_{10} *$ *mcf_tree*$(\alpha_{10}, \alpha_8, p)*$ $t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ *t.sib_prev* $\rightarrow$ *null* $* t.sib \rightarrow \alpha_3*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ *α₃.sib_prev* $\rightarrow$ *null* $* \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$
l3	$\mathbf{\Sigma_{3,1}}$: Unfold *q* *mcf_tree*$(r, null, null; q, \alpha_1)*$ *q.parent* $\rightarrow \beta_1 *$ *q.child* $\rightarrow \beta_3 *$ *mcf_tree*$(\beta_3, q, null)*$ *q.sib_prev* $\rightarrow \beta_2 *$ *q.sib* $\rightarrow \beta_4 *$ *mcf_tree*$(\beta_4, \beta_1, q)*$ $\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$ $\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$ *t.parent* $\rightarrow q *$ *t.child* $\rightarrow \alpha_2 *$ *mcf_tree*$(\alpha_2, t, null)*$ *t.sib_prev* $\rightarrow \alpha_1 *$ *t.sib* $\rightarrow \beta_3*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ $\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$	$\mathbf{\Sigma_{3,2}}$: Unfold q *mcf_tree*$(r, null, null; q, p)*$ *q.parent* $\rightarrow \beta_1 *$ *q.child* $\rightarrow \beta_3 *$ *mcf_tree*$(\beta_3, q, null)*$ *q.sib_prev* $\rightarrow \beta_2 *$ *q.sib* $\rightarrow \beta_4 *$ *mcf_tree*$(\beta_4, \beta_1, q)*$ $p.parent \rightarrow \alpha_8 * p.child \rightarrow \alpha_3*$ $p.sib\_prev \rightarrow \alpha_9 * p.sib \rightarrow \alpha_{10} * mcf\_tree(\alpha_{10}, \alpha_8, p)*$ *t.parent* $\rightarrow q *$ *t.child* $\rightarrow \alpha_2 *$ *mcf_tree*$(\alpha_2, t, null)*$ *t.sib_prev* $\rightarrow$ *null* $*$ *t.sib* $\rightarrow \beta_3*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ $\alpha_3.sib\_prev \rightarrow null * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$
l4	$\mathbf{\Sigma_{4,1}}$: Unfold β_3 *mcf_tree*$(r, null, null; q, \alpha_1)*$ $q.parent \rightarrow \beta_1 * q.child \rightarrow \beta_3*$ $q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*$ $\beta_3.parent \rightarrow q * \beta_3.child \rightarrow \beta_5 * mcf\_tree(\beta_5, \beta_3, null)*$ $\beta_3.sib\_prev \rightarrow t * \beta_3.sib \rightarrow \beta_6 * mcf\_tree(\beta_6, q, \beta_3)*$ $\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$ $\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$ $t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ $t.sib\_prev \rightarrow \alpha_1 * t.sib \rightarrow \beta_3*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ $\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$	$\mathbf{\Sigma_{4,2}}$: *mcf_tree*$(r, null, null; q, \alpha_1)*$ $q.parent \rightarrow \beta_1 *$ *q.child* $\rightarrow$ *null*$*$ $q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*$ $\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$ $\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$ $t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ *t.sib_prev* $\rightarrow \alpha_1 *$ *t.sib* $\rightarrow$ *null*$*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ $\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$
l5	$\mathbf{\Sigma_{5,1}}$: *mcf_tree*$(r, null, null; q, \alpha_1)*$ $q.parent \rightarrow \beta_1 * q.child \rightarrow t*$ $q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*$ $\beta_3.parent \rightarrow q * \beta_3.child \rightarrow \beta_5 * mcf\_tree(\beta_5, \beta_3, null)*$ $\beta_3.sib\_prev \rightarrow t * \beta_3.sib \rightarrow \beta_6 * mcf\_tree(\beta_6, q, \beta_3)*$ $\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$ $\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$ $t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ *t.sib_prev* $\rightarrow$ *null* $* t.sib \rightarrow \beta_3*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ $\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$	$\mathbf{\Sigma_{5,2}}$: *mcf_tree*$(r, null, null; q, \alpha_1)*$ $q.parent \rightarrow \beta_1 * q.child \rightarrow t*$ $q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*$ $\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$ $\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$ $t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ *t.sib_prev* $\rightarrow$ *null* $* t.sib \rightarrow$ *null*$*$ $\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$ $\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$
fold	$\mathbf{\Sigma_{6,1}}$: Fold $\alpha_1, \alpha_3, \beta_3$ *mcf_tree*$(r, null, null; q)*$ $q.parent \rightarrow \beta_1 * q.child \rightarrow t*$ $q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*$ $t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ $t.sib\_prev \rightarrow null * t.sib \rightarrow \beta_3 * mcf\_tree(\beta_3, q, t)$	$\mathbf{\Sigma_{6,2}}$: Fold α_1, α_3 *mcf_tree*$(r, null, null; q)*$ $q.parent \rightarrow \beta_1 * q.child \rightarrow t*$ $q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*$ $t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$ $t.sib\_prev \rightarrow null * t.sib \rightarrow null$

Table 3. Intermediate states of a tree update in 181.mcf

5. The Analysis

This section puts various pieces of the analysis together.

5.1 Code Pruning

Our interprocedural shape analysis includes a pre-pass in which a simple pointer analysis is performed to identify recursive data structures present in the program, and code that has no effect on the shape properties of these data structures is pruned away.

Because this shape analysis targets low-level code with no type information, a pointer analysis similar to Steensgaard's analysis [17] is used to roughly infer the high-level type of each pointer. This eliminates the need for shape analysis to track non-pointer data fields. These fields do not exhibit interesting recursive patterns and may confuse recursion synthesis. An inferred pointer type represents a set of runtime locations, e.g. the "next" field in all nodes of a linked-list. Our pointer analysis determines an inferred type for each load/store instruction, which over-approximates the set of locations the instruction accesses. Recursive types are identified as those associated with load instructions involved in traversing recursive data structures. These loads share the property that the destination register is used to compute the load address, a recurrence that is easily detected by computing strongly-connected components of the reaching-definition graph.

Code pruning is achieved by the following program slicing algorithm: It starts with an empty set of instructions and a set of tracked types initialized to the set of recursive types identified above. For each store to a tracked type, all instructions (including branches and possibly crossing procedure boundaries) that contribute to the computation of either the store address or the value to be stored are added. New pointer types that need to be tracked will be identified in the process, causing more instructions to be added. When the algorithm terminates, only instructions that may affect updates to recursive pointer fields are preserved. This step is essential for managing large benchmarks.

5.2 Interprocedural Analysis

As in [10] and [18], at each procedure entry, the analysis splits the state into a local heap and a frame, and sends the local heap, consisting of heap regions reachable from the actual parameters and the globals used in the procedure and all its callees, as the pre-state of the procedure (although any other splitting is sound). Cutpoints are preserved by telling the algorithm $fold_\theta$ not to fold them away. Our implementation adapts the tabulation algorithm in [18] which records procedure summaries for re-use under equivalent calling contexts. It is modified to perform inductive recursion synthesis, shown in Figure 8. In the interprocedural control flow graph, each call site is represented by a call node and a return node. Each procedure has an entry node and an exit node. The algorithm keeps a worklist of triples $\langle n, S_{entry}, S\rangle$, each saying that state S holds right before node n given that S_{entry} holds on entry of the procedure containing n. If n is a call node, the local heap of the callee is first extracted from S and the triple is registered as a calling context associated with the callee and this local heap. If no summary of the callee exists for this local heap, then it is pushed onto the worklist together with the callee's entry node. Upon popping the callee's exit node off the worklist, the updated local heap is propagated to all calling contexts registered with the callee to be re-combined into the frames. All other nodes are handled by *transform*$(n \to n', S)$, which updates state S according to the semantics of n. For memory efficiency, we do not keep around intermediate states, state duplication only occurs at split points of control flow and for recording procedure summaries and loop invariants.

5.2.1 Handling Recursive Procedures

Just like loops, recursive procedures are handled by symbolic execution along sample execution paths, then followed by inductive recursion synthesis. The control flow graph of a recursive procedure contains two kinds of loop back edges, one for recursive calls and one for recursive returns. Recursion synthesis is applied for both to infer the entry and exit invariants respectively. The choice of sample execution paths does not affect soundness as the inferred procedure entry/exit invariants are verified to see that they can derive themselves. We choose those paths that are good representative of the runtime behavior of the recursive procedures and are hence likely to yield valid invariants. On entry of a strongly-connected-component (SCC) in the call graph (representing mutually recursive procedures), the analysis follows an execution path that enters each procedure in the SCC at least twice. At each branch instruction reaching recursive calls, the analysis only propagates states to one branch target. The selection is performed by the subroutine *transform* in Figure 8, which returns NULL for the non-taken branch. If all procedures in the SCC have been visited at least twice, the target that does not lead to recursive calls is taken to ensure termination; otherwise, the other target is taken. If both targets lead to recursive calls, we favor the one leading to procedures that have not been visited twice yet. When checking the invariants of each procedure, all execution paths are taken into account.

```
worklist = {⟨entry_main, S0, S0⟩}
while worklist is not empty do
  remove ⟨n, S_entry, S⟩ from worklist
  if n is a call node then
    S_local = extract(S, callee)
    contexts(⟨callee, S_local⟩) += ⟨n, S_entry, S⟩
    if summary(⟨callee, S_local⟩) ≠ ∅ then
      for all S_exit ∈ summary(⟨callee, S_local⟩) do
        worklist += ⟨n_return, S_entry, combine(S_exit, S)⟩
    else
      worklist += ⟨entry_callee, S_local, S_local⟩
      if callee is recursive then
        latest_entry_state_callee = S_local
  else if n is an exit node then
    if callee is recursive then
      latest_exit_state_callee = S
      if caller is not in the callgraph SCC of callee then
        for all procedure p in callee's SCC do
          recursion_synthesis(latest_entry_state_callee)
          recursion_synthesis(latest_exit_state_callee)
        for all procedure p in callee's SCC do
          if !invariants_valid(p) then
            halt
    summary(⟨callee, S_entry⟩) += S
    for all ⟨n_call, S_e, S_c⟩ ∈ contexts(⟨callee, S_entry⟩) do
      worklist += ⟨n_return, S_e, combine(S, S_c)⟩
  else
    for all control flow edge n → n' do
      S' = transform(n → n', S)
      if n → n' is a back edge of loop l then
        if analysis converges or l has not iterated twice then
          worklist += ⟨n', S_entry, S'⟩
        else
          recursion_synthesis(S')
          if !invariants_valid(l) then
            halt
      else
        worklist += ⟨n', S_entry, S'⟩
```

Figure 8. The Interprocedural Algorithm

6. Experiment Results

This analysis has been implemented in our C compiler. Preliminary experiment results are reported in Table 4. Time was taken on a 3GHz P4 with 512KB cache and 2GB memory. In addition to 181.mcf which uses iterative algorithm to build and to traverse

its data structure, we tested on four Olden benchmarks which use recursive procedures. The 2nd column lists the recursive data structures used in the benchmarks. Our analysis is able to infer and maintain precise shape predicates that describe these data structures. For the most part, the shape analysis phase (last column) takes less time than the pre-pass (4th and 5th columns), demonstrating the effectiveness of code pruning.

Benchmark	Data Type	# Insts	Analysis Time (s)		
			Pointer	Slicing	Shape
181.mcf	*mcf_tree*	2158	0.59	0.22	0.55
treeadd	binary tree	162	0.09	0.02	0.05
bisort	binary tree	423	0.16	0.05	0.38
perimeter	quaternary tree w/ parent links	624	0.20	0.06	0.10
power	lists	1054	0.37	0.07	0.06

Table 4. Experiment results

7. Related Work

Recently there have been many interesting works in applying separation logic to program analysis, not just verification. Berdine et al. describe a form of symbolic execution that, for certain types of preconditions, generates post-conditions by updating the preconditions in-place [13]. It does not by itself yield a suitable abstract domain due to lack of guarantee for convergence. Two analyses of list- processing programs [4, 5] use rewrite rules tailored to the list predicate to reduce logic formulae and thereby arrive at fixed points. Both analyses are intraprocedural. An interprocedural analysis is given in [10], limited to pre-defined predicates as well. [19] studies pointer arithmetic in an abstract domain where each list node is a multiword.

Most similar to our work, Lee et al.'s grammar-based analysis [6] can also discover recursive predicates automatically. But since their grammar only allows one explicit argument, it cannot handle data structures with multiple backward links such as the *mcf_tree* and it cannot handle multiple pointers to the interior of a data structure. We support arbitrary number of parameters and can handle dags in some cases while they cannot.

In [20], inductive learning is also used to find instrumentation predicates. Their technique, based on successive refinement given positive and negative examples, is different from recursion synthesis. Although in principle their predicates can describe complex data structures, the inference of such recursive predicates is not evaluated in their experimentation.

8. Conclusion and Future Work

This paper presented an interprocedural shape analysis based on separation logic. With two powerful techniques, inductive recursion synthesis and generic recursion unrolling/rolling based on truncation points, the analysis is able to take separation logic based program analysis beyond simplistic data structures. For future work, we would like more results on the conditions under which the recursion synthesis algorithm would fail. Though we do not expect it to fail often in practice, when it does, a more elegant recovery scheme than halting is desirable. Potential solutions include a special predicate that says all pointers to a particular data structure may alias and prompting the user for further information.

References

[1] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 296–310, June 1990.

[2] N. D. Jones and S. S. Muchnick, "Flow analysis and optimization of Lisp-like structures," in *Program Flow Analysis: Theory and Applications* (S. S. Muchnick and N. D. Jones, eds.), pp. 102–131, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[3] J. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, July 2002.

[4] D. Distefano, P. W. O'Hearn, and H. Yang, "A local shape analysis based on separation logic," in *Lecture Notes in Computer Science*, vol. 3920, pp. 287–302, Springer-Verlag, 2006.

[5] S. Magill, A. Nanevski, E. Clarke, and P. Lee, "Inferring invariants in separation logic for imperative list-processing programs," in *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, January 2006.

[6] O. Lee, H. Yang, and K. Yi, "Automatic verification of pointer programs using grammar-based shape analysis," in *Prceedings of the 2005 European Symposium on Programming (ESOP)*, 2005.

[7] P. Summers, "A methodology for Lisp program construction from examples," *Journal ACM*, vol. 24(1), pp. 162–175, 1977.

[8] U. Schmid, *Inductive synthesis of functional programs*. Berlin, Germany: Springer-Verlag, 2003.

[9] A. M⊘ller and Schwartzbach, "The pointer assertion logic engine," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pp. 221–231, 2001.

[10] A. Gotsman, J. Berdine, and B. Cook, "Interprocedural shape analysis with separated heap abstractions," in *Proceedings of the 13th International Static Analysis Symposium (SAS)*, August 2006.

[11] N. Rinetzky, J., Bauer, T. Reps, M. Sagiv, and R. Wilhelm, "A semantics for procedure local heaps and its abstractions," in *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, pp. 296–309, January 2005.

[12] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[13] J. Berdine, C. Calcagno, and P. W. O'Hearn, "Symbolic execution with separation logic," in *Lecture Notes in Computer Science*, vol. 3780, pp. 52–68, Springer-Verlag, 2005.

[14] P. W. O'Hearn, H. Yang, and J. Reynolds, "Separation and information hiding," in *Proceedings of the 31st ACM symposium on Principles of Programming Languages*, pp. 268–280, January 2004.

[15] J. Berdine, C. Calcagno, and P. W. O'Hearn, "A decidable fragment of separation logic," in *Lecture Notes in Computer Science*, vol. 3328, pp. 97–109, Springer-Verlag, 2004.

[16] G. Tan and A. W. Appel, "A compositional logic for control flow," in *Lecture Notes in Computer Science*, vol. 3855, pp. 80–94, Springer-Verlag, 2006.

[17] B. Steensgaard, "Points-to analysis by type inference in programs with structures and unions," in *Lecture Notes in Computer Science, 1060*, pp. 136–150, Springer-Verlag, 1996.

[18] N. Rinetzky, M. Sagiv, and E. Yahav, "Interprocedural shape analysis for cutpoint-free programs," Tech. Rep. 26, Tel Aviv University, November 2004.

[19] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Beyong reachability: Shape abstraction in the presence of pointer arithmeetic," in *Lecture Notes in Computer Science*, vol. 4134, pp. 182–203, Springer-Verlag, 2006.

[20] A. Loginov, T. Reps, and M. Sagiv, "Abstraction refinement via inductive learning," in *Proceedings of the 17th International Conference on Computer Aided Verification*, pp. 519–533, 2005.