SHAPE ANALYSIS WITH INDUCTIVE RECURSION SYNTHESIS

BOLEI GUO

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

ADVISOR: DAVID I. AUGUST

JUNE 2008

Abstract

For program optimization and verification purposes, shape analysis can be used to statically determine structural properties of the runtime heap. One promising formalism for describing heap is separation logic, with recursively defined predicates that allow for concise yet precise summarization of linked data structures. A major challenge in this approach is the derivation of the predicates directly from the program being analyzed. As a result, current uses of separation logic rely heavily on predefined predicates, limiting the class of programs analyzable to those that manipulate only predefined data types. This thesis addresses this problem by proposing a general algorithm based on *inductive program synthesis* that automatically infers recursive predicates in a canonical form. This technique yields a separation logic based shape analysis that can be effective on a much wider range of programs.

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 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.

We present a fully implemented interprocedural analysis algorithm that handles recursive procedures. A combination of pointer analysis and program slicing is used to deal with the scalability issue typically faced by shape analyses.

Finally, we present a data dependence test for recursive data structures that takes advantage of the results of our shape analysis.

Acknowledgments

This thesis is a direct result of the vision of my advisor David I. August. He has always pushed for more functional and scalable memory analyses on low-level code. I thank him for believing in me and for his support and guidance throughout my years in graduate school.

This work was also made possible with the help of the entire Liberty Research Group. Spyridon Triantafyllis, Matthew J. Bridges, Guilherme Ottoni, Easwaran Raman and others are responsible for building the Velocity compiler, the framework that allows me to implement and experiment with my ideas. The insights and suggestions of many members of the group have improved my research. They have also given me tremendous help during paper deadlines.

I thank my committee members, Andrew Appel, Michael Hind, Brian Kernighan and David Walker, for providing valuable feedback on how to improve this thesis. I am also grateful to Limin Jia and Frances Perry for many interesting discussions that help to deepen my understanding of separation logic and shape analysis.

This research was generously supported by the Intel Foundation Ph.D. Fellowship and NSF. I would also like to thank Intel for offering me a valuable summer internship with their Programming Systems Lab. I had a great time working with Youfeng Wu, Cheng Wang and Bixia Zheng among others.

Finally, special thanks go to my parents, Duozhi Lei and Jiajie Guo, for their sacrifices, unconditional love and support. I always came back reinvigorated from those weekend trips to home. I could not have done it without them.

Contents

List of Figures

Chapter 1

Introduction

Shape analysis is a type of memory analysis that discovers deep properties of the runtime heap. It offers a higher level of precision than do the traditional pointer/alias analyses and can enable aggressive code optimizations, program verification, and program understanding tools.

In the presence of dynamic memory allocation, the size of a heap-allocated data structure can not be bounded at compile time. To deliver on the high degree of precision, shape analysis thus requires a sophisticated heap abstraction mechanism that is detailed enough to entirely capture the heap layout and yet compact enough to scale to large programs. Such a mechanism should also make it easy to reason about local updates to data structures once they have been constructed.

Separation logic [23] is a heap formalism that uses logic formulas to describe the runtime heap. It provides recursively defined predicates to summarize the global invariants of linked data structures and has locality built into the logic to facilitate modular reasoning about the heap. Although powerful, the difficulty in using separation logic in shape analysis is the lack of a general way of achieving fixed-point convergence of the

analysis. Specifically, it is difficult to derive the recursive predicates that would serve as the converged abstract respresentation of infinitely many runtime heaps. Existing research on deriving such predicates only works for a limited set of predicates [7, 16, 14].

As a result, separation logic is mostly useful only for program verification, which relies on user-supplied program specifications, typically in the form of loop invariants and procedure pre and post conditions. In order to make separation logic applicable to fully automatic program analysis with no human intervention at all, we attack the problem of discovering recursive predicates by developing a novel approach to loop invariant inference that avoids restricting the analysis to predefined predicates. We also design a new type of predicates that accommodates local updates to data structures where the global recursive shape invariants are typically broken temporarily and reestablished later. The thesis of this dissertation is that the combination of the abovementioned two techniques makes it possible to use separation logic to build a precise and scalable shape analysis that is effective on a wide range of programs.

1.1 Shape Analysis and Separation Logic

1.1.1 Overview of Separation Logic

Separation logic [23] is designed for formal reasoning about low-level imperative programs that manipulate pointers. It extends the standard predicate calculus ($\wedge \vee \neg \Rightarrow \forall \exists$) with four new types of assertions that describe the heap.

emp	empty heap
$exp_1 \rightarrow exp_2$	single-cell heap
$assertion_1 * assertion_2$	spatial conjuction
$assertion_l -\!\!* asseriont_2$	spatial implication

emp represents an empty heap. A heap containing a single location h whose content is v is written as $h \rightarrow v$. The central feature of separation logic is the spatial conjuction operator $*$, which connects together two disjoint pieces of heap. $A * B$ is true in a heap if it can be partitioned into two disjoint pieces such that A holds over one and B holds over the other. This definition implies that if we have $x \rightarrow \_ * y \rightarrow \_$, then it must be that $x \neq y$, that is, there is no possibility that x and y may alias. What is remarkable about this is that it eliminates the need to check the global state for aliasing conditions and to subsequently update the global state whenever the program writes to a single heap location. Thus if a memory store operation is performed on x, the analysis needs only to deal with the part of logic formula that directly references x knowing that the rest of the state will not be changed. The dependence on global state has been one of the main reasons that leads to high complexity of shape analysis, while spatial conjunction promotes modularity in the analysis allowing it to scale to program size. Finally, spatial implication A-*B says that the current heap is conjoined with another heap satisfying A, then the combined heap satisfies B. Spatial implication is useful for backward analysis where one is given the program state after the execution of a piece of code and needs to infer the prestate. Here, we use it in forward style reasoning for the purpose of describing a recursive data structure minus a portion where the global shape invariant might be temporarily broken (section 2.1).

Additionally, to model healp-allocated data structures whose size cannot be bounded statically, separation logic also includes inductively defined predicates. For example, an

acyclic linked list is captured by the recursive predicate $list(x) \doteq (x = null \wedge \mathbf{emp}) \vee (x \rightarrow \alpha * list(\alpha))$. It says that a list can either be an empty heap or a heap composed of a single cell and a list with the cell pointing to the head of that list. Compared to other approaches to providing finite heap representations, such as *summary node* [4] which groups concrete elements of a data structure into a finite number of abstract heap nodes, and *k-limiting* [12] which only distinguishes elements of a linked data structure up to depth k, recursive predicates are precise while being concise. They do not introduce approximation of the memory states, which leads to loss of information on the exact shapes of data structures.

For specification of program properties, separation logic also extends Hoare logic [11], which is a system of logic rules that specifies how the program state can be changed by the execution of code. It uses a form of judgement $\{P\}C\{Q\}$, named the Hoare Triple, where P and Q are logic assertions and C is a piece of code that can either be a single instruction or a composition of instructions. The Hoare Triple says that if the precondition P holds of the program state before the execution of C and if C terminates, then the postcondition Q will hold after the execution of C. In addition to the original inference rules for the basic constructs of low-level imperative language (assignment and conditional, etc), separation logic adds rules that deal with memory lookup, mutation, allocation and deallocation (all rules are described in detail in section 2.2). The important Frame rule is also introduced, which directly expresses the idea of local reasoning about heap.

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}} \ \textbf{FRAME}$$

C does not modify any free variable in R

The idea is that if the footprint of C is enclosed in the heap described by P, then the execution of C does not affect any heap that is disjoint from P. This proves to be very helpful in interprocedural analysis where C is a procedure and usin the Frame rule we can divide the precondition into the part that is relevant to C, namely P, and a *frame* R, transform P into Q according to the semantics of C, and combine it back with R to form the postcondition.

1.1.2 Research Challenges

As an important step torward separation logic based program analysis, Berdine et al. propose a form of symbolic execution that interprets separation logic formulas as symbolic heaps and updates the precondition in-place to model the actual updates of the heap [1]. However, it can only handle loop- free programs because with loops here could be infinitely many symbolic heaps and the analysis will fail to converge. The key problem here is the lack of ways to infer recursive predicates, which are capable of representing infinitely many symbolic heaps accurately. Current uses of separation logic typically have a handful of predefined predicates hardwired into the logic engine and relies on user-supplied specifications that a predicate holds at a certain program point. In the case of linked lists, clever logic rules can be designed to recognize certain patterns in the logic formulas and to rewrite them to synthesize the list predicate. Two analyses of list-processing programs have been proposed [7, 16], 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 [14] that automatically discovers grammars which can be translated to recursive predicates. However, their grammars can only have up to two parameters, thus limiting the class of data structures describable.

We propose a shape analysis that performs *inductive recursion synthesis* to infer recursive predicates in a canonical form, effectively reverse-engineering the data types used 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 [30, 27]. 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.

Not only is it hard to discover recursive predicates from the program, it is also hard to handle them in symbolic execution once discovered. While recursive predicates express global properties that hold over entire data structures, most programs perform many local alterations (insertions, deletions, rotations, etc.) to the data structures and reestablish 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 is designed to facilitate this kind of local reasoning through explicit expression of structure separation and aliasing. However, for data structures with complex internal sharing, it is often difficult to isolate a substructure separate from the rest of the data structure. In [7, 16], logic rules tailored to the list predicate are designed to unfold a list in order to expose a list element and to 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 parts from a data structure. Generic algorithms based on truncation points are then designed to unfold and fold arbitrary recursive data structures.

1.2 Research Objectives and Contributions

The goal of this research is to handle real applications like those from the SPEC benchmarks. Such programs present three challenges. First, their data structures are often 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 1.1, there is a great degree of internal sharing which makes both inferring its shape and reasoning about its shape challenging. We demonstrate that our analysis discovers the precise shape invariant of the *mcf_tree* and maintains this invariant through local updates to the tree. Second, many applications perform their own memory management by preallocating a large array and taking chunks from it when needed. To model this correctly, the analysis needs to track aliasing that arises from pointer arithmetic in addition to access paths. Finally, the analysis should also scale to large programs. To this end, our algorithm performs a prepass 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, a key to shape analysis. This also reduces noise introduced by nonpointer fields that may confuse the inductive recursion synthesis algorithm.

The application of inductive recursion synthesis to shape analysis was first proposed in [10]. The presentation in this dissertation includes a more detailed description of the

recursion detection algorithm and the interprocedural analysis algorithm, including an example for handling recursive procedures. Additionally, to allow code optimizations to extract information from the result of our shape analysis, we also design a loop-carried dependence test targeting loops that traverse linked data structures instead of arrays. This test identifies pointer-chasing loops and computes a dependence distantce between two iterations that may reference the same memory locations. Such result may enable automatic thread extraction.

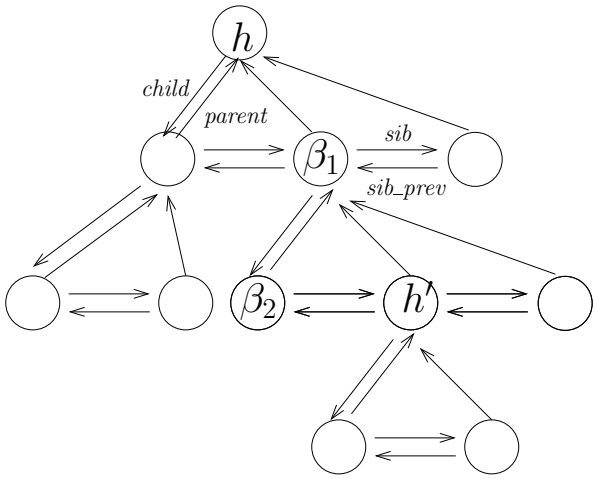


Figure 1.1: A specimen of the tree used in 181.mcf

In summary, the contributions of this thesis include:

- We develop a new approach to loop invariant inference that allows automatic discovery of recursive shape predicates (Chapter 3). This technique can handle any data type with a tree-like backbone and some other pointer fields that point to the backbone, possibly producing dags and cycles. This gives our analysis the same descriptive power as the Pointer Assertion Logic [17]. However, in their framework shape invariants are already given by nontraditional 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, data types, procedure summaries and loop invariants.

- Inductive recursion synthesis is also used to converge over recursive procedures (Chapter 5). Like the interprocedural shape analysis by Gotsman et al. [9], at each procedure entry, the analysis extracts the region of heap accessed by the procedure, called the *local heap*, and upon return, reincorporates the updated local heap using the Frame rule of separation logic. *Cutpoints* [24], the nodes that separate the local heap from the frame, need to be preserved so that the callee's effects on heap can be properly propagated to the caller. In the presence of recursive procedures, the number of cutpoints can be infinite. Gotsman et al. [9] bound the number of cutpoints at the cost of potential precision loss. In our case, recursion synthesis allows cutpoints to be described inductively in the entry/exit invariants of recursive procedures, hence there is no need to bound them.

- We describe a general algorithm based on truncation points for unrolling and rolling back arbitrary recursive data structures, even those with internal sharing (Chapter 4). We demonstrate that truncation points allow the modeling of cut and paste of subtrees in 181.mcf.

- We incorporate various techniques to handle real applications with special memory allocation semantics (Chapter 2) and large sizes (Chaper 6).

- Same as Ghiya et al. [8], our loop-dependence test (Chapter 7) begins by identifying the pointer used to traverse the recursive data structure in a given loop. However, by taking full advantage of the accuracy of our shape analysis, we provide more sophiscated algorithms to verify that the pointer visits a distinct node in each iteration and to prove that there is no other loop-carried dependences in addition to

that which is induced by pointer-chasing. If additional loop-carried dependences do exist, our dependence test also computes the dependence distance while the test described in [8] does not.

The abstract respresentatino of state is described in Chapter 2 Implementation and experimental results are reported in Chapter 6.2. Related work is further discussed in Chapter 8. Finally, Chapter 9 summarizes this thesis and discusses several directions for future work.

Chapter 2

Abstract Semantics

The target language of this shape analysis is an assembly-level intermediate language used in the Velocity compiler. The syntax of this language is shown in Figure 2.1. Globals are names of heap locations allocated for global variables. The set of expressions includes additions so that the analysis can handle pointer arithmetic. In situations where we cannot precisely compute the result of a pointer arithmetic operation, which is typically the case when array accesses are involved, indistinguishable array elements will be collapsed into a single one.

The rest of this chapter describes the abstract representation of states and an abstract operational semantics 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 exe-

Labels	l	$\in$	*Label*
Globals	g	$\in$	*Global*
Registers	r	$\in$	*Reg*
Exprs	e	$::=$	$\mathbf{null} \mid g \mid r \mid r+n \mid r_1+r_2$
Insts	s	$::=$	$r=e \mid [r_1]=r_2 \mid r_1=[r_2] \mid r=\mathbf{malloc}() \mid \mathbf{free}(r) \mid r=f(\vec{x}) \mid$ $\mathbf{goto}\ l \mid \mathbf{if}\ c\ \mathbf{goto}\ l$
Branch Conds	c	$::=$	$r_1=r_2 \mid r_1 \neq r_2$

Figure 2.1: Target Language

cution 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. Figure 2.2 gives the definition of the state.

Unlike the "symbolic heaps" defined by Berdine et al. [2], 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 prestate in a suitable form by going through the alias pairs, this facilitates the inductive recursion synthesis algorithm. As will be explained in Chapter 3, the access-path-like heap names encode important patterns of spatial relationships between heap locations, which can be recognized and then generalized via inductive reasoning. The heap names can be simply thought of as logic variables with long names. A heap name may also appear in curly braces, which indicate that it represents a set of indistinguishable con-

Vars	α	$\in$	Var
Recursion vars	A	$\in$	Rec
Heap names	h	$::=$	$g \mid \alpha \mid h.n \mid \{h\}$
Symbolic vals	v	$::=$	$null \mid h+n$
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	$::=$	$\mathbf{emp} \mid H \mid \Sigma * \Sigma$
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	$::=$	$\oslash \mid \Gamma, \langle f, \Pi_{entry} \mid \Sigma_{entry} \mid \Phi_{entry}, \Pi_{exit} \mid \Sigma_{exit} \mid \Phi_{exit} \rangle$

Figure 2.2: Abstract States

crete addresses usually belonging to an array. This is necessary because strong updates are performed on an abstract heap location only if it unequivocally represents a single concrete location.

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 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:

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

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

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 $(*_{i=1..m}\exists\beta_{i,1},...,\beta_{i,n-1}.A(h'_i,\beta_{i,1},...,\beta_{i,n-1}))-\!\!*A(h_1,...,h_n)$ (the iterated spatial conjunction operator is defined in [23]). 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 subgraphs rooted at $h'_1,...,h'_m$ cut out from it. The definition specifies that the subgraphs are mutually disjoint, i.e. no truncation point can be in the subgraph of another truncation point. This invariant is crucial for unrolling predicates as it constrains the number of possible outcomes (details are in Chapter 4). In Figure 1.1, suppose that at some program point, there is a pointer to an interior node h' of the tree, then the heap is described as $mcf\_tree(h,null,null;h') * mcf\_tree(h',\beta_1,\beta_2)$. By the definition of *mcf_tree*, we know

that the dangling points β_1 and β_2 of the heap $mcf\_tree(h', \beta_1, \beta_2)$ 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 $list(x,y) \doteq (x = y \wedge \mathbf{emp}) \vee (x \rightarrow \alpha * list(\alpha, y))$ defined by Berdine et al. [1]. 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 subtrees.

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

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

$$[\![r]\!]_{\Pi,\Phi} = \begin{cases} h' & \text{if } \Pi(r) = h + n \text{ and } \Phi \text{ records the alias } h + n = h' \\ \Pi(r) & \text{otherwise} \end{cases}$$

$$[\![r + n]\!]_{\Pi,\Phi} = \begin{cases} h' & \text{if } \Pi(r) + n = h + m \text{ and } \Phi \text{ records the alias } h + m = h' \\ \Pi(r) + n & \text{otherwise} \end{cases}$$

$$[\![r_1 + r_2]\!]_{\Pi,\Phi} = \begin{cases} \{h\} & \text{if } [\![r_1]\!]_{\Pi,\Phi} = h \\ \{h\} & \text{if } [\![r_1]\!]_{\Pi,\Phi} = \{h\} \\ \{h\} & \text{if } [\![r_2]\!]_{\Pi,\Phi} = h \\ \{h\} & \text{if } [\![r_2]\!]_{\Pi,\Phi} = \{h\} \end{cases}$$

In the case of $r_1 + r_2$, which typically results from translation of array accesses into low-level code, one of the two registers contains the base pointer while the other contains the offset. A simple pointer analysis phase (Chapter 6) that precedes the shape analysis can identify which register contains the pointer value, hence the function $[\![]\!]_{\Pi,\Phi}$ will return the correct result.

The partial order $\sqsubseteq$ over the set of abstract states can be 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 atomic H in Σ_1, $f^\dagger(H)$ is in Σ_2, $f^\dagger$ replaces each h appearing in H with $f(h)$, and *(ii)* for each atomic P in Φ_1, $f^\ddagger(P)$ is in Φ_2, $f^\ddagger$ replacess each h in P with $f(h)$. Intuitively, this definition says that the heap described by $\Pi_1 \mid \Sigma_1 \mid \Phi_1$ if a subheap of that which is described by $\Pi_2 \mid \Sigma_2 \mid \Phi_2$. Obviously, as the size of the heap is unbounded in the presence of dynamic memory allocation, there could be infinitely increasing 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 [31]. Some modifications are made to accommodate the fact that our logic rules are written for forward 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 default 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 2.1 and 2.2 list 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.

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

The **MUTATE** rule performs strong update if h_1 does not appear in braces. In the case of aliasing due to array elements collapsed into a single heap element, weak updates will have to be performed, in which case the analysis degenerates into Steensgaard's pointer analysis [29]. **MUTATE** invokes an important subroutine *rearrange_names*, shown in Figure 2.3, 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 $h_1.n$, that is, value v is to be stored at location $[h_1 + n]$, which previously contained the value h_2. 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 $h_1.n$

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

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

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

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

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

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

$$\frac{}{\{l : S\}, \{l\ (\textbf{goto}\ l_1)\ l'\} \vdash \{l_1 : S\}}\ \textbf{JUMP}$$

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

$$filter(r_1 = r_2)(l : \Pi \mid \Sigma \mid \Phi) =$$
$$\begin{cases} \{l : \Pi \mid \Sigma \mid \Phi \wedge [\![r_1]\!]_{\Pi,\Phi} = [\![r_2]\!]_{\Pi,\Phi}\} & \text{if } \Phi \nvdash [\![r_1]\!]_{\Pi,\Phi} \neq [\![r_2]\!]_{\Pi,\Phi} \\ \oslash & \text{otherwise} \end{cases}$$

$$filter(r_1 \neq r_2)(l : \Pi \mid \Sigma \mid \Phi) =$$
$$\begin{cases} \{l : \Pi \mid \Sigma \mid \Phi \wedge [\![r_1]\!]_{\Pi,\Phi} \neq [\![r_2]\!]_{\Pi,\Phi}\} & \text{if } \Phi \nvdash [\![r_1]\!]_{\Pi,\Phi} = [\![r_2]\!]_{\Pi,\Phi} \\ \oslash & \text{otherwise} \end{cases}$$

Table 2.1: Abstract Operational Semantics

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

$$\frac{\Psi_1,F_1\vdash\Psi_1'\quad\Psi_2,F_2\vdash\Psi_2'}{\Psi_1\cup\Psi_2,F_1\cup F_2\vdash\Psi_1'\cup\Psi_2'}\ \textbf{COMBINE}$$

$$\frac{\Psi\cup\{l:S\},F\vdash\Psi'\cup\{l:S\}}{\Psi,F\vdash\Psi'\cup\{l:S\}}\ \textbf{DISCHARGE}$$

$$\frac{\Psi,F\vdash\Psi_2'\quad\Psi_2'\Rightarrow\Psi_1'}{\Psi,F\vdash\Psi_1'}\ \textbf{WEAKEN}$$

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

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

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

Table 2.2: Abstract Operational Semantics (Continued)

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. In cases where the heuristic fails, the inductive recursion synthesis will not be able capture the correct patterns of recursive data structures. Our analysis compensates for this to some degree by performing a prepass consisting of simple pointer analysis and code pruning (Section 6.1) to remove code that is not relevant to the construction and manipulation of recursive data structures. As a result, assignments of cross pointers will be removed and will not confuse the shape analysis.

```
rearrange_names(h1, n, h2, v)
if v = a then
  if h2 = h1.n then
    replace h2 everywhere with a fresh variable
  replace v everywhere with h1.n
  return h1.n
else
  if v = h + n then
    if h2 = h1.n then
      replace h2 everywhere with a fresh variable
    record alias ⟨h + n, h1.n⟩
    return h1.n
  return v
```

Figure 2.3: Algorithm of *rearrange_names*

The rule **PROC_CALL** is the same as the one given by Gotsman et al. [9]. 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, $(f, \Pi_{entry} \mid \Sigma_{entry} \mid \Phi_{entry}, \Pi_{exit} \mid \Sigma_{exit} \mid \Phi_{exit})$, 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 **UNFOLD** is invoked when the analysis encounters a load or store instruction that accesses a recursive data structure. In order to expose specific points-to facts, the recursive predicate that globally describes the data structure needs to be unrolled. This is achieved by the algorithm $unfold_{\Theta}$, given in Chapter 4.

There are three rules for composing primitive instructions into program fragments. **COMBINE** merges states. **DISCHARGE** gets rid of redundant intermediate states as some labels are associated with both an entry point and an exit point when instructions are connected by control flows. **WEAKEN** is used both for further removing intermediate states and for converging over loops. It is guided by the binary relation $\Rightarrow$ between sets of labeled states. $\Psi_1 \Rightarrow \Psi_2$ if either Ψ_1 contains Ψ_2 or at least one normalization step can be performed on Ψ_1 to yield Ψ_2. It is the **normalize** rule that is responsible for guaranteeing termination of the analysis. There are two kinds of normalization operations. From the current state, **synthesis** infers a recursive description that is guaranteed to be more general (Chapter 3). **fold** reduces the size of the state by folding surrounding heap nodes into a recursive predicate (Chapter 4).

We use a `while` loop to illustrate how the composition rules work together in forward-reasoning style. In Figure 2.4, after the **COMBINE** step, the two instructions in Figure 2.4(a) are merged into a program fragment with entry labels $\{l, l_1\}$ and exit labels $\{l, l_1, l'\}$. As control flows are merged, label l identifies both an entry and an exit, so is label l_1. In the case of l_1, the states at the corresponding entry and exit are the same, so using the **DISCHARGE** rule the internal entry point can be eliminated, as shown in Figure 2.4(c). Finally, the **superset** and **WEAKEN** rules together remove l_1 as an exit point because a subset of labeled states are weaker than the superset. Now, the `while`

loop fragment in Figure 2.4(d) has one entry l carrying the initial state at iteration 0 and two exits, an outbound one l' carrying the state upon loop termination and an inbound one l_1 carrying the loop invariant. In the more complex case where the loop invariant has to be inferred, the **normalize** rule will be used.

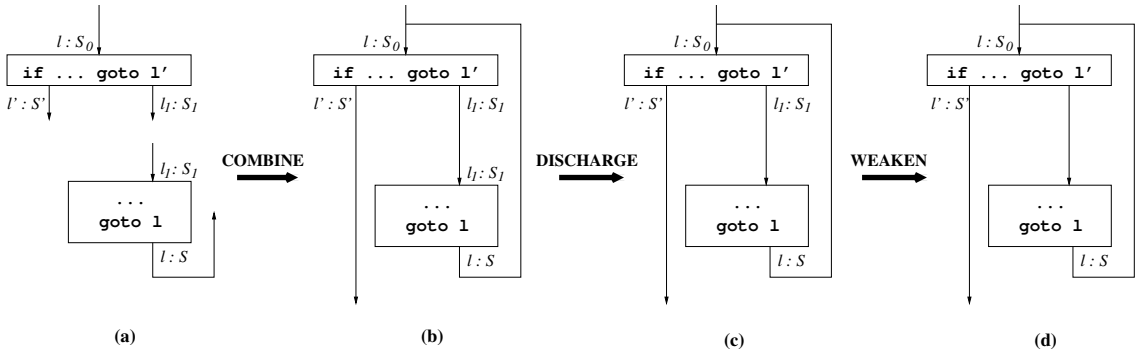


Figure 2.4: Instruction composition for `while` loop

Chapter 3

Inferring Recursive Predicates

This chapter 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, the state at the end of the path matches the loop 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 Inductive Recursion Synthesis

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 [30] 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 Chapter 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 3.1. A term can be viewed as a tree where each symbol is a node.

$$
\begin{array}{llll}
\text{Terms} & t ::= & x & \text{variables} \\
 & \mid & c & \text{constants} \\
 & \mid & f(t_1, .., .t_n) & \text{functions}
\end{array}
$$

Figure 3.1: 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 subtrees 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 subtree of the left hand side. All others are translated into name terms by the rewrite function $[\cdot]$, cutting the points-to link in a sense.

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

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

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

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 3.2 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 3.3(a) 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 $\stackrel{n}{\rightarrow}$ term whose left subterm is the name of the source location and the right subterm 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 3.3(a) 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. 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

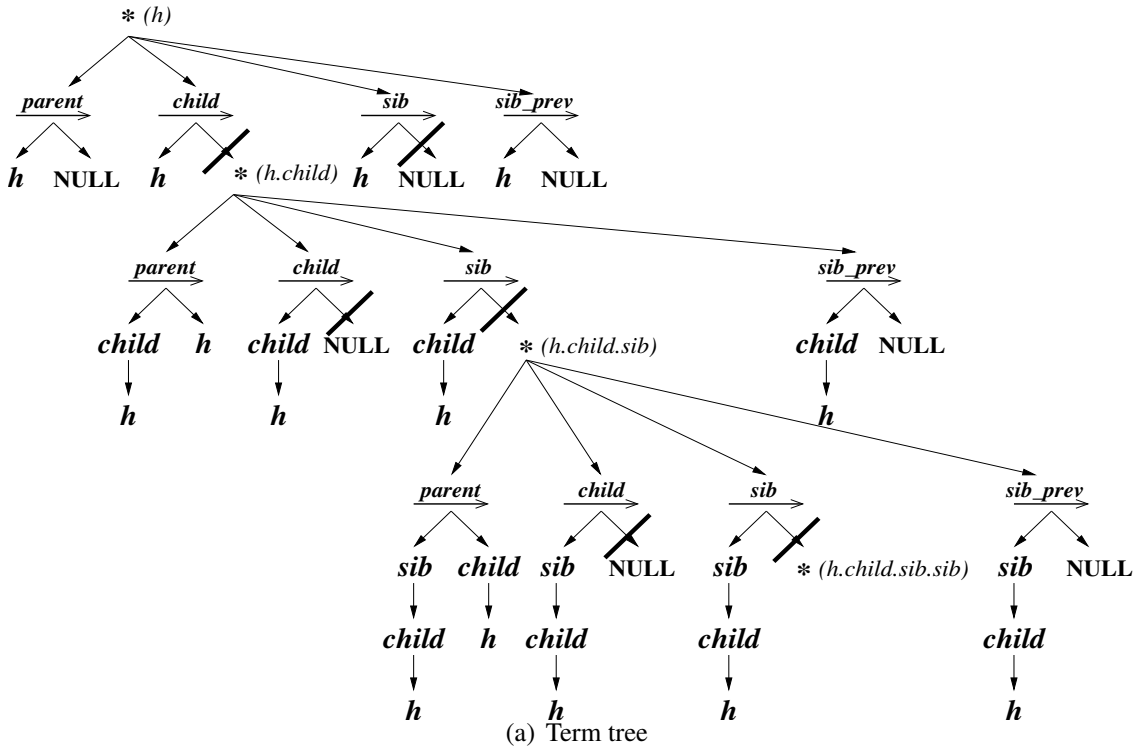
For convergence over loops, the recurrence detection algorithm is applied to each top-level term (a loop may touch multiple data structures). The algorithm we build on is

```
    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++;
    }
```

Figure 3.2: A Loop in 181.mcf that Builds its Tree

by Schmid [27]. 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 [30], 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 3.3(a), 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



(a) Term tree

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

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

(b) Recurrence

Figure 3.3: Inductive Recursion Synthesis for the Loop in Figure 3.2

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 "unexpanded" nodes.

The basic algorithm for finding a valid segmentation is given in Figure 3.4. 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 3.3(a), 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. Term tree nodes that may potentially be recursion points are either NULL nodes or function nodes that have the same symbol as the root of the tree (which is always the * operator in this analysis) and contain NULL nodes and unexpanded nodes in its subtree. The subroutine *is_potential_recursion_point* tests each tree node using this criterion. The reason for only considering * operators and NULL symbols as potential recursion points is that they are respectively where a term tree adds and stops adding new expansions to itself.

The 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

```
find_valid_segmentation(t)
R = {} // The set of recursion points
x = leftmost child of t
while x ≠ 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 ∨ (x has the same symbol as t ∧
        x contains NULL or unexpanded nodes in its term tree)

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 3.4: Algorithm for Finding a Valid Segmentation

symbol 0. All other paths are replaced by fresh variables at the highest points. For the term tree in Figure 3.3(a), let R consist of the top two unfolding points, then the corresponding skeleton is $*(x_1, \overset{child}{\rightarrow} (x_2,0), \overset{sib}{\rightarrow} (x_3,0), x_4)$, where the 0s correspond to the recursion points and $x_1..x_4$ are fresh variables. Because this stage of the algorithm is only concerned with finding the recursive backbone of the data structure, the skeleton discards all nonrecursive parts of the term tree, leaving in their place simple variables as place-holders. On the other hand, function symbols on the backbone are preservely in order to verify that these symbols are repeated recursively along the rest of the backbone. This is done by testing if t_{skel} can be "embedded" in every subtree rooted at an unfolding point. This notion of a term being embedded in another one is captured precisely by the binary relation $\leq$, defined inductively as

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

The idea is that $t_{skel} \leq t$ if for every path in t_{skel} that ends in a symbol 0 (i.e. a recursive point), there is a same path in t that ends in a subtree containing more unfolding points, while each variable in t_{skel}, being a place-holder for a nonrecursive subtree, maps to a subtree in t that contains no more unfolding points.

In Schmid's original work, R induces a valid segmentation if for each unfolding point u derived from R, $t_{skel} \leq u$ holds. In our case, because not all NULL nodes are actual unfolding points, an additional check is needed to weed out false positives. A NULL node is a valid recursion point only if for at least one unfolding point, the

subtree that is associated with the corresponding recursion point in the $\leq$ relation contains unexpanded nodes.

Figure 3.5 contains the algorithm for checking the validity of a segmentation. It does not actually compute t_{skel}. Instead, it compares the segmentation at the root of the term tree, from which t_{skel} is obtained, with all other segmentations one by one.

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 subterm pairs are replaced by the same variable throughout the whole term.

3. Find parameter substitutions. The subterms 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 subterm 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 $sub(x_j, r, \lambda)$.

```
is_valid_segmentation(R,t)
valid_NULL_nodes = ∅
for all unfolding point u below t do
  queue = {(t,u)}
  while queue ≠ ∅ do
    remove (n1,n2) from queue
    if n1 ∈ R then
      if n1 is a NULL node then
        if the subtree at n2 does not contain NULL nodes or unexpanded nodes
        then
          if n1 ∉ valid_NULL_nodes then
            R −= n1
          else
            return false
        else
          if the subtree at n2 contains unexpanded nodes then
            valid_NULL_nodes += n1
      else
        if the subtree at n2 does not contain NULL nodes or unexpanded nodes
        then
          return false
        for all child n1i of n1 do
          n2i is the i-th child of n2
          queue += {(n1i,n2i)}
    else if n1 can reach some r ∈ R then
      if n1 does not have same function symbol as n2 then
        return false
    else
      if the subtree at n2 contains unexpanded nodes but no NULL nodes then
        return false
for all r ∈ R do
  if r is a NULL node ∧ r ∉ valid_NULL_nodes then
    R −= r
```

Figure 3.5: Algorithm for Checking the Validity of a Segmentation

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

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 3.3(a), the recurrence body is shown in Figure 3.3(b) 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 Chapter 2.

3.2 What inductive recursion synthesis can and cannot do

The complete algorithm given in [27] 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.

Chapter 4

Truncation Points and Location Reasoning

To facilitate reasoning about local updates to recursive data structures, the idea of truncation points (Chapter 2) is introduced to model pointers to the interior of a data structure. When local updates occur, the data structure needs to be unfolded to expose specific points-to relations in its interior. Afterwards, the data structure needs to be folded back so as to reestablish global shape invariant. This chapter discusses the algorithms for unfolding and folding arbitrary recursive predicates. They correspond to two subroutines used in the symbolic execution rules of Chapter 2 – $unfold_{\Theta}(l{:}S, h)$ and $fold_{\Theta}(\Sigma)$ respectively.

4.1 Unfolding Recursive Predicates

$unfold_{\Theta}$ takes as arguments a state S and a heap location h, which is located either at the root of a recursive data structure or at the bottom sitting between the data structure and a truncation point. It unrolls the data structure at h according to the definition of the

recursive predicate that describes the data structure. As a result, the state S will contain explicit points-to assertions with h on the left hand side. $unfold_{\Theta}$ returns a set of states because case analysis is needed in the presence of truncation points.

If h is the root of a recursive data structure, indicated by the fact that it is the first parameter in the recursive predicate describing the data structure, then $unfold_{\Theta}$ needs to peel away the the top layer of the data structure. This 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. Since their exact positions relative to the root are not specified in the state, 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, it is necessary to enumerate all possible scenarios of relative positioning between h and the truncation points. Let n be the number of recursion points (Chapter 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 by calling $fold_{\Theta}$ to remove these truncation points. Furthermore, the possibilities for relative positioning are constrained by the invariant that the subheaps rooted at truncation points must be mutually disjoint. This means that if one truncation point h' is positioned at a recursion point under h, then no other truncation points can be truncation points of h'.

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* subtree of β_1.

Figure 4.1 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 case analysis algorithm serves both unrolling from top down and unrolling from bottom up, as shown in Figure 4.2.

```
case_analysis (h, T) :
valid_possibilities = {}
for all π do
  if ∀r.(∃t.π(t) = r) ⇒ (∄t'.t' ≠ t ∧ (π(t) = r ∨ π(t) = r̲)) then
    ok = true
    for all t ∈ T do
      if ∃ backward link t.n → h then
        let x_j be the parameter in A's definition s.t. x_1.n → x_j
        let r be the recursion point s.t. π(t) = r or r̲
        if π(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 ∧
            ∃r'.the recursive call at r' substitutes x_k for x_j then
            continue
        ok = false
        break
    if ok then
      valid_possibilities += π
return valid_possibilities
```

Figure 4.1: Algorithm for case analysis in $unfold_{\Theta}$

unfold$_\Theta$ (S, h) **:**
states = $\oslash$
if h is in recursive predicate $A(h, ...; [h'_1, ..., h'_m]$ **then**
  // Unroll from top down
  expand h according to the definition of A
  if $m > 0$ **then**
    for all $\pi \in case\_analysis(h, \{h'_1, ..., h'_m\})$ **do**
      $S' = duplicate(S)$
      assign $h'_1, ..., h'_m$ to positions specified by π
      states += S'
  else
    states += S
else
  // h is in recursive predicate $A(h_1, ..., h, ...)$
  // Unroll from bottom up
  expand h according to the definition of A
  if h_1 is a truncation point in another recursive predicate P **then**
    replace h_1 with h
  for all $\pi \in case\_analysis(h, \{h_1\})$ **do**
    $S' = duplicate(S)$
    assign h_1 to positions specified by π
    states += S'
return *states*

Figure 4.2: Algorithm for $unfold_\Theta$

4.2 Folding Recursive Predicates

The case analysis performed in $unfold_\Theta$ 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 $unfold_\Theta$ in an attempt to incorporate cut-out pieces of the original data structure back into it, thereby restoring the global invariant. $fold_\Theta$ takes a heap, looks for locations not pointed to by any live register, and tries to merge it into a neighboring data structure.

Like $unfold_{\Theta}$, 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 [7, 16], $p \rightarrow k * list(k,q) \rightsquigarrow list(p,q)$ 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) is absorbed.

Figure 4.3 contains the algorithm for $fold_{\Theta}$. It consists of three parts. As the first step, the set of *boundary locations* is identified. These locations must not be folded into neighboring recursive predicates. They include globals, cutpoints (Chapter 5), locations pointed to by procedure parameters, the return value register, and all other registers live at the current program point. The second part of the algorithm takes all recursive predicates that have truncation points and attempts to merge heap regions from below the truncation points into the predicates. A queue is used to hold heap locations that will be considered for the root of a new expansion to the recursive data structure. This test is performed by the subroutine *is_foldable*, which is given in Figure 4.4 and Figure 4.5. If the test fails, then the tested heap location becomes a new truncation point. Otherwise, all recursion points under the tested heap location according to the definition of the predicate are addeded to the queue. This process continues until the queue is exhausted. The last part of algorithm works backwards starting from all remaining recursive predicates (some recursive predicates might have been absorbed in the second part of the algorithm). In each iterative step, the subroutine *get_parent* obtains the parent of the current location. Location h_1 is considered as the parent of location h_2 if h_1 points to h_2 and h_2 inherits the access-path name of h_1, that is $h_2 = h_1.n$. *is_foldable* is then applied to the parent location to determine whether the heap region rooted at it matches the definition of the recursive

```
fold_Θ (Σ) :
boundaryLocations = {h ∈ Σ | h is pointed to by a procedure parameter ∨
                           h is pointed to by the return register ∨
                           h is pointed to by a live register ∨
                           h is a cutpoint ∨ h is a global}

top = {recursive predicates with truncation points}
while top ≠ ∅ do
  remove t from top
  newTruncationPoints = ∅
  queue = h.truncationPoints
  while queue ≠ ∅ do
    remove h from queue
    if !is_foldable(h, Θ(t), boundaryLocations) then
      newTruncationPoints += h
    else
      if h is a recursive predicate then
        queue −= h
      else
        queue += the set of recursion points under h according to Θ(t)
  t.truncationPoints = newTruncationPoints

bottom = {all remaining recursive predicates}
while bottom ≠ ∅ do
  remove b from bottom
  h = b
  hh = get_parent(h, Θ(b))
  while hh ≠ null ∧ is_foldable(hh, Θ(b), boundaryLocations) do
    bottom −= the set of recursion points under hh
    h = hh
    hh = get_parent(h, Θ(b))
  if b ∈ boundaryLocations then
    if b is not a recursion point under h then
      replace h with a predicate with the same definition as b
      h.truncationPoints += b
  else
    if hh ≠ null ∧ h ∈ boundaryLocations then
      replace h with a predicate with the same definition as b
```

Figure 4.3: Algorithm for $fold_\Theta$

```
is_foldable (h, definition, boundaryLocations) :
if h is a recursive predicate then
  return h ∉ boundaryLocations ∧ Θ(h) = definition
queue = {(definition.root, h)}
while queue ≠ ∅ do
  remove (h1, h2) from queue
  if h2 ∈ boundaryLocations then
    return false
  if mapping(h1) ≠ null then
    if mapping(h1) ≠ h2 then
      return false
  else
    mapping(h1) += h2
    if h1 is the first parameter in a recursive predicate then
      for all parameter pi of h1 do
        qi = find_param(h2, definition, i)
        if qi ≠ null then
          queue += (pi, qi)
        else
          return false
    else
      for all h1.n → h3 do
        if ∃h2.n → h4 then
          queue += (h3, h4)
        else
          return false
return true
```

Figure 4.4: Algorithm for *is_foldable*

```
find_param (h, definititon, i) :
if h is the first parameter in a recursive predicate then
   return the i-th parameter in that predicate
v_i = the i-th formal parameter in definition
queue = {(definition.root, h)}
while queue ≠ ∅ do
   remove (h_1, h_2) from queue
   for all h_1.n → h_3 do
      if h_3 = v_i then
         if ∃h2.n → h_4 then
            return h_4
         else
            return null
      else
         if ∃h2.n → h_4 then
            queue += (h_3, h_4)
return null
```

Figure 4.5: Algorithm for *find_param*

predicate. When the iteration terminates, either because a parent location cannot be found or because it fails the test *is_foldable*, the heap region from below the current location h that has been matched with the recursive predicate can be merged into it by replacing h with the recursive predicate. If the starting position b is a boundary location, then it is added as a truncation point of h.

is_foldable takes a heap location h, a heap formula that is the definition of a recursive predicate, and the set of boundary locations. It determines whether there exists a heap region rooted at h that has the exact structure as is described by the definition of the recursive predicate and does not contain any boundary location. If h is itself a recursive predicate with the same definition and h is not a boundary location, then *is_foldable* immediately returns true. Otherwise, it traverses the heap reachabe from h and the definition of the recursive predicate together in lock-step, returning false either when a structural mismatch is detected or when a boundary location is encountered.

4.3 A Detailed Example

```
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 4.6: Local modification to a tree in 181.mcf

To illustrate unfolding and folding, we will again turn to 181.mcf. Figure 4.6 contains a code fragment which cuts a subtree 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 subtree 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 4.1 to Table 4.4. 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 Chapter 2.1).

10

Σ_0 :
$mcf\_tree(r, null, null; q, t) * mcf\_tree(q, \beta_1, \beta_2) *$
$t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$t.sib\_prev \rightarrow \alpha_1 * t.sib \rightarrow \alpha_3 * mcf\_tree(\alpha_3, p, t)$

11

$\Sigma_{1,1}$: Unfold α_3
$mcf\_tree(r, null, null; q, t) * mcf\_tree(q, \beta_1, \beta_2) *$
$t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$t.sib\_prev \rightarrow \alpha_1 * t.sib \rightarrow \alpha_3 *$
$\underline{\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null) *}$
$\underline{\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)}$

$\Sigma_{1,2}$:
$mcf\_tree(r, null, null; q, t) * mcf\_tree(q, \beta_1, \beta_2) *$
$t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$t.sib\_prev \rightarrow \alpha_1 * \underline{t.sib \rightarrow null}$

12

$\Sigma_{2,1}$: Unfold α_1
$mcf\_tree(r, null, null; q, \underline{\alpha_1}) * mcf\_tree(q, \beta_1, \beta_2) *$
$\underline{\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null) *}$
$\underline{\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3 *}$
$t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$t.sib\_prev \rightarrow \alpha_1 * t.sib \rightarrow \alpha_3 *$
$\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null) *$
$\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$

$\Sigma_{2,2}$: Unfold p
$mcf\_tree(r, null, null; q, \underline{p}) * mcf\_tree(q, \beta_1, \beta_2) *$
$\underline{p.parent \rightarrow \alpha_8 * p.child \rightarrow \alpha_3 *}$
$\underline{p.sib\_prev \rightarrow \alpha_9 * p.sib \rightarrow \alpha_{10} * mcf\_tree(\alpha_{10}, \alpha_8, p) *}$
$t.parent \rightarrow p * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$\underline{t.sib\_prev \rightarrow null} * t.sib \rightarrow \alpha_3 *$
$\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null) *$
$\underline{\alpha_3.sib\_prev \rightarrow null} * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$

Table 4.1: Intermediate states of a tree update in 181.mcf

$\Sigma_{3,1}$: Unfold q
$mcf\_tree(r, null, null; q, \alpha_1)*$
$\underline{q.parent \rightarrow \beta_1 * q.child \rightarrow \beta_3 * mcf\_tree(\beta_3, q, null)*}$
$\underline{q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*}$
$\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$
$\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$
$\underline{t.parent \rightarrow q} * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$
$t.sib\_prev \rightarrow \alpha_1 * \underline{t.sib \rightarrow \beta_3}*$
$\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$
$\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$

13

$\Sigma_{3,2}$: Unfold q
$mcf\_tree(r, null, null; q, p)*$
$\underline{q.parent \rightarrow \beta_1 * q.child \rightarrow \beta_3 * mcf\_tree(\beta_3, q, null)*}$
$\underline{q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*}$
$p.parent \rightarrow \alpha_8 * p.child \rightarrow \alpha_3*$
$p.sib\_prev \rightarrow \alpha_9 * p.sib \rightarrow \alpha_{10} * mcf\_tree(\alpha_{10}, \alpha_8, p)*$
$\underline{t.parent \rightarrow q} * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$
$t.sib\_prev \rightarrow null * \underline{t.sib \rightarrow \beta_3}*$
$\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$
$\alpha_3.sib\_prev \rightarrow null * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$

Table 4.2: Intermediate states of a tree update in 181.mcf (Continued)

$\Sigma_{4,1}$: Unfold β_3
$mcf\_tree(r, null, null; q, \alpha_1)*$
$q.parent \rightarrow \beta_1 * q.child \rightarrow \beta_3*$
$q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*$
$\underline{\beta_3.parent \rightarrow q * \beta_3.child \rightarrow \beta_5 * mcf\_tree(\beta_5, \beta_3, null)*}$
$\underline{\beta_3.sib\_prev \rightarrow t * \beta_3.sib \rightarrow \beta_6 * mcf\_tree(\beta_6, q, \beta_3)*}$
$\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$
$\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$
$t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$
$t.sib\_prev \rightarrow \alpha_1 * t.sib \rightarrow \beta_3*$
$\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$
$\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$

14

$\Sigma_{4,2}$:
$mcf\_tree(r, null, null; q, \alpha_1)*$
$q.parent \rightarrow \beta_1 * \underline{q.child \rightarrow null*}$
$q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q)*$
$\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null)*$
$\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3*$
$t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null)*$
$t.sib\_prev \rightarrow \alpha_1 * \underline{t.sib \rightarrow null*}$
$\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null)*$
$\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$

Table 4.3: Intermediate states of a tree update in 181.mcf (Continued)

$\Sigma_{5,1}$:
$mcf\_tree(r, null, null; q, \alpha_1) *$
$q.parent \rightarrow \beta_1 * q.child \rightarrow t *$
$q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q) *$
$\beta_3.parent \rightarrow q * \beta_3.child \rightarrow \beta_5 * mcf\_tree(\beta_5, \beta_3, null) *$
$\beta_3.sib\_prev \rightarrow t * \beta_3.sib \rightarrow \beta_6 * mcf\_tree(\beta_6, q, \beta_3) *$
$\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null) *$
$\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3 *$
$t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$t.sib\_prev \rightarrow null * t.sib \rightarrow \beta_3 *$
$\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null) *$
l5 $\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$

$\Sigma_{5,2}$:
$mcf\_tree(r, null, null; q, \alpha_1) *$
$q.parent \rightarrow \beta_1 * q.child \rightarrow t *$
$q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q) *$
$\alpha_1.parent \rightarrow p * \alpha_1.child \rightarrow \alpha_7 * mcf\_tree(\alpha_7, \alpha_1, null) *$
$\alpha_1.sib\_prev \rightarrow \alpha_6 * \alpha_1.sib \rightarrow \alpha_3 *$
$t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$t.sib\_prev \rightarrow null * t.sib \rightarrow null *$
$\alpha_3.parent \rightarrow p * \alpha_3.child \rightarrow \alpha_4 * mcf\_tree(\alpha_4, \alpha_3, null) *$
$\alpha_3.sib\_prev \rightarrow \alpha_1 * \alpha_3.sib \rightarrow \alpha_5 * mcf\_tree(\alpha_5, p, \alpha_3)$

$\Sigma_{6,1}$: Fold $\alpha_1,\ \alpha_3,\ \beta_3$
$mcf\_tree(r, null, null; q) *$
$q.parent \rightarrow \beta_1 * q.child \rightarrow t *$
$q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q) *$
$t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$t.sib\_prev \rightarrow null * t.sib \rightarrow \beta_3 * mcf\_tree(\beta_3, q, t)$
fold
$\Sigma_{6,2}$: Fold $\alpha_1,\ \alpha_3$
$mcf\_tree(r, null, null; q) *$
$q.parent \rightarrow \beta_1 * q.child \rightarrow t *$
$q.sib\_prev \rightarrow \beta_2 * q.sib \rightarrow \beta_4 * mcf\_tree(\beta_4, \beta_1, q) *$
$t.parent \rightarrow q * t.child \rightarrow \alpha_2 * mcf\_tree(\alpha_2, t, null) *$
$t.sib\_prev \rightarrow null * t.sib \rightarrow null$

Table 4.4: Intermediate states of a tree update in 181.mcf (Continued)

Chapter 5

Interprocedural Analysis

The interprocedural algorithm is similar to the ones proposed by Gotsman et al. [9] and Rinetzky et al. [25]. All three algorithms separate the callee's local heap from the rest of the heap and tabulate summary transfer functions for reuse under equivalent calling contexts. Two things distinguish our analysis. First, the computation of summary transfer functions applies inductive recursion synthesis for termination. Second, recursion synthesis is also used to handle cutpoints [24] precisely in the presence of recursive procedures. In comparison, cutpoints are either bounded in number in [9] or summarized by a single cutpoint in [25], both leading to loss of precision.

We give the algorithm in its entirety in Section 5.1 and focus on the treatment of cutpoints and recursive procedures in Section 5.2.

5.1 Tabulation Algorithm

Like [9] and [25], at each procedure entry, the analysis splits the state into two disjoint pieces – a local heap and a frame. The local heap consists of heap regions reachable

from the actual parameters and from the globals referenced by the callee and all of its descendants in the call graph. [1] Based on the Frame rule [19], only the local heap needs to be considered when analyzing the behavior of the callee. Cutpoints, which refer to locations in the local heap other than the parameters and the globals that have pointers pointing to them from within the frame, need to be preserved when computing the transfer function so that the postheap upon returning from the callee can be properly pieced back together with the frame. This is achieved in our analysis by imposing a rule on the algorithm $fold_\theta$ such that no cutpoint is ever folded away.

Our interprocedural algorithm, shown in Figure 5.1 and Figure 5.2, traverses the callgraph from top down, starting with the entry of the *main* procedure. Each time a call site is encountered, the local heap of the callee is first extracted and then compared with all previously seen entry heaps of the callee. If a match is found, then the tabulated exit heap is used to update the caller's state. Otherwise, the callee needs to be analyzed with the local heap as entry heap. In addition to the benefit of reusing a transfer function under calling contexts that are different only in the frame portion of the heap, this top-down approach also means that the precise structure of the heap and all alias relationships are always known whenever the analysis steps into a callee and that the analysis only analyzes calling contexts that may arise in the execution of the program.

analyze_program $(prog)$ **:**
$stack = \{\langle main, S_0, \{\langle entry_{main}, S_0\rangle\}\rangle\}$
while *stack* is not empty **do**
  get $\langle proc, S_{entry}, worklist\rangle$ from *stack* top
  $analyze\_procedure(proc, S_{entry}, worklist)$

Figure 5.1: The Interprocedural Algorithm

[1] Information about the globals is easily gleaned by a pointer analysis applied before the shape analysis. This is discussed in Chapter 6.

```
analyze_procedure (proc, S_entry, worklist) :
while worklist is not empty do
  remove ⟨n, S⟩ from worklist
  if n is a call node then
    S_local = extract(S, callee)
    if summary(callee, S_local) ≠ ⊘ then
      for all S_exit ∈ summary(callee, S_local) do
        add ⟨n_return, combine(S_exit, S)⟩ to worklist
    else
      push ⟨callee, S_local, {⟨entry_callee, S_local⟩}⟩ on stack
      if callee is recursive then
        latest_entry_state_callee = S_local
      return
  else if n is an exit node then
    summary(proc, S_entry) += S
    if proc is recursive then
      latest_exit_state_proc = S
  else
    for all control flow edge n → n' do
      S' = transform(n → n', S)
      if n → n' is a back edge then
        if S' ∉ loop_invariant(n', S_entry) then
          if the loop has not iterated twice then
            add ⟨n', S'⟩ to worklist
          else
            recursion_synthesis(S')
            check_invariant(n, S')
            loop_invariant(n, S_entry) += S'
      else
        add ⟨n', S'⟩ to worklist
        if n' is a loop header then
          loop_invariant(n', S_entry) += S'
pop stack
if proc is recursive and caller is not in proc's callgraph SCC then
  for all procedure p in proc's SCC do
    recursion_synthesis(latest_entry_state_p)
    recursion_synthesis(latest_exit_state_p)
  for all procedure p in proc's SCC do
    check_summary(p, latest_entry_state_p, latest_exit_state_p)
    summary(p, latest_entry_state_p) += latest_exit_state_p
```

Figure 5.2: The Interprocedural Algorithm (Continued)

We now describe the algorithm and its notations in more details. In the interprocedural control flow graph of the program, each procedure has an entry node and an exit node, and each call site is represented by a call node and a return node. The algorithm maintains the following data structures:

- *stack*

  The *stack* models the actual call stack of the program. Each entry on the *stack* is a triple $\langle proc, S_{entry}, worklist \rangle$, representing the procedure currently being analyzed, together with its entry state and a worklist that records the current state in the symbolic execution of the procedure. Every time a callee needs to be evaluated under a new calling context, an entry for it will be pushed on the *stack*. When the evaluation terminates, the entry will be popped off the *stack*, exposing the caller's entry.

- *worklist*

  A *worklist* contains events $\langle n, S \rangle$. Each event represents a state S that holds right before an instruction n. Before a procedure is evaluated, a *worklist* is initialized to contain an entry state at the procedure entry. The subroutine *analyze_procedure* iterates through the *worklist*, terminating either when a call instruction is encountered and no previously tabulated summary transfer function of the callee can be used or when the *worklist* is exhausted, indicating that the analysis has reached the end of the procedure.

- *summary*

  This data structure tabulates summary transfer functions. For each procedure, it maps an entry state to a corresponding set of exit states.

- *loop_invariant*

  To reduce memory consumption of the analysis, we do not maintain states at all program points. Instead, the end result of the analysis consists only of loop invariants and procedure summaries. We believe that optimizations are most likely to query shape information at this level of granularity. Besides, the states associated with other program points can be easily derived from this chosen set of states, if needs be. The data structure *loop_invariant* maps each loop header and an entry state of the procedure containing the loop to a set of loop invariants that are valid under the given calling context. Additionally, *loop_invariant* also serves as a temporary place-holder for loop entry states so that at the end of each loop iteration the analysis can compare the current state with the previously remembered loop entry state to determine whether it has converged over the loop. If, in the case of the analysis diverging, inductive recursion synthesis is invoked to produce a loop invariant, it will be added to *loop_invariant* and the original entry states, subsumed by the invariant, will be removed.

The algorithm *analyze_procedure* removes one event from the *worklist* in each iterative step and performs different operations depending on whether the current event contains a call node, a procedure exit node, or a node that is neither. It invokes the following subroutines:

- $extract(S, callee)$

  Given a call site state S whose heap portion is denoted by Σ, *extract* determines two disjoint pieces of heap Σ_{local} and Σ_{frame} such that $\Sigma = \Sigma_{local} * \Sigma_{frame}$. Σ_{local} is defined as the part of Σ that is reachable from the actual parameters and the globals referenced by the callee. Let T denote the set of heap locations pointed to by the

actual parameters and the globals. Σ_{local} is the spatial conjunction of the smallest set W of atomic heap assertions that satisfies

1. $\{H(h) \mid h \in T\} \subseteq W$
2. $\{H(h) \mid \exists h', \Sigma' \,.\, H(h', h) \in R \wedge \Sigma = H(h) * \Sigma'\} \subseteq W$,

$H(h) ::= h.n \rightarrow h' \mid A(h, ...), H(h_1, h_2) ::= h_1.n \rightarrow h_2 \mid A(h_1, \ ... \ h_2, \ ...)$.

Next, *extract* maps the heap locations in Σ_{local} to the callee's namespace to form the callee's local state S_{local}.

- $combine(S_{exit}, S_{frame})$

  combine is performed when a tabulated entry state of the callee is found to match the local state determined by *extract*. It takes the corresponding exit state of the callee, projects it back to the caller's namespace, and returns the spatial conjunction of it and the frame.

- $transform(n \rightarrow n', S)$

  transform updates the state S according to the abstract semantics of n (which are described in detail in Chapter 2.2). One implementation detail concerns branch instructions. As mentioned before, for memory efficiency we do not keep intermediate states around, rather states are typically updated in place. However at split points of control flow, one state needs to split into two states if the analysis is to be accurate. Hence, if n is a branch instruction that tests pointer values or it has at least one store instruction that is control-dependent on it, then a copy of S is generated. This is the only situation, other than the tabulation of procedure summaries and loop invariants, in which duplication of states occurs.

transform is also responsible for performing the $fold_{\Theta}$ operation to bring states to the most compact form by eliminating truncation points that no longer have live registers pointing to them. This step could be performed at each program point, but it is only essential at call nodes, procedure exits, and loop headers. Hence to make the analysis fast, we choose to do the latter.

- *recursion_synthesis*(S)

  This subroutine performs inductive recursion synthesis on the heap portion of the state S, modifying it in place. If no recursion is identified, S is left unchanged.

- *check_invariant*(n, S)

  The result of *recursion_synthesis* is only a hypothesis for the loop invariant. To verify its validity, *check_invariant* performs symbolic evalution of the loop body one more time starting with the hypothesis as the state at the loop header. If the states eventually propagated to the loop back edge all match the hypothesis, then we have a valid loop invariant. Otherwise, *check_invariant* will report failure and halt the analysis.

- *check_summary*$(proc, S_{entry}, S_{exit})$

  Since *recursion_synthesis* is also called to generate hypotheses for the summary transfer functions of recursive procedures, their validity is verified by *check_summary*. (More on this in the next section.)

After the while-loop in *analyze_procedure* terminates, the procedure's corresponding entry is popped off the *stack*. Some extra operations are performed if the procedure is recursive. We discuss them in the next section.

5.2 Recursive Procedures and Cutpoints

In this section, we first show how inductive recursion synthesis is applied to infer the summary transfer functions of recursive procedures without introducing unnecessary approximation. Next we show that cutpoints can also be inductively summarized in the transfer functions, leading to a more precise treatment of them.

5.2.1 Basic Methodolgy

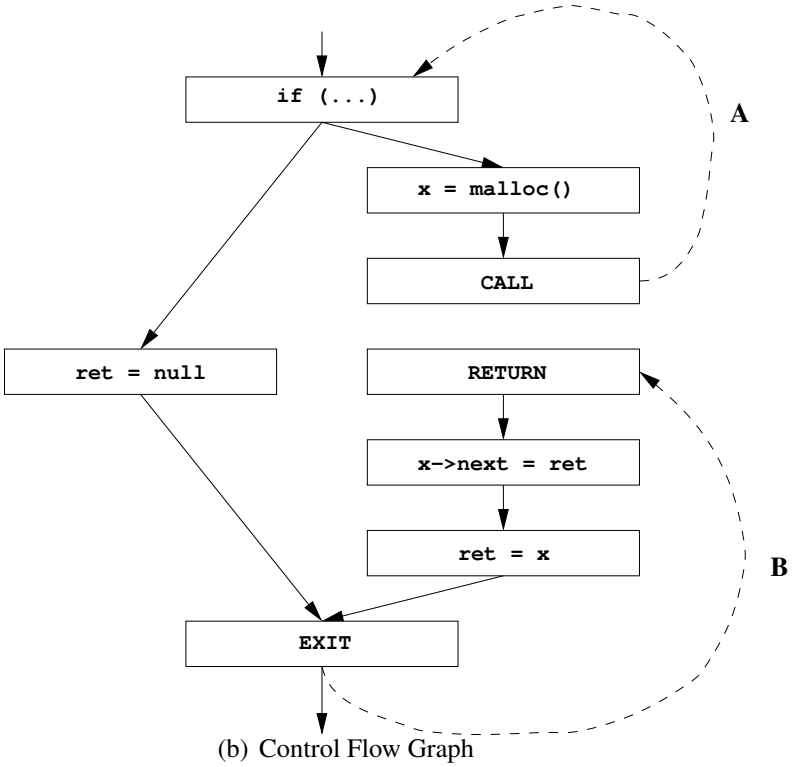
Recursive procedures are handled based on the same methodology by which loops are handled. When represented by control flow graphs, recursive procedures can be seen as a special type of loops with two kinds of loop back edges, one for recursive calls and one for recursive returns. Figure 5.3(a) shows a procedure `build_list`, which constructs a linked list by calling itself recursively. Its control flow graph is shown in Figure 5.3(b). There are two loops in the graph, whose respective back edges are drawn as dotted arrows. Back edge *A* connects the recursive call to `build_list` to its own entry node. Back edge *B* connects the exit node to the corresponding return node.

Just as for loops, inductive recursion synthesis is applied at back edges to detect recursive patterns in the pre/post conditions of recursive procedures. A group of mutually recursive procedures, identified as strongly-connected-components(SCCs) of the callgraph, is analyzed together as a unit. The whole process proceeds in the following steps:

1. Symbolically evaluate the SCC along a sample execution path. Record the states propagated to procedure entries and exits.

```
build_list(...) {
    if (...)
        return null;

    x = malloc();
    x->next = build_list(...);
    return x;
}
```

(a) build_list



(b) Control Flow Graph

Figure 5.3: A Recursive Procedure that Builds a Linked List

2. For each procedure entry/exit, apply inductive recursion synthesis to the last state associated with it. This yields a hypothesis of summary transfer functions for each procedure involved.

3. To ensure soundness, verify the validity of the hypothesis by checking that for each procedure, the postcondition can be arrived at from the precondition, assuming the hypothesized summary transfer functions for all recursive callees. If the test fails on any procedure, then report failure and halt. Otherwise, the hypothesis is valid.

Because the inferred summary transfer functions are checked for their correntness, the choice of sample execution paths does not affect soundness. However, we do want to pick the execution paths such that the states presented to the inductive recursion synthesis algorithm are most likely to exhibit detectable recursive patterns and to yield valid hypotheses of summary transfer functions. To this end, we design a heuristic for choosing sample execution paths guided that is by two criteria:

- The execution paths should be good representatives of the runtime behavior of the program. All control flow edges in the SCC needs to be visited at least once. Each procedure in the SCC need to be called recursively from within the SCC at least twice and only interprocedurally valid execution paths [28] (i.e. with matching calls and returns) are chosen.

- They should be as short as possible to minimize the time spent in symbolic execution.

The first step in the heuristic is to identify exit edges of the loops that result from recursive calls. At runtime, these control flows will cause the recursive procedures to terminate and to eventually return to a caller that is not in the same SCC. We refer to

these edges as *termination edges*. At each branch instruction, if one of its branches is a termination edge while other is not, then only one of the two branches is taken depending on whether all constituent procedures of the SCC have already been called at least twice. Until this condition is true, states are not propagated along the termination edge. The rationale behind this is that this ordering of termination and nontermination edges results in an interprocedurally valid execution path through the SCC and at the same time minimizes propagation of states.

The heuristic is implemented by the subroutine *transform* in Figure **??**, which returns NULL for the branch that is not taken. However, this heuristic is not in effect when verifying the inferred transfer functions through symbolic evalution. In the verification stage, we no longer need to iterate over the SCC. Each procedure is verified independently, assuming the transfer functions that are already inferred for its recursive callees. Hence states are propagated along both control flows out of each branch instruction.

In Figure 5.3(b), the sample execution path selected by our heuristic is given by the sequence of numbers on the control flow edges. The entry and exit heap formulas presented to the inductive recursion synthesis algorithm are show in Figure 5.4 on the top, and the result is shown at the bottom, which after verification turns to be a valid transfer function for `build_list`. (*ret* is a special register that holds the return value of a procedure.)

Entry heap: **emp**
Exit heap: $ret.next \rightarrow ret.next * ret.next.next \rightarrow ret.next.next$

Precondition: **emp**
Postcondition: $list(ret)$

Figure 5.4: Entry and Exit Heaps of `build_list`

5.2.2 Cutpoints

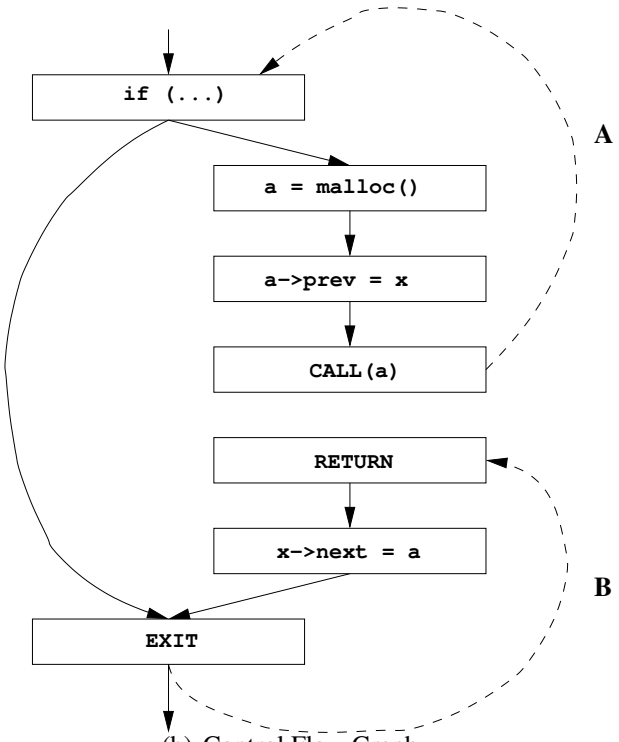
Cutpoints [24] are targets of dangling pointers from the frame to the local heap of a callee. They are also distinct from the formal parameters and the globals, therefore, it is possible that they are not referenced at all during the execution of the callee and hence are not really a part of the callee's memory footprint. Even though removing such heap nodes from the callee's pre and post conditions still yiels a valid summary of the callee's behavior, it is necessary to keep them there because upon returning to the caller, we need to instantiate them with actual values of the dangling pointers in order to properly combine the heap at the exit of the callee with the frame. Otherwise, the caller would have no idea where the dangling pointers point at after the merge.

This simple treatment of cutpoints becomes problematic in the presence of recursive procedures, because it can potentially lead to an infinite number of cutpoints. Consider the recursive procedure `build_dlist` in Figure 5.5, which constructs a doubly linked list by repeatedly calling itself and passing the node most recently added to the list as an argument. At the the call site `s1`, the heap location pointed to by `x` is a cutpoint, because it is reachable from `a`, hence a part of the callee's local heap, and it is also accessible from the caller (it is in fact referenced after the callee returns at instruction `s2`).

As the analysis iterates over this procedure, the heap at the entry of the procedure in the ith iteration is shown in Figure 5.6. The formal parameter is denoted by x_i. $x_0, ..., x_{i-1}$ are all cutpoints. In fact, they are instances of the formal parameters from previous iterations of the procedure. The number of cutpoints grows with the symbolic execution process, causing the analysis to diverge. However, their growth does exhibit a recursive pattern, hence it is natural to use recursion synthesis to capture this pattern and in the process summarizes cutpoints in the pre and post conditions inductively.

```
build_dlist(x) {
    if (...)
        return;

    a = malloc();
    a->prev = x;
s1: build_dlist(a);
s2: x->next = a;
}
```

(a) build_dlist



(b) Control Flow Graph

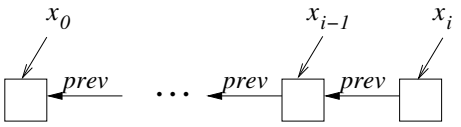
Figure 5.5: A Recursive Procedure that Builds a Doubly Linked List

Figure 5.6 also shows the heap at the exit of the procedure in the ith iteration. From these two heaps, inductive recursive synthesis is able to infer the pre and post conditions, shown at the bottom. The recursive predicates $list_{prev}$ and $dlist$ are defined as follows

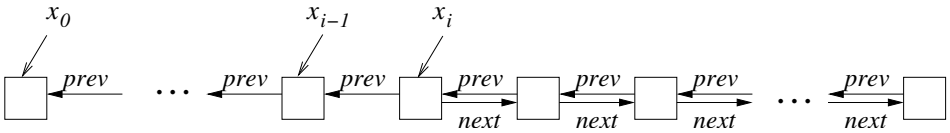
$$list_{prev}(x) \doteq (x = null \ \wedge \ \mathbf{emp}) \ \vee \ (x.prev \rightarrow \alpha \ * \ list_{prev}(\alpha))$$
$$dlist(x_1, x_2) \doteq (x_1 = null \ \wedge \ \mathbf{emp}) \ \vee \ (x_1.prev \rightarrow x_2 \ * \ x_1.next \rightarrow \alpha \ * \ dlist(\alpha, x_1)).$$

The cutpoints $x_0, ..., x_{i-1}$ are described inductively by the predicate $list_{prev}(x)$. When they are accessed at instruction `s2`, the predicate is unrolled to expose the points-to relations of x.

Entry heap



Exit heap



Precondition: $list_{prev}(x)$
Postcondition: $dlist(x, \alpha) \ * \ list_{prev}(\alpha)$

Figure 5.6: Entry and Exit Heaps of `build_dlist`

In [9], cutpoints beyond a preset bound are abstracted away by simply ignoring the dangling pointer pointing to them. This allows the extra cutpoints to be removed from the summary transfer functions. However, the loss of information about the targets of dangling pointers may lead to imprecise result. Although, as noted in [9], cases of infinite number of cutpoints are expected to be rare, using recursion synthesis to deal

with cutpoints presents a uniform approach to handle loops, both in the usual sense and in the form of recursive procedures. In cases where recursion synthesis is able to infer valid procedure summaries, the information is precise.

Chapter 6

Implementation

6.1 Code Pruning

The shape analysis algorithm is preceded by a code-pruning phase that identifies the part of the program where the accuracy provided by the shape analysis is needed. This part of the program manipulates pointers in recursive data structures, if any such data structure exists in the program. The need for extracting this part of the code is twofold. First, scalability to large programs is important for shape analysis to gain widespread application. To accurately model destructive updates to heap data structures, the algorithm of the shape analysis is flow-sensitive, meaning that it follows the program control flow and obtains different states at different program points. This is more computationally expensive than flow-insensitive algorithms. Furthermore, inductive recursion synthesis involves global inspection of states and backtracking, so it is also expensive. As a result, it is necessary to be precise about where to spend the computation resources. Second, nonessential fields in the data structures should be absent from the abstract states in order to minimize the possibility that the pattern detection algorithm of inductive recursion synthesis may be

confused. Such fields include nonpointer fields and pointer fields whose targets are never involved in a recursive data structure.

The code pruning phase includes a fast pointer analysis to identify recursive data structures present in the program, and a program slicing algorithm to prune away code that has no effect on the shape properties of these data structures.

6.1.1 Pointer Analysis

Since this shape analysis targets low-level code with no type information that may be used to identify recursive data strutures, a pointer analysis is used to essentially reverse-engineer the high-level type of each pointer. It is a modified Steensgaard's analysis [29] in which each inferred pointer type represents a set of runtime locations. For example, the "next" field in all nodes of a linked list are represented by a single inferred type. The pointer analysis is both flow and context insensitive and approaches linear time in complexity [29].

As the result of the pointer analysis, an inferred pointer type is be associated with each load/store instruction, which over approximates the set of locations accessed by the instruction. A set of *recursive pointer types* is then identified by looking for load instructions that may be involved in traversing recursive data structures. These load instructions share the property that the destination register contributes to the computation of the load address. Such a recurrence is easily detected by computing strongly connected components (SCC) of the reaching-definition graph. An SCC that contains more than one instructions or one whose single element feeds itself in the reaching-definition graph indicates the presence of cycles in the reaching-definition graph. To be conservative, all load instructions belonging to such SCCs are considered to be possibly traversing

recursive data structures. Finally, the pointer types associated with these load instructions form the set of recursive pointer types.

6.1.2 Program Slicing

The goal of the program slicing algorithm is to determine all code that may affect contents of memory locations represented by recursive pointer types (the contents must be pointers too). The algorithm is given in Figure 6.1. The set *trackedFields* contains all locations whose contents may affect memory writes to recursive pointer fields. It is initialized to the set of recursive pointer type. For each store instruction to a tracked field, all instructions (including branches and possibly crossing procedure boundaries) that lead to the computation of either the store address or the value to be stored are added to the slice. New pointer types that need to be tracked will be identified in the process, causing more instructions to be added. The algorithm terminates when no more instruction can be added.

6.2 Experimental Results

This analysis has been implemented in the Velocity compiler. Preliminary experimental results are reported in Table 6.1. In addition to 181.mcf which uses iterative algorithm to build and to traverse its data structure, the analysis was also evaluated on four Olden benchmarks which use recursive procedures. The second column lists the types of recursive data structures in each benchmark. In our experiments, the analysis was able to infer and maintain shape predicates that precisely describe these data structures.

Column 3 lists the number of assembly-level instructions in each benchmark. The number of instructions remaining after pointer analysis and program slicing is shown in

```
slice = ⊘
trackedFields = {recursive pointer types}
repeat
  newTrackedFields = ⊘
  for all store instruction s0 to a tracked field do
    visited = ⊘
    queue = {s0}
    while queue ≠ ⊘ do
      remove instruction s from queue
      visited += t
      slice += t
      if !(s is a branch instruction that does not test pointer values) then
        for all definition s' reaching a use in s do
          if s' ∉ queue ∧ s' ∉ visited then
            queue += s'
      b = the branch that s is directly control-dependent on
      if b ∉ queue ∧ b ∉ visited then
        queue += b
      if s is a load instruction ∧ the field f accessed by t contains a pointer ∧
        f ∉ trackedFields then
        newTrackedFields += f
  tracedFields += newTrackedFields
until newTrackedFields = ⊘
```

Figure 6.1: The Algorithm for Program Slicing

Benchmark	Data Type	# Insts	# Insts after slicing	Analysis time (s)		
				Pointer	Slicing	Shape
181.mcf	*mcf_tree*	2158	366	0.59	0.22	0.55
treeadd	binary tree	162	12	0.09	0.02	0.05
bisort	binary tree	423	70	0.16	0.05	0.38
perimeter	quaternary tree w/ parent links	624	31	0.20	0.06	0.10
power	lists	1054	19	0.37	0.07	0.06

Table 6.1: Experiment results

column 4, which is dramatically smaller for all five benchmarks. The time measurements was taken on a 3GHz P4 with 512KB cache and 2GB memory. The execution time is divided among the pointer analysis phase, the program slicing phase and the shape analysis phase. Except for the benchmark bisort, the shape analysis phase takes less time than does the pointer analysis, demonstrating the effectiveness of code pruning. The reason why shape analysis runs relatively longer on bisort is that the program slice consists mainly of two self-recursive functions and a pair of mutually recursive functions that manipulate a binary tree. The shape analysis spent much of its time symbolically unrolling these recursive functions.

Chapter 7

Bridging Optimizations and Shape Analysis

There are many applications for shape analysis. Being a more advanced type of pointer analysis, it can be used as a building block in program understanding tools. It can also be used to verify safety properties of a program. This chapter focus on another category of applications, which is aggressive code optimizations, particularly those that exploit coarse-grain parallelism at the loop and function level.

Consider a simple code example in Figure 7.1(a). If shape analysis can determine that the data structure traversed by the while loop is a simple linked list, then we know that there is no cycle and hence no sharing of any node in the list. As a result, different iterations of the loop can execute in parallel. Straightforward as this may sound, the question remains: how do optimizations extract this information from shape analysis? In other words, what is the bridge between optimizations and the internal representation of heap properties understood only by shape analysis?

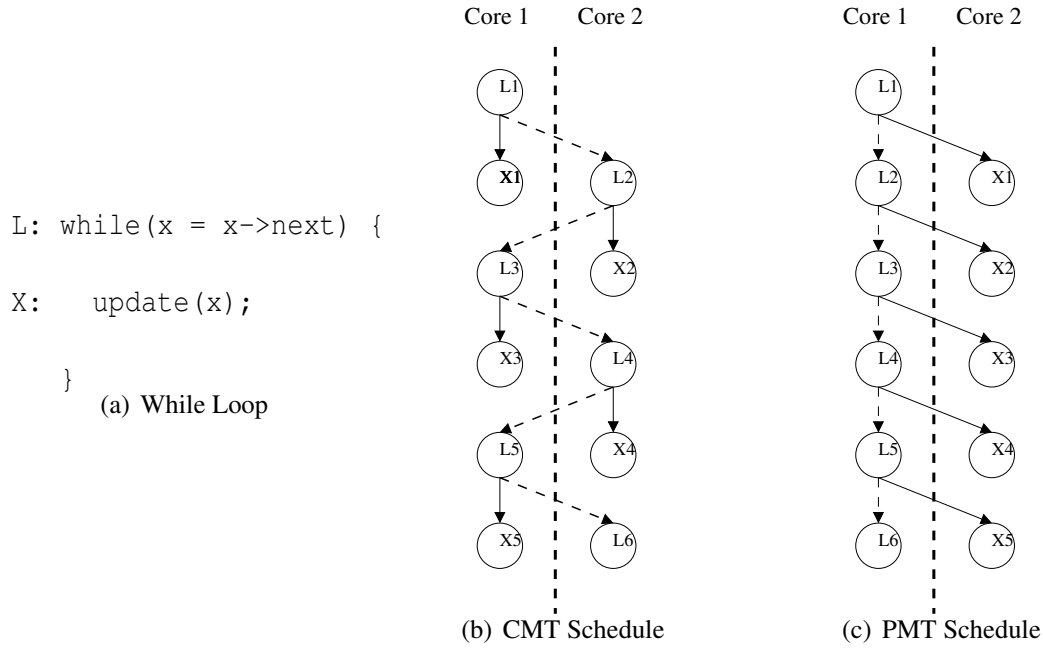


Figure 7.1: A While Loop that Traverses a Linked List

Dependence tests are one way to put queries to shape analysis. Memory dependences can be divided into those that are loop-carried (meaning that two instructions from different iterations of a loop may interfere with each other) and those that are not. The latter category is easy to determine. Shape analysis can return a points-to set for each instruction that accesses memory, just like a typical pointer analysis (but generally with higher precision). One only needs to compare the points-to set of two instructions to decide if they may ever access the same memory location. Loop-carried dependences are more intricate and are usually hard to determine just by using pointer analysis. This chapter will show how our analysis can be used to perform dependence test on loops that traverse recursive data structures.

Among such loops, the so-called "pointer-chasing" loops are particularly amenable to parallel transformations. The body of a pointer-chasing loop performs some computation on a distinct node of the data structure in each iteration, however the pointer to the next

node is computed using the pointer from the current iteration. The loop in Figure 7.1(a) demonstrates the simplest of such behavior. Pointer-chasing loops can be parallelized in various ways, including Cyclic Multithreading (CMT) [5] and Pipelined Multithreading (PMT) [21]. Figure 7.1(b) and (c) show the thread schedules produced from applying CMT and PMT transformations to the loop in Figure 7.1(a), assuming a two-core machine. The arrows represent data dependences. Solid arrows correspond to intraiteration dependences while dashed arrows correspond to loop-carried dependences.

The correctness of these optimizations all stems from the guarantee that no two iterations of the loop will visit the same node. The more precise a shape analysis is, the better it is able to prove such guarantees. For example, a data structure may as a whole contain cycles, but if it is only traversed along a subset of pointer fields that never reach the same node twice, then the pointer-chasing property still stands. This requires a shape analysis that can precisely describe the patterns of internal sharing in a recursive data structure and the capability to deduce certain properties from the patterns.

Following the approach of [8], the process of establishing pointer-chasing behavior takes two steps, as outlined in Figure 7.2. First, a "navigator" is found, which is the pointer used to traverse the data structure. Next, dependence test is performed to prove that in addition to the one due to the navigator, there is no other loop-carried dependence at all or only up to a certain dependence distance. The difference from [8] is that we perform more sophisticated tests in both steps so that more optimization opportunities can be captured.

I. Find a valid navigator.

   i. Identify the navigator and the navigator expression.

   ii. Determine if the data structure is acyclic considering only the pointer fields in the navigator expression.

   iii. Check that updates made to such fields in the loop do not break the acyclicness of the navigator.

II. Disprove all other loop-carried dependences.

Figure 7.2: Finding Pointer-Chasing Loops

7.1 Identifying the Navigator

If only nonspeculative optimizations are considered, then candidates for pointer-chasing loops must have a regular control flow pattern, that is, a single exit guarded by the loop condition test. This restriction can be lifted if the optimizations are able to squash speculatively executed loop iterations when the loop terminates irregularly.

The navigator can be identified by looking for recurrence in the chain of definitions that reaches the variables involved in the loop condition test. For completeness, we summarize the algorithm from [8] in Figure 7.3. Each definition in the chain is the unique loop-resident definition that definitely reaches the base variable used by its predecessor to access memory. If the same instruction is encountered twice when tracing the definition chain, then a potential navigator is found. An access path for the variable used in the loop condition test can be obtained from the definition chain. The base variable in this access path is the *navigator* and the portion of the access path involved in the recurrence is the *navigator expression*.

```
identify_navigator (loop) :
loopCond = loop.cond
pair = find_navigator(loopCond.lhsVar, loopCond, loop)
if pair = null then
  pair = find_navigator(loopCond.rhsVar, loopCond, loop)
return pair

find_navigator (var, varInst, loop) :
defChain = [ ]
defChain = get_loop_def_chain(var, varInst, loop, defChain)
if defChain = null then
  return null
navigatorExpr = get_expression(defChain)
navigator = get_base_variable(navigatorExpr)
return (navigator, navigatorExpr)

get_loop_def_chain (var, varInst, loop, defChain) :
defs = get_loop_defs(var, loop)
if defs contains only one element then
  def = get_element(defs)
  if def is already in defChain then
    return defChain
  if def definitely reaches varInst then
    defChain += def
    baseVar = get_base_var(def.rhs)
    return get_loop_def_chain(baseVar, def, loop, defChain)
return null
```

Figure 7.3: Algorithm for Identifying the Navigator (based on [8])

7.2 Determining if the navigator advances along an acyclic path

The shape analysis used in [8] can only categorize a data structure as being a tree, a DAG or a cyclic graphs. Hence even if the pointer fields used to advance the navigator will not result in the same node being visited twice, the navigator will be invalidated in this step as long as the data structure is labeled as cyclic. Because our shape analysis provides

detailed characterization of internal sharing in a data structure, a more advanced test can be performed.

From the recursive predicate associated with the data structure, a set of regular expressions that describe all possible cyclic paths in the data structure can be inferred. They are referred to as the set of *cyclic regular expressions*. The navigator expression yields another regular expression which summarizes all possible paths traversed by the navigator. If this regular expression does not intersect with any cyclic regular expression, then the navigator definitely advances along an acyclic path. Checking whether two regular expressions intersect can be done by first converting them into two deterministic finite automata M_1 nd M_2, constructing the automaton $M = M_1 \cap M_2$, and finally checking whether the final state of M is reachable from its start state. Although this approach offers the most precise result, implementing it may prove to be an overkill for the problem at hand simply because the navigator expression is usually very short. Instead, a simpler test can be performed, which may sometimes reject a valid navigator. This test only looks at the set of pointer fields involved in each regular expression, but ignores the order in which they need to be visited in order to form a cycle. It is then just a matter of checking that the set of pointer fields in the navigator expression does not contain any subset that is associated with a cyclic regular expression. This algorithm is given in Figure 7.4. The subroutine *get_fields* returns the set of distinct pointer fields present in an access path. The subroutine *get_cyclic_fields* returns a set, which, for each cyclic regular expression, contains the set of distinct pointer fields that appear in the regular expression.

Figure 7.5 contains the algorithm for *get_cyclic_fields*. Given a recursive predicate $A(x_1, ..., x_n)$, the first step is to compute the set of cyclic regular expressions. This is achieved by building the *parameter substitution graph* of the predicate. It is a directed graph $G = (E,V)$. Each node in V corresponds to an instance of parameter substitution,

```
check_acylic (predicate, navigatorExpr) :
navigatorFields = get_fields(navigatorExpr)
setOfCyclicFields = get_cyclic_fields(predicate)
for all s ∈ setOfCyclicFields do
  if s ⊆ navigatorFields then
    return false
return true
```

Figure 7.4: Algorithm for Determining if the Navigator Traverses an Acyclic Path

represented by a pair of numbers (r,i) where r identifies a recursive call site within the definition of the predicate and i identifies a parameter among $\{x_2, ..., x_n\}$. $x_2, ..., x_n$ are dangling pointers pointing to outside of the heap region rooted at x_1, hence they may potentially be backward links that form cycles in the data structure. For each node (r,i), if the actual value passed to x_i at call site r is some $x_j \in \{x_2, ..., x_n\}$ where x_j need not be different from x_i, then an edge is added from (r',j) to (r,i) for every call site r', including the case when $r' = r$. The edge represents that fact that the value of the dangling pointer x_i at r may be traced to that of x_j at r', that is, if the recursive call at r' is unfolded, the actual value passed to x_j will be passed on to x_i when the unfolded recursion makes the next level of recursive call at r. In Figure 7.6, the definition of *mcf_tree* is shown on the top. It has two recursive calls. The one on the second line is numbered call site 1 and the one on the third line is numbered call site 2. The parameter substitution graph of *mcf_tree* is shown at the bottom. It contains four nodes since there are two call sites and two dangling pointers x_2 and x_3. There are two edges, which enter node $(2,2)$ from $(1,2)$ and from $(2,2)$ itself. This is because at call site 2, the second parameter x_2 is substituted with x_2, which in turn can be traced back to each of the two call sites, introducing two possible values for the second parameter at call site 2.

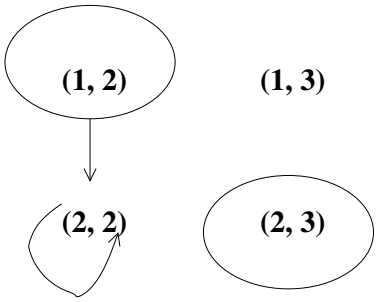
get_cyclic_fields $(A(x_1, ..., x_n))$ **:**
$E = \oslash, V = \oslash, G = (E,V)$
for all $(r,i), 2 \leq i \leq n$ **do**
  if x_i is not substituted with NULL at r **then**
    V += (r,i)
for all $(r_1,i),(r_2,j) \in V$ **do**
  if x_j is substituted with x_i at r_2 **then**
    E += $(r_1,i) \rightarrow (r_2,j)$
setOfCyclicFields $= \oslash$
for all $(r_0,i_0) \in V$ **do**
  $h =$ the heap location passed to x_{i_0} at r_0
  if $h \notin \{x_2, ... ,x_n\}$ **then**
    // (r_0,i_0) is an entry node
    for all $(r_0,i_0)\ (r_1,i_1)\ ...\ (r_l,i_l) \in get\_paths(G,(r_0,i_0))$ **do**
      $p =$ the access path from h to the heap location passed to x_1 at r_0
      for $k = 1..l$ **do**
        $w =$ the access path from x_1 to the heap location passed to x_1 at r_k
        $p = pw$
      $w =$ the access path from x_1 to x_{i_l}
      $p = pw$
      setOfCyclicFields += $get\_fields(p)$
return *setOfCyclicFields*

Figure 7.5: Algorithm for Inferring Sets of Pointer Fields that May Result in Cycles

The parameter substitution graph also has a set of entry nodes. An entry node (r,i) must satisfy the requirement that the substitution for x_i at r is not in $\{x_2, ..., x_n\}$, hence it is an end point in the iterative process of tracing the values of dangling pointers. In Figure 7.6(b), entry nodes are shown in circles. Each path in the parameter substitution graph that starts from an entry node correponds to a cycle in the data structure. Given such a path $[(r_0,i_0)\ (r_1,i_1)\ ...\ (r_l,i_l)], l \geq 0$, the corresponding cycle can be obtained by concatenating the following segments of access paths from the definition of the recursive predicate:

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

(a) Predicate Definition



(b) Parameter Subsitution Graph

Figure 7.6: The Parameter Substitutation Graph for *mcf_tree*

1. the access path that leads from the heap location passed to x_{i_0} at r_0 to the heap location passed to x_1 at r_0,

2. the access path that leads from x_1 to the heap location passed to x_1 at r_{i_k}, for $k = 1..l$,

3. the access path that leads from x_1 to x_{i_l}.

The presence of cycles in the parameter substitution graph itself indicates that there are infinitely many cycles in the data structure, which can be summarized by regular expressions with Kleene stars. For example, there are three sets of paths in the parameter subsitution graph of *mcf_tree*: $\{[(1,2)]\}$, $\{[(1,2)\ (2,2)],\ [(1,2)\ (2,2)\ (2,2)],\ ...\}$ and $\{[(2,3)]\}$, which yields three cyclic regular expressions: *child parent*, *child sib sib* parent* and *sib sib_prev*. Because we are not interested in the regular expressions per se, but rather in the pointer fields that they are made up of, we only need to look at the paths dicovered in

a depth first search of the parameter substitution graph with all back edges ignored. The subroutine *get_paths* in Figure 7.5 is reponsible for collecting such paths starting from a given entry node. For *mcf_tree*, it turns up exactly three paths: $[(1,2)]$, $[(1,2)\ (2,2)]$ and $[(2,3)]$, which correspond to the cycles *child parent*, *child sib parent* and *sib sib_prev*.

7.3 Validating the Navigator

After it has been determined that the navigator advances along an acyclic path, we still need to check that the loop does not alter pointer fields in the navigator expression in a way that could violate the guarantee of acyclicness. In [8], this step is performed by simplying checking that no field in the navigator expression is updated at all within the loop. This rule can actually be relaxed. As long as the dependence between the update and the traversal load is an anti-dependence [13], that is, the update always writes to a location that has been read in a previous iteration or in the same iteration but prior to the update, then the threads produced by both CMT and PMT are still correct. This is because due to the existence of interthread communication, both two transformations already ensure that the ith instance of the traversal load comes before all $(i+\delta)$th instances of the loop body.

Such anti-dependences often arise in loops that tear down pointer links as they traverse the data structure. For an instance of such behavior, consider the code example in Figure 7.7, which is a loop taken from the benchmark 181.mcf. This loop traverses the *mcf_tree* along the *parent* links. The navigator expression is `p->parent` at line 15. The only update to the field `parent` inside the loop is at line 7. The base variable used in the update is `t` which points to the ith node, while the load is performed on `p` which points to the $(i+1)$th node. Hence there is an anti-dependence between them.

```
while (t != jminus)
{
1   if (t->sib)
2       t->sib->sib_prev = t->sib_prev;

3   if (t->sib_prev)
4       t->sib_prev->sib = t->sib;
5   else
6       p->child = t->sib;

7   t->parent = q;
8   t->sib = q->child;

9   if (t->sib)
10      t->sib->sib_prev = t;

11  q->child = t;
12  t->sib_prev = 0;

    ...

13  q = t;
14  t = p;
15  p = p->parent;
}
```

Figure 7.7: A Loop that Traverses and Modifies a Tree in 181.mcf

7.4 Disproving Other Loop-Carried Dependences

The existence of loop-carried dependences in addition to the one involving the navigator determines whether the computation part of the loop from different iterations may overlap. The CMT transformation requires that no such dependence exists. While the particular instance of PMT transformation in Figure 7.1(c) does not depend on this, disproving such dependences would allow multiple worker threads instead of one to be spawned to receive the values produced by the traversal thread. If loop-carried dependences do

exist but the dependence distances [18] can be determined, then multiple worker threads may still be spawn, the total count of which is bounded by the smallest dependence distance. Dependence distance is originally a concept in array dependence analysis. It refers to difference between values of the loop induction variable that cause two array index expressions to access the same array element. It can be used to characterize access patterns of elements in a linked data structure along the same vein. Here the role of the loop induction variable is played by the navigator.

Ghiya et al. [8] use a simple test to determine whether additional loop-carried dependences exist. For each pointer in the loop that is not loop-invariant, it is possible to construct an access path with respect to the navigator, called the "navigator access path". Since only pointer fields may possibly lead to a neighboring node of the one currently being visited by the navigator, the test concludes that two navigator access paths do not lead to a loop-carried dependence if neither of them includes any pointer field.

To be more precise in the case where the access paths do involve pointer fields, we propose another test to take advantage of the information provided by the shape analysis. The basic idea is that inside a recursive data structure some pointer fields never point to the same node. For example, the left child pointers and the right child pointers in a binary tree always point to different nodes. Hence if the loop itself never assigns a right child pointer to a left child pointer, then it is guaranteed that no two such pointers inside the loop may alias, no matter what iterations they are from. If, on the other hand, the loop does assign a right child pointer to a left child pointer, then a dependence distance may be computed. Of course, for correctness, all expressions whose values may be propagated to a given pointer through a series of assignments need to be checked. This is similar to constructing def-use chains, but in this case, a definition and a use are not tied together by shared variable name, but by may-alias relationship instead. In this test, two pointer

expressions are considered to be may-aliases if they access the same pointer field and their respective base pointer subexpressions cannot be proven to not alias.

Since proving that two pointers do not alias will affect the dependence test on other pointers, the overall algorithm as shown in Figure 7.8 is iterative, terminating when a fixed-point is reached. It invokes the subroutine *test_pair* on each pair of pointers and records the result in the map *distance*. The algorithm for *test_pair* is given in Figure 7.9. For a pointer $expr_1$ in instruction $inst_1$ and a pointer $expr_2$ in instruction $inst_2$, *test_pair* returns a dependence distance if the pointers may access the same node from two iterations. 0 is returned if they may access the same node only from the same iteration. ∞ is returned if they never alias with each other at all. It first checks that according to the predicate associated with the data structure, the two pointers do not alias. This is true if in the definition of the predicate, the respective fields accessed by the two pointers point to heap locations that are passed as the first argument to distinct recursive calls to the predicate itself. If this check fails, then a conservative dependence distance of 1 is returned to indicate that every iteration is dependent on the previous one. Otherwise, two dependence distances are computed, one for potential assignment of $expr_1$ to $expr_2$ and the other for assignment in the opposite direction. One of them is returned based on the following categorization of dependence distances in descending order of priority: smaller non-0 non-∞ numbers, larger non-0 non-∞ numbers, 0 and ∞.

In Figure 7.9, the recursive function *trace_definition_chain* explores all possible sequences of assignments that may result in a value of $expr_1$ being written to $expr_2$. The last argument in the function is the set of assignments in a particular definition chain that currently reaches $inst_2$. It ensures that these assignments are not revisited when the chain is extended beyond $inst_2$, thereby guaranteeing that the function terminates.

```
loop_carried_dependence_test (loop, predicate) :
repeat
  repeat = false
  for all expr1, inst1, expr2, inst2 do
    d = test_pair(expr1, inst1, expr2, inst2, predicate)
    if d ≠ distance(expr1, inst1, expr2, inst2) then
      repeat = true
      distance(expr1, inst1, expr2, inst2) = d
until !repeat
```

Figure 7.8: Algorithm for Detecting Loop-Carried Dependece

The function *get_distance* computes the distance of a loop-carried dependence resulting from a given pointer assignment. It constructs the navigator access paths for the left and right hand sides of the assignment respectively. Let $base_1.field_1$ and $base_2.field_2$ be the two access paths. If $base_1$ points to the Δ_1th node from the navigator and $base_2$ points to the Δ_2th node from the navigator, then the desired distance is $\mid \Delta_2 - \Delta_1 \mid$. If either navigator access path cannot be constructed, then the dependence distance cannot be computed precisely, in which case the conservative solution 1 is reported.

Ghiya et al. [8] describe how to construct navigator access paths. A chain of definitions that starts from the navigator and definitely reaches the given use is first built, from which the access path is then extracted. The first step is similar to *get_loop_def_chain* in Figure 7.3, terminating either when the navigator is encountered or when a unique and definite definition cannot be found, in which case the function *get_navigator_access_path* returns with a null value.

We note that in each iteration of the outer loop of *loop_carried_dependence_test*, *test_pair* need not be applied to all pairs of pointer expressions. First of all, a pair of expressions needs to be tested only if they are the base pointers in two memory accesses to the same field, one of which is a write. Secondly, two pairs of expressions ($expr_1$, $inst_1$; $expr_2$, $inst_2$) and ($expr_3$, $inst_3$; $expr_4$, $inst_4$) have the same dependence

test_pair ($expr_1$, $inst_1$, $expr_2$, $inst_2$, *predicate*) **:**
if *may_alias*(*predicate*, *get_field*($expr_1$), *get_field*($expr_2$)) **then**
  return 1
d_1 = *trace_definition_chain*($expr_1$, $inst_1$, $expr_2$, $inst_2$, $\oslash$)
d_2 = *trace_definition_chain*($expr_2$, $inst_2$, $expr_1$, $inst_1$, $\oslash$)
if $d_1 = \infty \vee (d_2 \neq 0 \wedge d_2 < d_1)$ **then**
  return d_2
else
  return d_1

trace_definition_chain ($expr_1$, $inst_1$, $expr_2$, $inst_2$, *visited*) **:**
$d = \infty$
if $inst_2 \notin$ *visited* **then**
  for all assignment *inst* from which $inst_2$ is reachable **do**
    if *get_field*(*inst.lhs*) = *get_field*($expr_2$)$\wedge$
      distance(*get_base*(*inst.lhs*), *inst*, *get_base*($expr_2$), $inst_2$) $\neq \infty$ **then**
      $d' =$ *get_distance*(*inst*)
      if *get_field*(*inst.rhs*) $\neq$ *get_field*($expr_1$) **then**
        d' += *trace_definition_chain*($expr_1$, $inst_1$, *inst.rhs*, *inst*, *visited* + $inst_2$)
      if $d = \infty \vee (d' \neq 0 \wedge d' < d)$ **then**
        $d = d'$
return d

get_distance (*inst*) **:**
$path_1$ = *get_navigator_access_path*(*inst.lhs*, *inst*)
if $path_1$ = **null** **then**
  return 1
$path_2$ = *get_navigator_access_path*(*inst.rhs*, *inst*)
if $path_2$ = **null** **then**
  return 1
Δ_1 = the number of navigator expressions in $path_1$
Δ_2 = the number of navigator expressions in $path_2$
return $\mid \Delta_2 - \Delta_1 \mid$

Figure 7.9: Algorithm for *test_pair*

relation if 1). $expr_1$ and $expr_3$ access the same pointer field or variable, which is not updated between $inst_1$ and $inst_3$, and 2). the same is true for $expr_2$ and $expr_4$. For the code example in Figure 7.7, only five pairs of pointer expressions are interesting. The result of running *loop_carried_dependence_test* on them is listed below.

Pairs of Pointer Expressions	Dependence Distances
$(p,6;\ q,8)$	2
$(t,2;\ t \rightarrow sib,2)$	∞
$(t,2;\ t \rightarrow sib,10)$	∞
$(t \rightarrow sib,10;\ t \rightarrow sib,2)$	2
$(t,1;\ t \rightarrow sib\_prev,4)$	∞

Chapter 8

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 postconditions by updating the preconditions in-place [2]. It does not by itself yield a suitable abstract domain due to the lack of a guarantee for convergence. Two analyses of list- processing programs [7, 16] 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 [9], limited to predefined predicates as well. [3] 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 [14] can also discover recursive predicates automatically. The abstract domain of their analysis consists of shape graphs whose summary nodes have grammars associated with them to describe the shapes of the concrete structures represented by the summary nodes. Their grammars can have parameters, an implicit self parameter referring to the root of the concrete structure and an explicit parameter representing the target of a cyclic backward link. These grammars

can be translated to equivalent recursive predicates defined in separation logic. The first major difference between our work and theirs is that while their grammars allow only one explicit parameter, our recursive predicates can have an arbitrary number of parameters and an arbitrary number of truncation points. As a result, their analysis cannot describe data structures with multiple backward links such as the *mcf_tree*. Neither can it handle multiple pointers to the interior of a data structure. Another difference is that they arrive at grammar definitions by using a "cut" rule that removes cyclic connections between two shape graph nodes, therefore their analysis cannot handle DAGs at all. By comparison, inductive recursion synthesis is designed to detect all patterns that are recursive in nature, not any particular type of patterns, thus it represents a more general way of inferring recursions.

Outside of separation logic, various types of heap abstraction have been proposed to recover the precision lost due to approximations such as k-limiting. Nonuniform alias analyses [6, 32, 33] distinguish between elements of recursive data structures by correlating aliasing relationships with the positions of the elements in the data structures. For example, one of the aliasing relationships may be "the ith node of list A may alias with the $2i$th node". The positions are represented either by timestamps on abstract heap nodes [33] or by numbers of recursive field traversals in access paths [6, 32]. Describing tree-based data structures with a counter system such as this is not intuitive and may not be able to precisely capture complex internal sharing patterns.

3-valued logic analysis (TVLA) [26] is a parametric shape analysis framework where instrumentation predicates are associated with summary nodes to provide more precise descriptions of the concrete nodes. Their predicates are written in first-order logic and are no less expressive than ours. However, until their recent work on automatically generating new predicates [15], the instrumentation predicates had to be provided to the

shape analysis framework by the user. Like our work, [15] uses inductive learning to discover instrumentation predicates. But the particular technique they use is a completely different one from inductive recursion synthesis. It is based on successive refinement given positive and negative examples. Although in principle their predicates can describe complex data structures such as the *mcf_tree*, the inference of such recursive predicates is not demonstrated in their paper. Instead, the paper shows how to discover predicates that describe sortedness of a list and whether a heap node may be shared.

Chapter 9

Conclusion and Future Work

This dissertation has taken steps to allow separation logic based shape analysis to be applied to a wide range of programs without a priori restrictions on the types of data structures. This chapter summarizes the contributions made by this work and discusses possible directions for future work.

9.1 Summary

This dissertation presented an interprocedural shape analysis that uses separation logic formulas to describe heaps. The combination of two novel techniques, inductive recursion synthesis and generic recursion unrolling/rolling based on truncation points, makes separation logic based program analysis applicable beyond simplistic data structures. It has been shown that the analysis can determine the exact shapes of data structures in both iterative and recursive programs.

Different from the common approach of guaranteeing termination through widening (i.e. approximations), inductive recursion synthesis is inspired by inductive reasoning,

the ability to draw general conclusions from particular observations. The conclusions, if verified to be valid, do not suffer precision loss that results from approximations. Inductive recursion synthesis also represents a new approach to loop invariant inference, which frees logic-based program reasoning from having to rely on user-supplied invariants.

In addition to loops, recursion synthesis is also used to handle recursive functions. Some form of k-limiting (with k often equal to 1) is usually applied in the unrolling of recursive functions to ensure finiteness of the abstract domain. Approximations like this is avoided in our analysis by sampling an interprocedurally-valid execution path through the recursive functions and then applying inductive recursion synthesis. Another source of approximation that is also avoided is the need to bound the number of cutpoints that arise from local reasoning of callees.

To make the results of the shape analysis readily usable by code optimizations, a loop-carried dependence test for recursive data structures is designed. It improves upon a previous dependence test with new algorithms that query the shape analysis about existence of cycles in a data structure and whether a node may be shared by two pointers. The shape analysis is not only able to discover the precise shapes of the data structures, it is also able to reason about the shapes and prove properties about them. This test also computes dependence distances when loop-carried dependences cannot be disproved. All of these features contribute to more optimization opportunities being identified.

9.2 Future Directions

Many opportunities exist for improving and extending this work. For starters, applying the analysis to a wider set of benchmarks would help to formulate more precisely the conditions under which inductive recursion synthesis may fail. Our evaluation has shown

that it works very well on data structures that contain cycles. Though in theory it can also work for DAGs, as pointed out in Chapter 3.2, it is possible for the code to not reveal much clue as to the shape of the data structure. Hence it would be interesting to see how successful the analysis is on real applications.

A more graceful recovery from the situations where recursion synthesis does fail is also desirable. Instead of simply halting when the analysis fails to converge over a loop, extending the symbolic evalution of the loop for a couple of more iterations may uncover more information about the data structure that ultimately leads to a valid recursive description. If that fails, a different kind of predicates can be used to express that all pointers to a particular data structure may alias, allowing other parts of the heap to be analyzed as usual. Another possibility is to prompt the user for further information and then proceed.

Another area of improvement concerns the scalability of the shape analysis to large system software with millions of lines of code and hundreds if not thousands of individual data structures. At issue here is probably not memory consumption because the number of symbolically executed iterations per loop is bounded by a constant, which means the size of the abstract state is also bounded. A more real concern comes from the fact that inductive recursion synthesis is applied to all top-level term trees (essentially all data structures) each time the analysis needs to terminate over a loop; this may slow down the analysis.

Inductive recursion synthesis can be extended to handle arrays. For example, for an array of pointers to linked lists, it is often important to maintain in the abstract state that the array elements point to distinct lists. By introducing to the logic an additional heap assertion for pointer arithmetic $h_1 + n = h_2$, the state after symbolically unrolling of the loop will reveal patterns in the pointer arithmetic as well (arrays can be thought

of in terms of recursion too), just like it does for points-to relations. Adding support for pointer arithmetic in this way would facilitate reasoning about heap structures that involve a combination of arrays and linked data structures. But since the same location can be reached by both pointer arithmetic and access paths, decisions need to be made as to when to use pointer arithmetic to name the location and when to use access paths. Choosing the appropriate names has a huge effect on whether a valid recursion can be inferred.

Finally, evaluating the impact of shape analysis on automatic code parallelization and exploring other types of optimization opportunities would help increase the usability of shape analysis. Decoupled Software Pipelining (DSWP) [22, 20] is a new technology that extracts multiple concurrently executable threads from a sequential loop by identifying strongly-connected components (SCC) in the loop's dependence graph. It holds the promise of dramatically speeding up sequential programs on multi-thread and multi-core architectures. Without shape analysis, DSWP will be constrained by conservative memory dependence information, resulting in small number of large SCCs, which in turn leads to small number of threads. With shape analysis, not only can more threads be extracted, multiple threads can be spawned to execute a portion of the loop from different iterations. Another potential optimization opportunity is that linked lists can be converted to arrays to speed up their accesses.

9.3 Closing Remarks

Despite the higher level of accuracy offered by shape analysis over traditional pointer analysis, it is far from being widely deployed in optimizing compilers due to the fact that in general, its algorithm is necessarily more complex and it consumes even more memory

and execution time. Separation logic is an important step towards providing intuitive and efficient way of reasoning about the heap. However, it cannot be widely used in automatic code analysis and optimization unless the analysis can discover recursive predicates and arrive at loop invariants without user intervention. This dissertation takes a significant step in this direction by equipping shape analysis with the ability of inductive reasoning. It holds the promise of allowing the advantages of separation logic as a heap formalism to be exploited beyond the domain of program verification, in aggressive optimizing compilers that can better utilize the next generation of architectures.

Bibliography

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

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

[3] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Lecture Notes in Computer Science*, volume 4134, pages 182–203. Springer-Verlag, 1995.

[4] 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*, pages 296–310, June 1990.

[5] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–884, 1986.

[6] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

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

[8] R. Ghiya, L. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive data structures. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 159–173, March 1998.

[9] 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.

[10] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–265, New York, NY, USA, 2007. ACM.

[11] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[12] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[13] D. J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, New York, NY, 1978.

[14] 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.

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

[16] 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.

[17] A. Mœller and Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 221–231, 2001.

[18] Y. Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, University of Illinois, Urbana-Champain, IL, February 1971.

[19] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Lecture Notes in Computer Science*, volume 2142, pages 1–19. Springer-Verlag, 2001.

[20] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, pages 105–116, November 2005.

[21] D. A. Padua. *Multiprocessors: Discussion of some theoretical and practical problems*. PhD thesis, University of Illinois, Urbana, IL, November 1979.

[22] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[23] 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.

[24] 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 International Symposium on Principles of Programming Languages*, pages 296–309, January 2005.

[25] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. Technical Report 26, Tel Aviv University, November 2004.

[26] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

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

[28] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

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

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

[31] G. Tan and A. W. Appel. A compositional logic for control flow. In *Lecture Notes in Computer Science*, volume 3855, pages 80–94. Springer-Verlag, 2006.

[32] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2-3):223–248, November 1999.

[33] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Proceedings of the 9th International Symposium on Static Analysis*, pages 36–51, 2002.