PRACTICAL TECHNIQUES FOR INTERPROCEDURAL HEAP ANALYSIS

by
Rakesh Ghiya

School of Computer Science
McGill University, Montreal

January 1996

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

Abstract

Accurate alias analysis is critical for optimizing/parallelizing compilers that support languages with pointers. Efficient techniques have been developed to calculate aliases introduced by pointers to named memory locations (typically on the stack). However, practical and effective techniques for detection of aliases induced by heap-based dynamic data structures, have yet to be developed. Existing approaches are either efficient but overly conservative, or sophisticated but expensive.

In this thesis, we present a new and practical approach for analyzing the alias properties of heap data structures. The important features of our approach include: (i) we analyze heap-directed pointers after resolving the points-to relationships of stack-directed pointers, (ii) we use a *storeless* model and estimate the heap structure by abstracting the relationships between heap-directed pointers, and not by explicitly abstracting the heap as a graph, and (iii) we employ a hierarchical approach and design different abstractions to solve the problem at different levels of complexity.

We present a hierarchy of three practical abstractions for analyzing heap data structures, namely connection, direction and interference matrix abstractions. These abstractions respectively capture the following *boolean* relationships between any two given heap-directed pointers: (i) if they can point to the same heap data structure, (ii) if an access path exists between the heap objects they point to, and (iii) if they can access a common heap object. Connection matrix information helps detect pointer accesses to completely disjoint data structures. The other two abstractions work together to identify if the given program builds tree-like or dag-like structures.

We have implemented context-sensitive interprocedural analyses for these abstractions in the framework of the McCAT C compiler. For each abstraction, we first present basic analysis rules applicable to any language that supports pointers. We then describe C specific features of the analyses. We demonstrate the effectiveness of the analyses by providing examples as well as empirical results for real C programs.

Résumé

L'analyse précise d'alias est critique pour les compilateurs optimisateurs/ parallelisateurs, supportant des langages qui utilisent des pointeurs. Des techniques efficaces ont été développées pour calculer les alias introduits par des pointeurs qui pointent sur des locations mémoires nommées (typiquement sur la pile). Pourtant des techniques efficaces et pratiques pour détecter les alias introduits par les structures de données dynamiques basées sur le heap, sont encore à être développées. Les approches qui existent maintenant sont: ou efficaces mais très conservatives, ou compliquées mais chères.

Dans cette thèse, nous présentons une nouvelle méthode qui est pratique pour analyser les propriétés des structures de données de type 'heap'. Les principales caractéristiques de notre méthode sont: (i) nous analysons les pointeurs pointant sur le heap après avoir résolu les relations 'pointe-sur' des pointeurs pointant sur la pile, (ii) nous utilisons un modèle qui ne requiert aucune mémoire et nous estimons les structures du heap en faisant abstraction des relations entre les pointeurs qui pointent sur le heap, et non pas en faisant abstraction du heap comme un graphe, et (iii) nous employons une approche de type hierarchique et nous avons conçu différentes abstractions pour résoudre le problème à plusieurs niveaux de complexité.

Nous présentons une hiérarchie de trois abstractions pratiques pour analyser les structures de données de heap: abstractions de matrices de connection, de direction, et d'interférence. Chacune de ces abstractions reconnaît une relation booléenne entre deux pointeurs qui pointent sur un heap. Respectivement: (i) si ces pointeurs peuvent pointer sur la même structure de donnée du heap, (ii) s'il existe un chemin d'accès entre les objets du heap qu'ils pointent, et (iii) s'ils peuvent accéder à un objet commun du heap. La matrice de connection aide à détecter l'accès des pointeurs à des structures de données complètement disjointes. Les deux autres abstractions

identifient ensemble si le programme considéré construit des structures de données de type arbre ou 'dag' (graphe dirigé sans cycle).

Nous avons implémenté des analyses inter-procedurales qui tiennent compte du contexte pour ces abstractions dans l'environnement du compilateur C McCAT. Pour chacune de ces abstractions, nous présentons d'abord les règles d'analyse de base appplicable a n'importe quel langage supportant des pointeurs. Puis, nous décrivons les caractéristiques qui sont spécifiques pour C. Nous démontrons aussi l'efficacité de ces analyses en présentant des exemples ainsi que le résultat d'éxperiences réalisées avec de véritables programmes écrits en C.

Acknowledgments

Many thanks to:

Laurie Hendren for being a very caring and cheerful advisor. She has provided constant encouragement and kept me in good spirits all through. The numerous discussions I had with her, greatly helped me in making my fuzzy ideas concrete and in maintaining my focus. She has also supported my studies and saved me from the worldly worries. It has been fun sharing the penchant for pointers with her.

Professor Guang Gao for providing clear insights into the compiler-architecture relationship through his foundational courses. He always greeted me with a smile and boosted my morale with words of encouragement. I must also add that his keen sense of humor has given me many light moments during his lectures and seminars.

My mentors at TRDDC at Pune in India, for the first lessons. Professor Kesav Nori initiated me into the realm of compilers and Hemant Pande inducted me into pointer research.

Maryam Emami for making things tractable. She solved the first half of the problem and provided me a framework to attack the second part. She patiently answered all my questions, and many a times took pains to explain intricate details with elegant examples. I owe her a debt of gratitude.

The multi-cultural melange of people in the ACAPS lab, for adding much color to life. Chris Donawa showed me the ropes around in Montréal and provoked me into many interesting discussions of all kinds. Ana Erosa made us jealous with really long holiday trips to Europe, and kept us happy with her parties. Luis Lozano always cheered me with a warm hello, and has been a considerate friend. Bhama, Cecile, Maryam and Justiani arranged some wonderful get-togethers, which will always remind me of the good old days. This pack however remains incomplete without including Ian and Helga, who have always been great fun.

The fun friends in Montréal, for always having time for me. Nasser Elmasri provided company during the late night hacking sessions. He consistently humbled me on the squash courts, but invariably consoled me with a dinner invitation. Alain Turki and Herve Avril always invented attractive alternatives to work, and never said no to a snooker game. And Dhrubajyoti Goswami has been a good-natured friend since the first days in Montréal.

The very helpful people in the administrative office: Lorraine Harper, Franca Cianci and Lise Minogue, for making life a lot easier.

People at home: my parents for their quiet support, uncle S. N. for always coming to rescue, and everyone else for being so much fun.

This work was partly supported by the EPPP project, financed by Industry Canada, Alex Parallel Computers, Digital Equipment Canada, IBM Canada and the Centre de recherche informatique de Montréal (CRIM).

In the memory of my grandfather Jagdish Prasad Ghiya

Contents

List of Figures

Chapter 1

Introduction and Related Work

Optimizing and parallelizing compilers rely upon accurate static disambiguation of memory references i.e. determining at compile-time, if two given memory references would always access disjoint memory locations. Unfortunately the presence of aliases in programs makes memory disambiguation a non-trivial issue. An alias arises in a program when there are two or more distinct ways to refer to the same memory location. In the presence of aliasing, two seemingly dissimilar references can access the same memory location. Program constructs that introduce aliases are arrays, pointers [1] and pointer-based dynamic data structures. For example, the array references `a[i+2*j]` and `a[j+2*i]`, the pointer dereferences `*q` and `*p`, and the structure accesses `p->item` and `q->next->item`, can lead to the same memory location.

Over the past twenty years, powerful data dependence analyses have been developed to resolve the problem of array aliases [Ban88, Wol89, ZC90]. These analyses use integer programming techniques to determine if two array subscript expressions can evaluate to the same value. They form the core of present day optimizing/parallelizing compilers. The problem of calculating pointer-induced aliases, termed *pointer analysis*, has so far remained a topic of mostly academic interest. It has not progressed beyond prototype implementations, as it is a much harder problem shown to be undecidable in its generality [Lan92]. However, as languages supporting pointers such as C, C++ and Fortran90 continue to gain popularity, an increasing need is being felt to develop approximate but effective pointer analysis techniques. Further emphasis on this problem comes from application areas which primarily use

[1] Call-by-reference parameters can be considered as a restricted case of pointer usage.

pointer-based dynamic data structures. Important examples include: computational fluid dynamics, computational geometry, computational biology, computer graphics, N-body and circuit simulations.

1.1 Pointer Analysis

To properly understand the pointer analysis problem, we first divide it into two distinct subproblems. The first subproblem focuses on pointers pointing to statically allocated memory objects (typically on the stack). We call them stack-directed pointers. The second subproblem deals with heap-directed pointers, which point to objects dynamically allocated in the heap. A pointer pointing to an object, implies that it contains the memory address of the given object. For example in Figure 1.1, pointer `p` points to the object `x`. Further, pointer variables `p` and `q` are stack-directed, while `r` and `s` are heap-directed. Note that the pointer variables themselves are resident on the stack. Also, a pointer can fall into both categories, if it can possibly point to locations on the stack as well as in the heap.

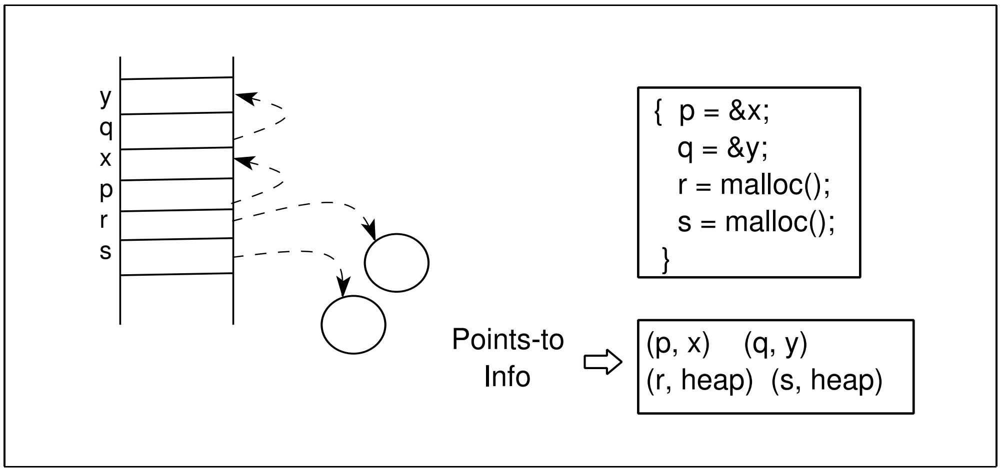


Figure 1.1: The Pointer Classification

Stack-directed pointers exhibit the important property that their targets always possess a name. This is because all data objects allocated on the stack, have compile-time names. Using this property, alias information for such pointers can be conveniently captured in the form of *points-to* pairs. For example in Figure 1.1, we have points-to pairs `(p, x)` and `(q, y)` denoting that pointer variable `p` points to the data object `x` and pointer variable `q` points to the data object `y`. An alias analysis algorithm for stack-allocated data objects, based on the points-to abstraction, has been implemented in the framework of our McCAT (McGill Compiler Architecture Testbed) C compiler [HDE+93, Ema93, EGH94]. The empirical results reported in [Ema93, EGH94] indicate that the points-to information collected is highly precise.

Unfortunately this nice property does not hold for heap-allocated data items. In fact all the objects in the heap are *anonymous*. They can be accessed only through pointer dereferences like `*r` or `r->item` or `a[i]`, where `a` is a heap-directed pointer. One cannot also use a simple naming scheme to name heap objects, as a potentially infinite number of them can be created. Further, objects in the heap are dynamically linked, and more importantly delinked. Hence, there is no natural way of naming even collections of objects (e.g. linked structures). Unlike arrays, both the number of linked structures and the number of objects belonging to a given linked structure, vary dynamically. Thus for a heap-directed pointer, the points-to abstraction only captures the very coarse information that it points to the heap. The points-to pairs `(r, heap)` and `(s, heap)` in Figure 1.1 demonstrate this point. Thus in order to estimate more accurate information about heap-directed pointers, a different approach is required.

This thesis focuses on developing some practical techniques for heap analysis. In the following sections we first give an overview of the problem, discuss existing methods and approaches, and then briefly describe our approach.

1.2 Heap Analysis

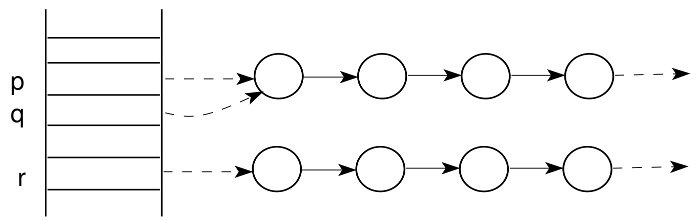
The problem of heap analysis has the following two components:

- **Data Structure Analysis.**
- **Interference Analysis.**

The goal of data structure analysis is to statically estimate the structure of the heap at each program point. A typical data structure analysis should be able to answer the question: "can two heap references at a given *program point*, lead to the same heap location?" This question can also be rephrased as: "are the two references aliased at the given program point?" For example in Figure 1.2(a), at program point S, the heap references p->item and q->item would access the same heap location, while the references p->item and r->item would not. Note that in the light of theoretical results [Lan92], the analysis is not expected to give precise information. It is allowed to err conservatively i.e. two heap references may be reported to access the same location, even when they would not, in any execution of the program.

```
{   p = build_list();
    r = build_list();
    q = p;
S:  p->item = q->item + r->item;
    p = p->next;
    r = r->next;
T:  p->item = q->item + r->item;
}
```

(a) Program



(b) Heap Structure

Figure 1.2: Example Program for Heap Analysis

Interference analysis attempts to answer the same question, albeit in a different context. Here the heap references of concern are typically at two different program points. Two statements interfere, if both statements access a common memory location, and one of them writes to it. Interference analysis for heaps needs to establish a connection between the heap locations accessed at different statements in the program. This is a difficult problem, because seemingly similar references can access different locations. For example, the heap reference p->item in Figure 1.2(a), accesses different locations at statements S and T. It should be noted that the term interference analysis is analogous to other terms used in the literature: conflict analysis and data dependence analysis.

Any sort of interference analysis for heaps, depends on a precise data structure analysis. Alternatively, the information collected by data structure analysis, may be directly supplied by the user, using programmer annotations. Over the last fifteen years, a good deal of work has been done on the different problems of heap analysis. We give an overview in the next section.

1.3 Related Work

Jones and Muchnick [JM81] proposed one of the first approaches to the data structure analysis problem. They analyze LISP-like structures for a simple language without procedures. They abstract the structure of the heap at each program-point, in the form of a set of graphs. Nodes in the graph represent objects in the heap, while edges represent the links between these objects. Nodes bound to variables are labeled by variable names. Nodes which can possibly be shared (have more than one parent) or become part of a cycle, are respectively labeled as shared and cyclic. They use set union as data flow merge operator at join points, which results in a set of graphs at each program point. Since the graphs abstract recursive structures, they can have unbounded number of nodes. To avoid building infinite graphs they use the notion of *k-limiting*, whereby all the nodes in a graph accessible from a variable after traversing k or more links, are coalesced into one summary node. For example, a 2-limited linked list is shown in Figure 1.3.

The goal of their analysis is to optimize storage allocation. The k-limited graphs at all program points are analyzed to classify variables into three categories: (i) variables which cannot access any shared or cyclic nodes at any program point, (ii) variables which can access shared nodes but no cyclic nodes, and (iii) variables which can access cyclic nodes. The heap cells accessible from the first variety of variables can be deallocated as soon as pointers to them are destroyed, as their reference count never exceeds one. Those accessible from the second variety can be reference-counted while the rest need to be garbage-collected.

Although an interesting analysis for improving storage allocation, it does not provide precise enough estimation of the heap structure for program optimization. Due to k-limiting all the information about nodes beyond depth k is lost. The introduction of summary nodes can generate spurious cycles in otherwise acyclic structures. Finally, maintaining a set of graphs at every program point, can prove to be quite expensive.

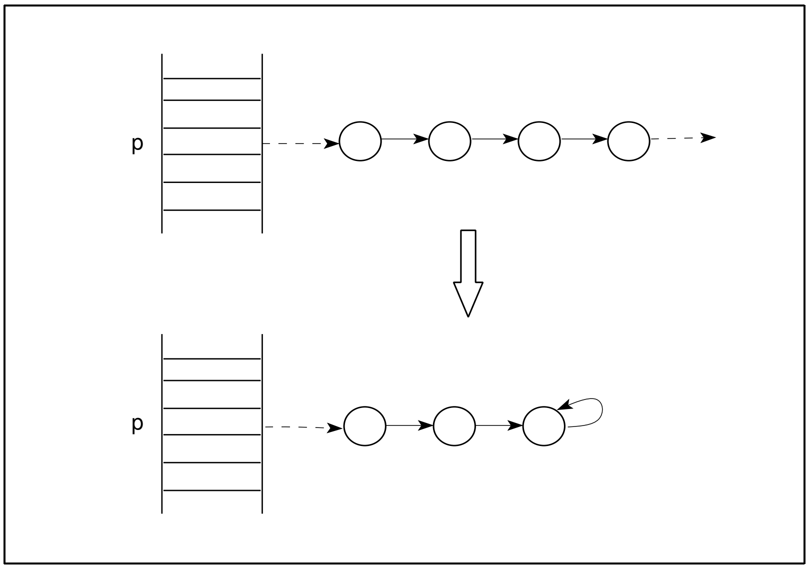


Figure 1.3: An Example of 2-limiting

Jones and Muchnick [JM82] also proposed a flexible framework for analysis of programs with recursive data structures. They designate program points which create or modify recursive structures with tokens. The tokens can be considered as local representations of the data structures at the given program points. They then define a retrieval function to finitely represent the relationships between tokens and data values. The definition of the retrieval function is based on the simulation of program statements, using abstract interpretation [CC77]. The analysis framework is parameterized by the choice of token sets. Thus a wide range of analyses can be expressed in this framework. However, this method has remained mostly of theoretical interest, being expensive in both space and time.

Larus and Hilfinger [LH88] use a variation of k-limited graphs called *alias graphs* for analyzing Lisp programs. Their goal is to detect potential conflicts between heap accesses at different program statements. They label edges in the alias graph by names of corresponding accessors (pointer fields). In addition, they label nodes either by a variable name or by a path expression. Path expressions are regular expressions summarizing possible access paths from a variable to the node being labeled. A newly allocated node is labeled by an aggregate of the labels of the arguments to the allocation function(cons). This proves to be more precise than labeling the node by the program point where it is allocated. Unlike [JM81], they maintain only one alias graph per program point instead of a set of graphs. They define a meet operator that combines two alias graphs into a new alias graph that contains all aliases in either graph. To keep the size of the resulting graph finite, they introduce summary nodes using s-l limiting. In an s-l limited alias graph, no node has more than s outgoing arcs (except the node representing the bottom element), and no node has a label longer than l.

Once an alias graph is computed for each program point, conflict detection is done. A potential conflict exists if access paths at the given statements can lead to a node with the same label in their respective alias graphs. This method works well only for simple data structures like trees and lists. It is rendered expensive by its complex meet, node summary and node labeling operations.

Horwitz, Pfeiffer and Reps [HPR89] presented another variation on k-limited graphs, called storage graphs which abstract the dynamic store. They present a variety of ways to k-limit the storage graphs. The goal of their analysis is detection of dependences between program statements. To this end, they label each node in the storage graph with the program point that last set its contents (unlike Larus and Hilfinger, who use path expressions for labeling nodes). A statement `S` is (flow) dependent on statement `T`, if `S` reads a location whose abstraction in the storage graph is labeled with statement `T`.

Their notion of dependence analysis is more precise than conflict analysis of Larus and Hilfinger, as the latter do not take into account the intervening writes between the statements under consideration. Further, they maintain a set of storage graphs at each program point, unlike a single alias graph [LH88]. This makes their analysis more precise, but also more expensive. They use abstract interpretation [CC77] augmented with a fourth semantics called instrumented semantics, to prove the correctness of their technique. However, it is unclear how effective it would prove to be in practice.

Another approach to abstracting the heap structure in the form of a bounded graph was given by Chase, Wegman and Zadeck [CWZ90]. Their abstraction, called storage shape graph (SSG), contains one node for each variable and one for each allocation site in the program. It is based on the premise that nodes allocated at different places tend to be treated differently, while the ones allocated at a given site would be updated similarly. This abstraction can introduce cycles in otherwise unaliased structures like lists and trees. For example, if all nodes of a list are allocated at the same site, they would be represented by a single summary SSG node with a self-cycle.

To avoid this possibility, they augment their abstraction with reference counts for each node, where nodes with reference counts less than two, would represent trees and lists. Further refinements to the model include: keeping multiple *interesting* instances of an allocation site (i.e. SSG node), enabling *strong updates*, and defining a precise meet operator for join points. An SSG node is considered interesting if it is pointed to by a *deterministic* variable i.e. a variable which does not point to any other node except possibly to `nil` node. A strong update involves replacing edges leaving a node with a new set of edges, giving more precise information. It can only be performed for nodes representing a single heap location. Finally, the meet operator tries to minimize the creation of summary nodes, and only merges nodes representing the same allocation site.

This method would give precise results in some special cases. In general, it can be overly conservative because of one SSG node abstracting several run-time locations. For example, it would give highly imprecise results, if the program uses a single routine for allocating nodes (authors suggest the use of function inlining to overcome this.). Further, the meet operation is fairly complex.

Plevyak, Chien and Karamcheti [PCK94] have extended the model of Chase et al. [CWZ90] to handle regular cyclic structures like doubly linked lists and trees with parent pointers, more precisely. They introduce additional nodes called choice nodes, to represent that two given links coming into a summary node would not exist at the same time. They also annotate summary nodes with identity paths, to indicate which combinations of link fields can create cycles. Presently they do not handle procedure calls. Further, their analysis needs empirical verification, though they give some examples in the paper. The effectiveness of their analysis would become clearer, once they implement it in their *Concert* compiler.

The approaches described so far are termed as *store-based* techniques [Deu92], as they attempt to explicitly abstract the dynamic store in the form of a bounded graph. They basically differ from each other in the way they choose to bound the graph.

Further, nodes in the graph are sometimes labeled to facilitate conflict detection between statements [LH88, HPR89]. Procedure calls are either not handled [JM81, HPR89, PCK94] or are analyzed with different degrees of precision [JM82, LH88, CWZ90]. The restriction of representing several heap locations with one abstract location, forms the main source of imprecision for the *store-based* techniques.

To avoid this trap, Hendren and Nicolau [HN90] took a different approach. They focus on abstracting the properties of data structures being built and manipulated, instead of abstracting each cell in the heap. Their main focus is on identifying data structures with regular properties like trees and dags. The knowledge about the underlying data structures is then used for interference analysis and parallelization. For example, computations on left and right subpieces of a binary tree can be scheduled in parallel.

To collect such information, they perform *path matrix analysis*. A path matrix `P`, is a matrix of *stack-resident* heap-directed pointers, called *handles*. An entry `P[r,s]` in the matrix contains the summary of possible access paths from pointer `r` to pointer `s` in the heap, at the given program point. Access paths consist of link fields, and are represented as restricted regular expressions called *path expressions*. Path relationships between pointers are used to determine when a tree temporarily becomes a dag. Dag nodes are reference-counted to detect when they again become tree-like (i.e. when the reference count becomes less than two). They perform context-sensitive interprocedural analysis and handle recursion precisely, which is important, as recursion is the main tool to build and use recursive data structures. Once the path matrix analysis determines the underlying data structure to be a tree, they perform interference analysis based on this information. This exposes the divide and conquer type of parallelism, induced by recursive traversal of tree-like structures.

Their method is precise and effective for trees and to some extent for dags. However, it cannot handle cyclic structures, which are commonly used in programs, like doubly-linked lists and trees with parent pointers.

All the techniques discussed above, consider the heap analysis problem in isolation from stack analysis. They assume that pointers only point to heap objects, and cannot point to objects on the stack. This assumption is valid for languages like Lisp and Pascal. However, it is not valid for languages like C, which have the address-of (i.e `&a`) operator. Here, one has to provide solutions to both the problems. As discussed above, it is desirable to have separate abstractions for performing stack and heap analyses. However, several schemes have been proposed, which use a unified framework for both the analyses. Each of them depend upon, and propose, a different strategy for naming anonymous heap objects. We discuss them below.

Guarna [Gua88] proposed one of the first approaches to analyze C pointers for dependence detection. He constructs syntax trees to name heap objects, and intersects them to detect dependences. His analysis *assumes* the underlying data structure to be a tree, and is not *safe* otherwise. Further, it does not handle procedure calls.

Landi and Ryder [LR92] collect alias information in the form of pairs of *object names*. An *object name* consists of a variable and a (possibly empty) sequence of dereferences and field accesses. Typical alias pairs are: `(**a, *b)`, `( *(a->next), *(b->next) )`. In the presence of recursive data structures, the number of object names is infinite. To avoid this, they k-limit object names (as opposed to k-limiting data structures [JM81]), where no object name can have more than `k` dereferences. For example, for `k = 1`, `p->l->r` would be represented by `p->l`. They also name heap objects according to the malloc site that allocates them.

Their method effectively resolves stack-based aliases. It is not designed to accurately handle heap-allocated recursive data structures. In some special cases it can help detect completely unaliased data structures (lists and trees) built by a program, but neither empirical nor theoretical evidence is available to draw any general conclusions.

Choi, Burke and Carini [CBC93] also compute aliases of pairs of *access paths*. Their access paths are similar to object names [LR92]. However, they do not use access paths to name heap objects. They use the place (statement) in the program, where an anonymous heap object is created, to name it, as in [CWZ90]. To avoid giving the same name to heap objects created at the same statement, but along different call-chains, they qualify the names with procedure strings. In the presence of recursion, this qualification proves to be of limited use. They mention that they combine this naming scheme with k-limiting to analyze recursive structures. It is not clear from their paper, what type of k-limiting they perform [MLR+93].

Harrison and Ammarguellat [HA93] present a unified framework for parallelizing C, Lisp and Fortran programs, in their Miprac compiler. It uses a very low-level intermediate representation called MIL, which can be considered as a machine-independent assembly language. In MIL, all memory references are made explicit and all loops are converted into tail-recursion. They perform whole program abstract interpretation [CC77] on MIL, and use procedure strings [Har89] to perform interprocedural analysis. Program analysis performed at such a low level becomes less sensitive to program syntax, but also fails to take hints from the program structure. The viability of this approach is hard to determine, until empirical results from their implementation become available.

Deutsch [Deu92, Deu94] calculates aliases in the form of pairs of *symbolic access paths*. This abstraction is particularly suited to recursive data structure analysis. A symbolic access path (SAP) is an access path possibly containing symbolic expressions of the form B^k, where B is a set of access paths called a *basis* and k is a variable. For example, the SAP `X->`$(tl)^i$`->hd`, has its *basis* as tl. This SAP, when parameterized on i, finitely represents an infinite number of access paths from the head of a list to the `hd` fields of its nodes. No imprecision is incurred, as happens with the k-limiting of object names [LR92].

An alias pair in this framework consists of a pair of symbolic access paths qualified by an equation. Thus a *position dependent* alias relationship of the form: "the *i*th element of list X is aliased to the *2i + 1*th element of list Y", would be precisely expressed as `(< X->`$(tl)^i$`, Y->`$(tl)^j$` >, j=2i+1)`. Although a more powerful and expressive framework, it is not clear if it is practical enough to be implemented in a real compiler.

Emami, Ghiya and Hendren [EGH94] proposed the approach of decoupling stack and heap analyses. They focus on analysis of stack-directed pointers, and collect alias information in the form of points-to relationships. A points-to relationship is denoted by a triple `(p, x, d)`, which indicates that pointer `p` definitely (`d = D`) or possibly (`d = P`) points-to the location named `x`. As locations in the heap are anonymous, they are represented by one abstract location called *heap*. All heap-directed pointers are reported to be pointing to this location.

The points-to abstraction provides a more compact representation for calculating aliases than exhaustive alias pairs based on access paths. It also enables simultaneous calculation of both possible and definite relationships. Empirical results reported indicate that this method collects highly precise information for stack-based aliases. At the same time, it builds a framework for conducting a variety of heap analyses such as those presented in this thesis. This approach of decoupling the stack and heap analyses might incur some imprecision, when pointer fields in heap cells point to locations on the stack. However, the authors provide empirical evidence that it does not commonly happen in real programs.

The techniques proposed by [LR92, CBC93, EGH94] handle procedure calls in a context-sensitive manner i.e. the effect of a procedure call is estimated specific to a calling context, and not just summarized for all possible calling contexts. They use different strategies to abstract calling contexts: assumed alias sets [LR92], last call site and source alias sets [CBC93], and invocation graphs [EGH94]. In addition, Emami et al. [EGH94] precisely handle indirect calls through function pointers in C. Deutsch [Deu94] describes how to handle procedure calls in general, and does not propose any particular strategy for interprocedural analysis.

Besides the automatic analysis techniques discussed above, certain language-based approaches have been proposed to get the information from the programmer. A brief discussion follows.

Lucassen and Gifford [LG88] defined a language (FX-87), which incorporates both an *effect* and a *type* system. The effect of a computation must be explicitly associated with a region of memory. The effect system differentiates between totally disjoint linked structures, but fails to distinguish between disjoint subpieces of a data structure.

Klappholz et al. [KKK90] proposed Refined C, which extends C with special partitioning constructs. Run time code is associated with these constructs to check if any interference occurs, which can result in substantial overhead.

Hendren, Hummel and Nicolau [HHN92a, HHN92b] presented a mechanism called ADDS (Abstract Description of Data Structures), to explicitly convey the *dimension* and *direction* properties of a data structure, to the compiler. This involves enriching the type definitions of data structures with some semantic information. For example, consider an ADDS type definition of a doubly linked list:

```
type TwoWayLL [X]
 { int       data;
   TwoWayLL *next is uniquely forward along X;
   TwoWayLL *prev is uniquely backward along X;
 };
```

This description tells the compiler that if the list is traversed using only `next` links, then all the nodes visited are unaliased. They demonstrate that using such information, the compiler can perform useful transformations like software pipelining.

Hummel, Hendren and Nicolau [HHN94] presented a more formal approach, to convey the alias properties of data structures. They propose a language based on regular expressions, which captures alias properties in terms of axioms applicable to the type definition of a data structure. For example, the alias properties of a doubly linked list can be easily expressed as:

$\forall$ `p, p.next <> p.`ϵ
$\forall$ `p, p.prev <> p.`ϵ
$\forall$ `p, p.next.prev = p.`ϵ

Using these aliasing axioms, fairly complicated data structures like sparse matrices and two-dimensional range trees, can be precisely described. The ADDS descriptions can also be translated to aliasing axioms.

They also present a general purpose dependence test for dynamic data structures. To detect dependence between two program statements, they first traverse the program segment to find the relative position of the heap locations accessed by them. This is achieved by determining the possible access paths to these heap locations, with reference to a pointer pointing to a fixed heap location. A theorem prover is then used to determine if the given access paths can lead to a common heap location, on the basis of the aliasing axioms provided for the data structure being traversed.

This technique is general purpose (i.e. is not restricted to data structures of certain types like lists and trees). The initial results provided in the paper are encouraging. It would be interesting to see more detailed experimental results.

Klarlund and Schwartzbach [KS93] also proposed a similar approach called *Graph Types* to describe data structures using regular-like expressions. With graph types, pointer fields are separated into two types, *tree* and *routing* fields. The tree fields must create a spanning tree for the data structure, and the routing fields are defined in terms of the tree fields and the underlying spanning tree. Thus, graph types can only describe data structures with a spanning tree backbone.

A large body of work on analysis of heap-allocated objects, has focused on other problems like reference counting and memory lifetimes [Hud86, ISY88, RM88, **?**, Deu90, WH92].

1.4 Our Approach

Our overall goal was to design and implement a sophisticated pointer analysis framework for the McCAT C compiler. We followed the approach of decoupling stack and heap analyses, instead of solving the two problems using the same abstraction. As already mentioned in section 1.1, an interprocedural points-to analysis, which calculates the points-to relationships of variables on the stack, has already been implemented in the framework of the McCAT compiler [Ema93, EGH94]. The points-to analysis uses one abstract location called *heap* for all heap locations. Any pointer pointing to a heap location is reported to be pointing to *heap*. Thus all heap-directed pointers appear to be aliased after points-to analysis.

Our specific goal was to complement the points-to analysis, by further refining the alias relationships of the heap-directed pointers. We found the heap analysis techniques described in the literature (which we have discussed above), to be quite complex and expensive to be implemented in a real C compiler. We realized that any analysis framework aimed to solve the problem in its generality, would tend to become complex. So, we focused our attention on identifying *interesting* sub-domains of the problem, for which simple and efficient analyses could be developed. Further, we decided to follow the *storeless* analysis approach [Deu92]. Accordingly, we estimate the structure of the heap, by capturing the relationships between stack-resident heap-directed pointers, as opposed to explicitly abstracting each cell in the dynamic store.

1.5 Thesis Contributions

With the strategy adopted in the previous section, we developed two practical heap data structure analyses. These analyses use simple *storeless* abstractions that capture boolean relationships between *stack-resident* heap-directed pointers in the program, computed at each program point. Below, we briefly describe the two analyses and identify their specific application domains:

- **Connection Analysis:** This analysis determines if two heap-directed pointers point to the same linked structure (i.e. they are connected) or to *disjoint regions* in the heap (i.e. they are not connected). It uses a *connection matrix abstraction*, which is a boolean matrix of heap-directed pointers, to collect *connection* information. This information is useful in disambiguating heap accesses to completely disjoint data structures like dynamically allocated arrays and other non-recursively defined structures. Scientific applications written in C typically use these constructs.

- **Shape Analysis:** This analysis focuses on estimating the shape of the structure *accessible* from a given pointer: is it tree-like, dag-like or a general graph containing cycles. It uses four simple abstractions to achieve this goal, which include:

1. *Direction Matrix:* This abstraction approximates the *path existence* relationship between heap-directed pointers, i.e. if there exists an access path in the heap from one pointer to another pointer.
2. *Interference Matrix:* This matrix computes, if a common heap location can be accessed, starting from two given heap-directed pointers. It is computed in conjunction with the direction matrix, and is designed to handle dag-like structures, where two pointers may not have a path to each other, but can still *interfere* i.e. access a common heap location. It forms a superset of the direction matrix.
3. *Shape Attribute:* This attribute is associated with each heap-directed pointer to store the shape of the data structure accessible from the given pointer.
4. *Root Attribute:* It abstracts the following property: if the object pointed to by a given heap-directed pointer forms the root of the data structure (i.e. has no incoming links) or an intermediate node.

The motivation behind shape analysis is to identify tree and list-like structures in programs in a simple and efficient way. This knowledge can then be gainfully exploited for parallelizing programs [Lar89, Hen90], and performing optimizing transformations like loop unrolling [HG92] and software pipelining [HHN92a]. There is a large body of applications which use trees and lists as principal data structures.

It should be noted that these abstractions are practical variations on the path matrix model of Hendren and Nicolau [HN90]. The differences lie in collecting coarser path information for efficiency reasons, and associating additional attributes with each pointer (e.g. shape attribute). Our strategy is to run these analyses in a *hierarchical* fashion. If the points-to analysis reports no heap-directed pointers, no heap analysis needs to be performed. Otherwise, we first run a simple and cheap analysis like connection analysis. If it provides overly conservative results, we proceed to shape analysis. Next, more complex analyses like that of Deutsch [Deu94], or programmer supplied information [HHN92a, HHN94], can be used. Thus, the cost of an expensive analysis is incurred only if the input program requires so.

We have implemented context-sensitive interprocedural analyses for these abstractions in the McCAT C compiler. We have augmented the interprocedural analysis framework used for points-to analysis [EGH94]. Our method precisely handles recursion, indirect calls through function pointers, and variables indirectly accessible through pointers (invisible variables). We have performed experiments on a set of heap-intensive C benchmark programs of medium size. The empirical results indicate that each analysis provides reasonably precise information for its target application domain, and *safe* conservative approximations otherwise, as expected. The analyses run efficiently, as boolean matrices enable fast update and merge operations.

In brief, the main contributions of this thesis include:

- Design of two practical heap data structure analyses, connection analysis and shape analysis, which use simple and efficient *storeless* abstractions, and form part of a *hierarchy* of pointer analyses.
- A context-sensitive interprocedural implementation of these analyses in a real C compiler, handling almost all the complexities of the C language.
- Verification of the effectiveness of these analyses by an empirical study of a set of heap-intensive C benchmark programs of medium size (of upto 5,000 lines).

1.6 Thesis Organization

The rest of this thesis is organized as follows. In chapter 2 we describe the overall setting for the implementation of heap analyses in the McCAT C compiler. In chapters 3 and 4, we provide the analysis rules for connection analysis. The rules for estimating shape information are given in chapter 5. In chapter 6, empirical data is presented to demonstrate the effectiveness of these analyses on real C programs. Finally in chapter 7, we draw conclusions and discuss the scope for future work.

Chapter 2

Setting

In this chapter, we outline the setting in which our analyses have been designed and implemented. We chose C as the language under analysis, as it supports all the interesting and challenging pointer features, and is widely used. We modeled our analyses as practical variations on the path matrix abstraction of Hendren and Nicolau [HN90]. The McCAT C compiler formed the ideal platform for implementing our analyses. The main reasons for this choice were: (1) it provides a simple and structured intermediate representation called SIMPLE, specially designed for efficient pointer analysis, (2) it already has an implementation of the points-to analysis [Ema93], that calculates the points-to relationships of variables on stack, enabling us to focus solely on the analysis of heap-directed pointers, and (3) it provides a framework for general purpose context-sensitive interprocedural analysis, that accurately handles recursion and function pointers [HEGV93, EGH94].

We give a brief overview of the McCAT compiler in the first section. The next section focuses on the SIMPLE intermediate representation. In section 2.3, we briefly discuss the interaction of points-to and heap analyses. The interprocedural analysis framework is described in section 2.4. In the last section we review the path matrix analysis, to put our analysis techniques in proper perspective.

2.1 The McCAT C Compiler

The McCAT C compiler is part of the McGill Compiler Architecture Testbed, being developed to study the interaction between smart compilation techniques and advanced architectural features [HDE+93]. The compiler is built on top of the front-end of the GNU C compiler (version 1.37.1) [Sta90]. The most important goal in its design was to develop appropriate intermediate representations to facilitate implementation of various high-level and low-level analyses and transformations in a simple and straightforward manner. Thus it supports a family of tree-based intermediate representations (IR's), namely FIRST, SIMPLE and LAST.

FIRST is a high-level Abstract Syntax Tree (AST) representation of the entire source program. It is built to separate the front-end processing (e.g. scanning, parsing, and type-checking) from the analysis, optimization and code-generation phases of the compiler. It retains the original format of the source program and its data structures. Analysis at this level can become cumbersome, specially if the program uses complex constructs and the programmer has resorted to various tricks allowed by high-level languages (specially C!).

In order to make the implementation of analyses simple and straightforward, FIRST is transformed to another AST intermediate representation called SIMPLE. As the name implies, SIMPLE breaks down all complex program constructs into a series of simple and regular constructs. It also makes control flow structured and explicit. SIMPLE forms the appropriate program representation for high-level analyses like alias and dependence analysis, and for high-level loop and parallelization transformations. We discuss it in more detail in the next section. A complete description is given in [Sri92].

SIMPLE is further transformed to a lower-level representation called LAST(Lower-level Abstract Syntax Tree). It exposes the memory hierarchy, address calculations, and architectural features like delay slots. This IR is designed for low-level optimizations like register allocation and instruction scheduling, and for code generation. More details on LAST can be found in [Don94].

The overall design of the McCAT compiler is shown in Figure 2.1. Note that the compiler takes as input a set of C files, which are linked by a source level linker. This is necessary in order to have the whole source program available for interprocedural analysis.

2.2 SIMPLE **Intermediate Representation**

The SIMPLE intermediate representation has been specially designed to facilitate accurate pointer analysis for C programs. Its major advantage lies in being simple

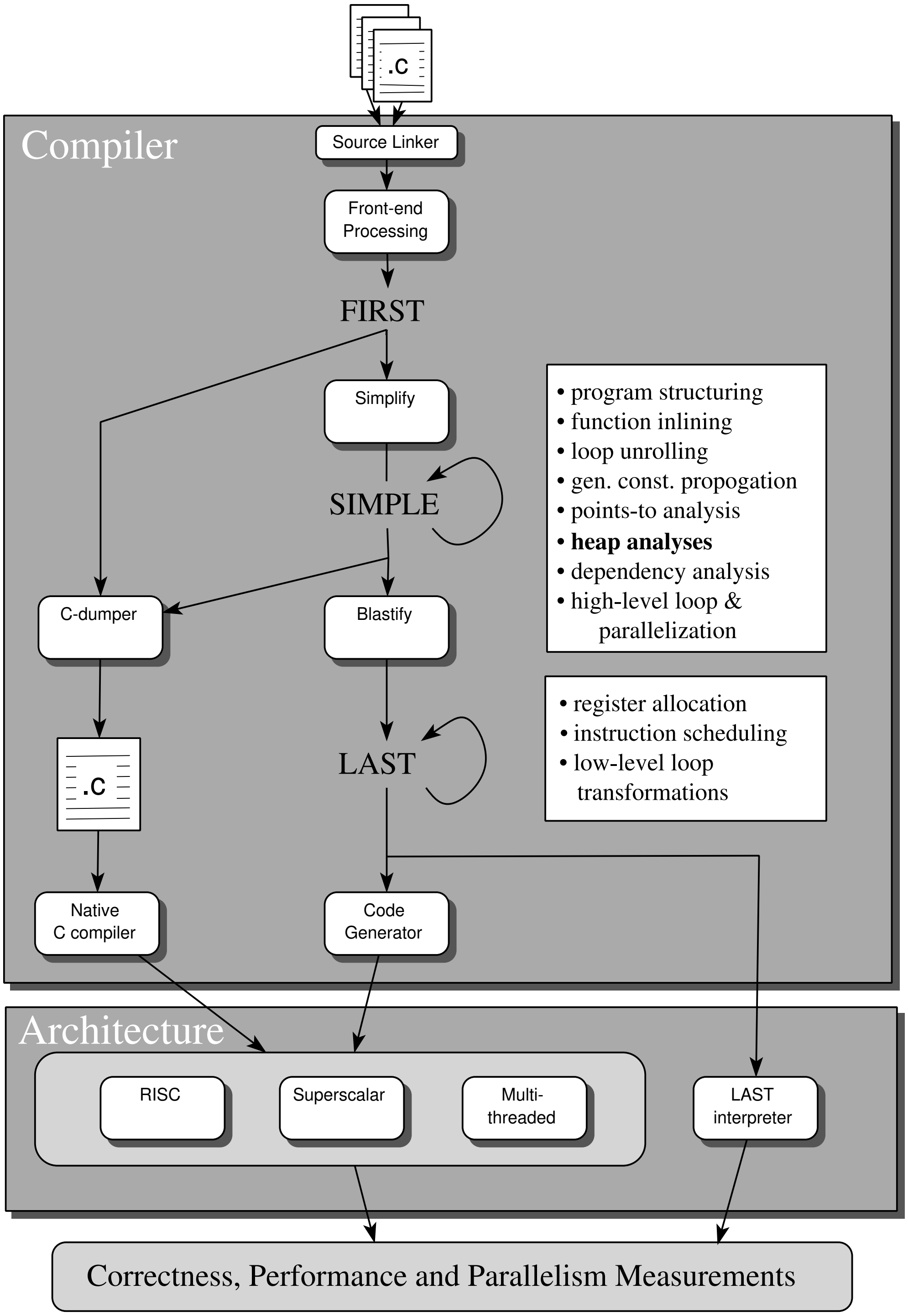


Figure 2.1: The McCAT Compiler

to analyze. This is achieved by performing a number of simplifying transformations. Typical examples include: compiling complex statements into a series of basic statements, breaking down complex variable references into a series of simpler ones, simplifying procedure arguments to either constants or variable names, and moving variable initializations from declarations to statements in the body of the appropriate procedure. We illustrate some simplifying transformations in Figures 2.2 and 2.3. SIMPLE however retains the identity of high-level variable references like array and structure references, and complete type and type-casting information. Most high-level analyses can derive useful hints from this information. For example, array dependence can make use of information like array dimension and array size, while pointer analyses can benefit from type information.

```
                                  temp1 = &a.b;
                                  temp2 = &temp1[3];
f = a.b[3].c.d[2][5].e     =>     temp3 = &(*temp2).c.d
                                  temp4 = &temp3[2][5];
                                  f = (*temp4).e;
```

Figure 2.2: Variable Transformation

```
                                  temp1 = b * c;
                                  temp2 = *d;
a = b * c + (*d) / e;     =>      temp3 = temp2 / e;
                                  a = temp1 + temp3;
```

Figure 2.3: Basic Statements Transformation

SIMPLE restricts the number of basic statements in a program to fifteen. Different types of basic statements in a C program are broken down into one or more of these fifteen statements. In Figure 2.4 we list this set of basic statments. Note that variables '`x`' and '`y`' denote `varnames`, whereas the variables '`a`', '`b`', and '`c`' denote `vals`. The SIMPLE grammar for a `varname` is shown in Figure 2.5.

Development of any new analysis is greatly simplified, as basic analysis rules need to be specified for only fifteen simple statements. Pointer analysis is further facilitated by the fact that only one level of indirection is allowed in any indirect variable reference. Indirect references of multiple level are broken down to adhere to this format, during simplification. Figure 2.6 shows an example of simplifying indirect references. Indirect references augmented with field accesses (i.e. component references) like `(*a).next` and `(*a).next.item` , and indirect array references like `a[i]`

1.	`x`	`=`	`a binop b`	*where* `binop` *is any binary o peration*
2.	`*p`	`=`	`a binop b`	
3.	`x`	`=`	`unop a`	*where* `unop` *is any unary operati on*
4.	`*p`	`=`	`unop a`	
5.	`x`	`=`	`y`	
6.	`*p`	`=`	`y`	
7.	`x`	`=`	`f(args)`	*where* `args` *is a possibly empty list of arguments*
8.	`*p`	`=`	`f(args)`	
9.	`x`	`=`	`(cast)b`	*where* `cast` *is any typecast*
10.	`*p`	`=`	`(cast)b`	
11.	`x`	`=`	`&y`	
12.	`*p`	`=`	`&y`	
13.	`x`	`=`	`*q`	
14.	`*p`	`=`	`*q`	
15.	`f(args)`			

Figure 2.4: List of the 15 Basic SIMPLE Statements. Variables `x` and `y` denote `varname`. Variables `a`, `b`, and `c` denote `val`. Variables `p` and `q` denote `ID`.

```
val : ID
    | CONST

varname : arrayref
        | compref
        | ID

arrayref : ID reflist

reflist : '[' val ']'
        | reflist '[' val ']'

idlist : idlist '.' ID
       | ID

compref : '(' '*' ID ')' '.' idlist
        | idlist
```

Figure 2.5: SIMPLE Grammar for a `varname`

where `a` is a pointer to an array, are represented in the basic statements (Figure 2.4) by variables '`x`' and '`y`' (as can be seen from the grammar shown in Figure 2.5). However, the level of indirection remains restricted to one.

```
**pp = q;          =>        temp1 = *pp;
                             *temp1 = q;
```

Figure 2.6: Simplification of an Indirect Reference

Another important feature of SIMPLE is that it provides a compositional representation of the program, and makes the control flow structured and explicit. The compositional control statement forms supported by SIMPLE are simplified versions of: statement sequences, `for`-loops, `while`-loops, `do`-loops, `switch/case` statements, and `if/else` statements. In addition, `return` statements is supported for exiting a procedure, and `break` and `continue` statements are supported for exiting a loop. Since the unrestricted use of `goto` is not compositional, the compiler provides a structuring phase that eliminates all `goto` statements from a C program [Ero94, EH94].

An important simplification for compositional control constructs involves reducing complex conditional expressions into simple expressions with no side-effects. Figure 2.7 gives an example. Another significant transformation concerns making the control flow in `switch/case` statements structured and explicit. This involves ending each `case` statement with a `break`, `continue` or `return`, and introducing a `default` statement at the end of each `switch` statement. An example transformation is shown in Figure 2.8.

```
                                  temp1 = a+ b;
while (a + b > c)                 while (temp1 > c)
   {                                 {
   ...                 =>            ...
   }                                 temp1 = a + b;
                                     }
```

Figure 2.7: Simplification of a `while`-loop Conditional Expression

With a compositional representation, structured analysis techniques can be used to analyze all control constructs. For example, a while loop can be analyzed by considering only its components: the conditional expression and the body. A structured analysis framework is easier to implement, as only one analysis rule needs to be defined for each of the compound statements such as conditionals and loops. Further, it becomes easier to reason about the fixed-point computations for loop constructs.

```
switch (a)
{
  case 12:
  default:
  case 13:
     { int i;
       stmt1;
  case 14:
       stmt2;
     }
     break;
}
```

$\Rightarrow$

```
switch (a)
{
  int i;
  case 12:
  case 13:
     stmt1;
     stmt2;
     break;
  case 14:
     stmt2;
     break;
  default:
     stmt1;
     stmt2;
     break;
}
```

Figure 2.8: A Switch Statement Transformation

Our heap analyses are performed at the SIMPLE level, using structured analysis techniques. We analyze the program in the source order, as a SIMPLE tree-walk naturally follows this order.

2.3 Points-to Analysis

In C, one can have pointers to locations on the stack as well as in the heap. As mentioned earlier, we follow the strategy of separating the analysis of stack-directed and heap-directed pointers. So we first resolve the points-to (alias) relationships of variables on the stack, using an analysis called *points-to* analysis. This analysis abstracts the set of all accessible stack locations with a finite set of named abstract stack locations. An abstract location may correspond to: (1) the name of a local variable, global variable or a parameter; or (2) a symbolic name that corresponds to locations indirectly accessible through a pointer dereference, when these locations correspond to variables not in the scope of the procedure under analysis; or (3) the symbolic name *heap* that represents all accessible heap locations. Note that symbolic abstract stack locations can represent more than one real stack location. We further elaborate on this in the next section.

Given that each real stack location has a corresponding named abstract stack location, alias information is then captured in the form of definite and possible points-to relationships between abstract stack locations, defined as follows:

Definition 2.3.1 *Abstract stack location* `x` definitely points-to *abstract stack location* `y`*, at a given program point, if* `x` *and* `y` *each represent exactly one real stack location at that program point, and the real stack location corresponding to* `x` *contains the address of the real stack location corresponding to* `y`*. This is denoted by the triple* `(x,y,D)`*.*

Definition 2.3.2 *Abstract stack location* `x` possibly points-to *abstract stack location* `y`*, at a given program point, if it is possible that one of the real stack locations corresponding to* `x` *contains the address of one of the real stack locations corresponding to* `y` *at that program point. This is denoted by the triple* `(x,y,P)`*.*

The complete description of points-to analysis can be found in [Ema93] and an overview in [EGH94]. However, we demonstrate it on an example program, in Figure 2.9. Part (a) of the figure shows the original program, while part (b) shows the simplified program decorated with program-point-specific points-to information. Note that all heap-directed pointers are reported to be possibly pointing to the abstract stack location *heap*. A pictorial representation of the abstract stack at program point D is shown in part (c) of the figure. Solid lines in the figure denote definite relationships while dashed ones represent possible relationships. The abstract stack is implemented using two boolean matrices, which respectively store the definite and possible points-to relationships.

Points-to analysis lays the foundation for performing heap analyses. First, it determines the set of heap-directed pointers in the program. This set consists of pointers which are reported to be possibly pointing to *heap* at some point in the program. For example, for the simplified program in Figure 2.9(b), only pointers `p` and `q` would fall into this set. All heap analyses only need to approximate the relationships between these heap-directed pointers. This helps in reducing the storage requirements for the abstraction being implemented.

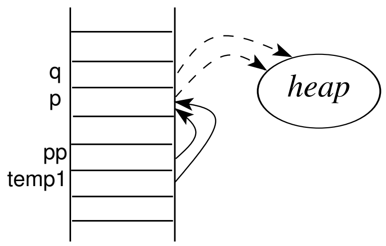
Secondly, the points-to information is used by heap analyses to accurately handle indirect references. For example, consider the analysis of statement `p->next = q` (`(*p).next = q`) in Figure 2.10. To estimate its effect on any heap analysis, we first need to know what locations pointers `p` and `q` can point-to. If both of them point-to *heap*, then this statement links the actual pointed-to locations through the `next` link, as shown in Figure 2.10(a). Similarly, if `q` points-to *heap*, and `p` points-to a stack-resident structure `x`, we would have `x.next` pointing to the same heap location as `q`, as shown in Figure 2.10(b). On the other hand, if `q` points-to a stack location, and `p` points-to the structure `x`, with its `next` field also pointing to a stack

```
main()
{ int *p, *q, **pp;

   p = (int *) malloc();
   q = p;
   pp = &p;
   **pp = 5;
}
```

(a) Original Program

```
main()
{ int *p, *q, *temp1, **pp;

   p = (int *) malloc();
   /* A: { (p,heap,P) } */
   q = p;
   /* B: { (p,heap,P), (q,heap,P) } */
   pp = &p;
   /* C: { (p,heap,P), (q,heap,P)
            (pp,p,D) } */
   temp1 = *pp;
   /* D: { (p,heap,P), (q,heap,P)
            (pp,p,D) (temp1,p,D) } */
   *temp1 = 5;
}
```

(b) Simplified Program with Points-to Information



(c) Abstract Stack Representation

Figure 2.9: An example for Points-to Analysis

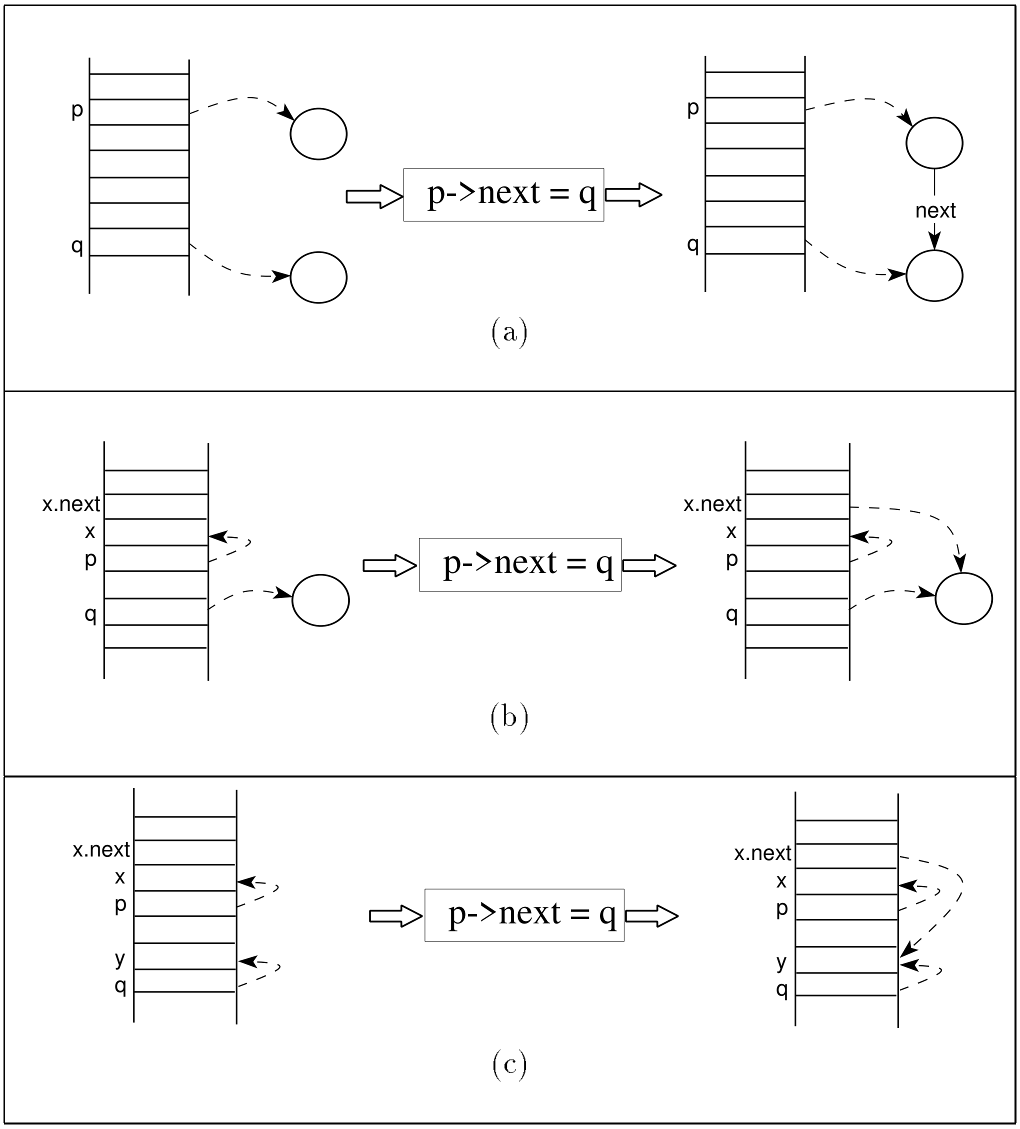


Figure 2.10: Points-to Information and Heap Analyses

location, the statement does not affect heap analysis, as no pointer to a heap location is updated (Figure 2.10(c)). Thus points-to information is of fundamental importance for performing accurate heap analysis. We will further explore the dependence of heap analyses on points-to information, in the following chapters.

2.4 Interprocedural Analysis Framework

Points-to analysis handles procedure calls in a context-sensitive manner i.e. it estimates the effect of a procedure call, within its specific calling context, and not as a summary of all possible calling contexts. To support this analysis strategy, it builds a framework for interprocedural analysis. Other context-sensitive interprocedural analyses like heap analyses, are built on top of this framework. A complete description of the framework can be found in [Ema93, HEGV93, EGH94]. In the following chapters, we specialize this framework for different heap analyses. Below, we briefly describe its salient features, which are of relevance for this purpose.

2.4.1 Representing Calling Contexts

In general, a calling context depends on the *invocation path* followed by the program i.e. the chain of procedure invocations starting from `main` and ending with the procedure call under analysis. Points-to analysis builds an *invocation graph*, where all invocation paths are **explicitly** represented. In the absence of recursion, the invocation graph is constructed by a simple depth-first traversal of the invocation structure of the program. Consider for example, the invocation graph for the program in Figure 2.11(a). An important characteristic of the invocation graph is that each procedure invocation chain is represented by a unique path in it, and vice versa. Using the invocation graph one can distinguish not only calls from two different call-sites of a procedure (calls to f() in Figure 2.11(a)), but one can also distinguish two different invocations of a procedure from the same call-site when reached along different invocation chains (call to f() in Figure 2.11(b)).

In the presence of recursion the exact invocation structure of the program is not known statically, and one must approximate all possible unrollings of the recursion. Figure 2.12 illustrates a program with simple recursion and the set of all possible invocation unrollings for this program, and our invocation graph that is used to approximate all possible unrollings. To build the graph in the case of recursion one terminates the depth-first traversal each time a function name is the same as that of one of the ancestors on the call chain from `main`. The leaf node (representing the repeated function name) is labeled as an approximate node, and its matching ancestor node is labeled as a recursive node. The pairings of these nodes are indicated with a special back-edge from the approximate node to the recursive node. It should be noted that these back-edges are used only to match the approximate node with its appropriate recursive node, and they are therefore quite different from the other tree edges which correspond to procedure calls. This scheme is completely general. Consider, for example, the invocation graph for a program with mutual recursion displayed in Figure 2.13.

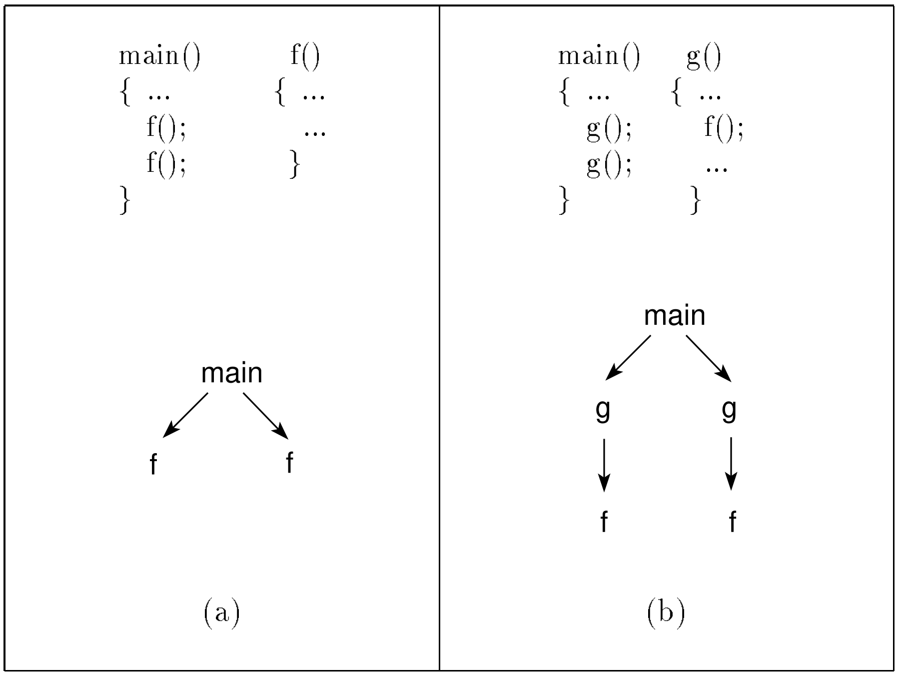


Figure 2.11: Invocation Contexts

The approach of explicitly building the invocation graph has the following advantages: (1) it cleanly separates the abstraction for any interprocedural analysis from the abstraction required to encode the calling context, (2) it allows one to deposit information computed from one analysis that can be useful for the next analysis, (3) it provides a place to store (memoize) IN/OUT pairs previously computed to summarize the effect of the function call (so that extra computation can be avoided at analysis time), and (4) it provides a simple framework for implementing simple compositional fixed-point computations for recursion.

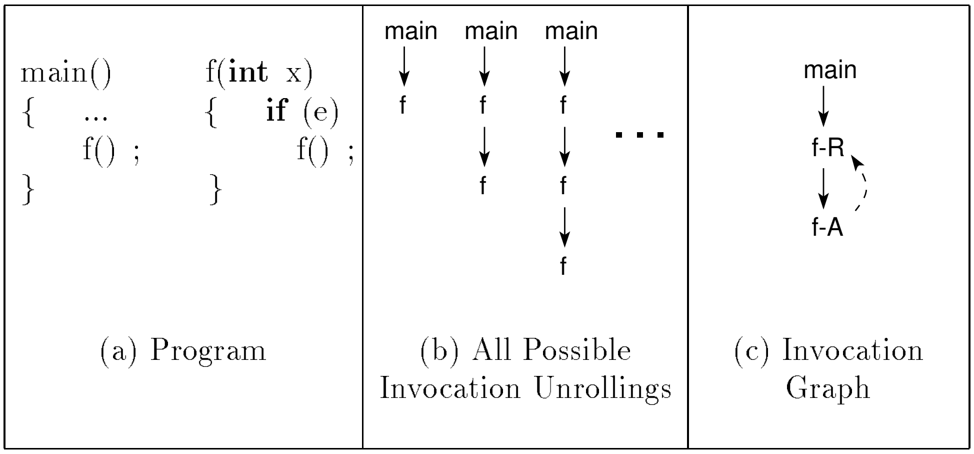


Figure 2.12: Invocation Graph for Recursion

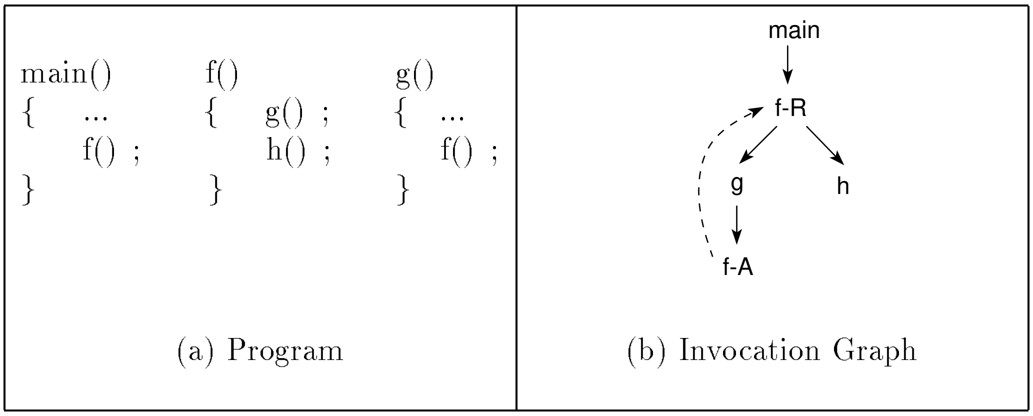


Figure 2.13: Invocation Graph for Mutual Recursion

2.4.2 Map Information in the Invocation Graph

In the presence of procedure calls, an indirect reference in a procedure can refer to variables that are outside its scope (henceforth termed as *invisible* variables). This can happen, for example, when the address of a local variable is passed as a parameter, or when a global pointer points-to local variables of the caller. As each *accessible* real stack location needs to be represented by a *named* abstract stack location, points-to analysis generates special symbolic names to represent such invisible variables. A symbolic name is generated for each possible level of indirection of formal parameters and global pointers. Next each invisible variable is *mapped* to a unique symbolic name.

For example in Figure 2.14, for the formal parameter `ppu` with type `int**`, symbolic names `1_ppu` and `2_ppu` with types `int*` and `int` are generated. Now, since the address of the local variable `pa` (invisible to `ind_swap`) is passed to `ppu`, the symbolic name `1_ppu` is used to represent `pa` in the procedure `ind_swap`. Similarly, as the invisible variable `a` is accessible through the indirect reference `**ppu`, it gets represented by the symbolic name `2_ppu`. This association of invisible variables with symbolic names is recorded in the invocation graph nodes as *map* information. Figure 2.14(b) shows the map information for various procedure calls. Complete details of the mapping process can be found in [Ema93].

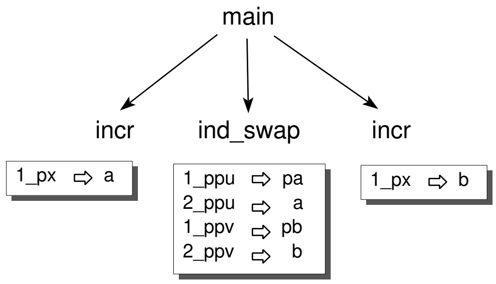
The map information is context-sensitive as can be seen from the different mappings for the two calls to procedure `incr` in Figure 2.14(b). The symbolic names are independent of the context. Points-to analysis and other interprocedural analyses use the symbolic names in a context-free manner when analyzing a procedure. On returning from a procedure, they are *unmapped* to appropriate variables based on the map information recorded in the invocation graph, for the given calling context.

2.4.3 Resolving Function Pointers

In C, pointers may not only point to memory locations, but also to functions. This means that the complete invocation graph cannot be built by a simple textual pass over the program. Thus the points-to analysis must complete the invocation graph by resolving which functions are invocable from each indirect function pointer call. One might ask why the completion of the invocation graph must proceed at the same time as the points-to analysis. Consider that the complete invocation graph cannot be built before points-to analysis because the meaning of an indirect call `(*pf)()` is determined by examining the objects that `pf` may point-to. However, points-to

```
void ind_swap(int **ppu, int **ppv)
{ int  *ptemp;
  ptemp = *ppu;
  *ppu = *ppv;
  *ppv = ptemp;
}

void incr(int *px)
{ int temp_0;
  temp_0 = *px;
  *px = temp_0 + 1;
}

int main()
{ int a; int b; int *pa; int *pb;
  int **temp_2; int **temp_1;
  a = 1; b = 10;
  pa = &a; pb = &b;
  incr(pa);
  temp_1 = &pa; temp_2 = &pb;
  ind_swap(temp_1,temp_2);
  incr(pa);
}
```

(a) Example Program



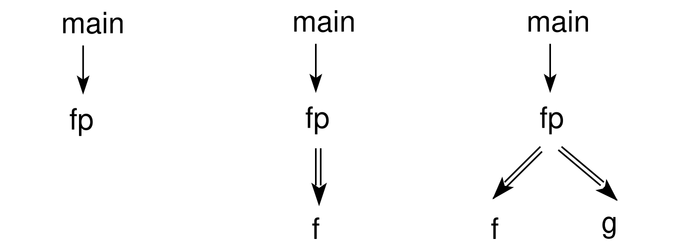
(b) Invocation Graph and Map Information

Figure 2.14: Map Information

analysis itself is a context-sensitive interprocedural analysis that needs the invocation graph. Thus the two approximations must be calculated at the same time.

```
int (*fp)() ;
main()                    int f()              int g()
{   ...                   {   ...              { ... }
    fp = f ;                  fp = g ;
    while (e)                 ...
    {   ...               }
        fp() ;
        ...
    }
}
```

(a)



(b) (c) (d)

Figure 2.15: Invocation Graph for Function Pointers

To explain how points-to analysis completes the invocation graph, we give an example. Consider the program in Figure 2.15(a), with an indirect call `fp()` inside the while loop. The (incomplete) invocation graph of the program before points-to analysis is given in Figure 2.15(b). When points-to analysis encounters the indirect call `fp()`, during the first iteration of the while-loop fixed-point computation, it finds the current points-to set of `fp` to be `(f)`. The invocation graph is updated according to this information, as shown In Figure 2.15(c) and the function `f` is analyzed in the current calling context. During the second iteration of the fixed-point computation, the points-to set of `fp` becomes `(f,g)`, and the invocation graph again gets updated as shown in Figure 2.15(d). Future iterations do not modify the points-to set of `fp`, and thus the points-to analysis constructs the complete invocation graph for other interprocedural analyses. The complete algorithm for resolving function pointers can be found in [Ghi92, EGH94].

Other interprocedural analyses can measure the effect of a function pointer call, by merging the outputs obtained by individually analyzing in the current calling context, all the functions represented by its children nodes in the invocation graph.

2.5 Path Matrix Analysis

Our abstractions for heap analysis are variations on the path matrix model of Hendren and Nicolau [HN90]. The path matrix approach captures the structure of the heap using an abstraction, orthogonal to the k-limited graphs. It essentially exploits the fact, that though there are potentially infinite number of objects in the heap, they are always accessed using access-paths which *originate* from *stack-resident heap-directed* pointers. Figure 2.16 provides an illustration of this observation. It can be easily noted, that access to any node of the data structure built in the heap, has to originate either from pointer variable `p` or pointer variable `q`. They term such *heap-directed* pointers as *handles*, as these are the pointers the programmer has handle on. Since *handles* are themselves resident on the stack, they are not many in number, and are relatively inexpensive to reason about.

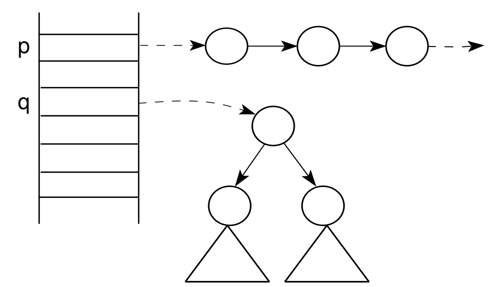
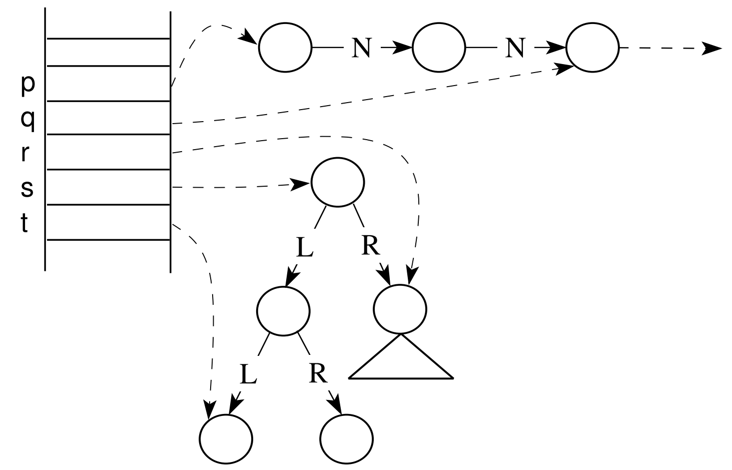


Figure 2.16: Identifying the *handles*

Based on the above observations, the abstraction developed by Hendren and Nicolau, is a matrix of *handles* `P`, where an entry `P[p,q]`, contains the access path inside the heap, from handle `p` to handle `q`. The access paths are expressed in the form of restricted regular expressions. Figure 2.17 presents an example of how the heap structure is captured using the path matrix. An empty entry, say `P[r,q]` indicates that the heap object pointed to by `r` cannot be reached by using an access path originating from pointer `q` and vice versa. The symbol `S` in an entry denotes that the two pointer variables point to the same heap object. The more complicated expressions, represent the access paths using the link fields.



(a) Heap Structure

	p	q	r	s	t
p	S	N^2			
q		S			
r			S		
s			R	S	L^2
t					S

(b) Path Matrix Information

Figure 2.17: An example Path Matrix

However, the path matrix analysis assumes and verifies that the underlying data structure being created and manipulated by the program is a tree. The path matrix information is used to distinguish between pointers accessing disjoint subpieces of the

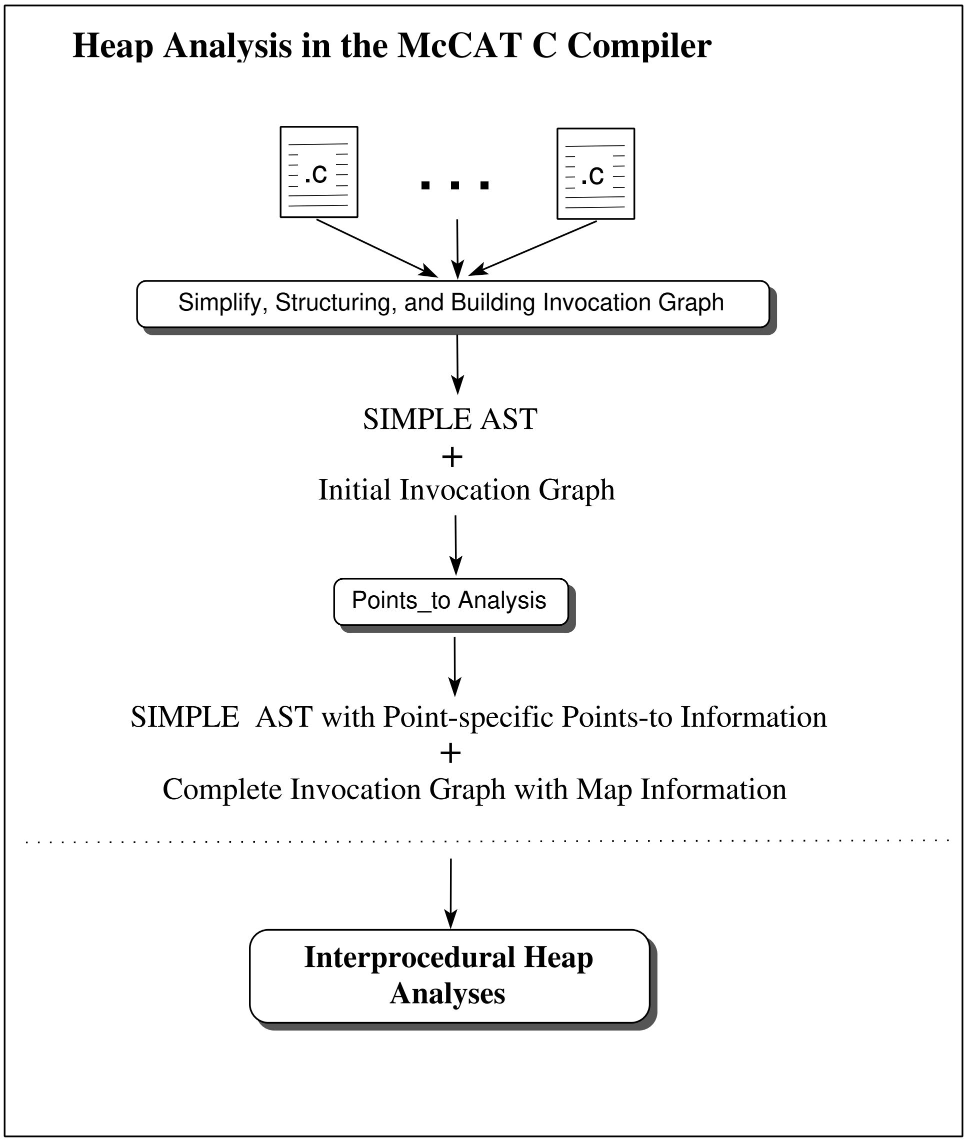


Figure 2.18: The Overall Setting For Heap Analyses

tree, and to detect the creation and elimination of temporary DAG nodes during tree updates. It cannot handle cyclic structures. Further, the encoding of precise path relationships in the form of path expressions, can prove to be potentially expensive information to store.

In the light of these problems, we have adapted the model for more practical and general purpose heap analysis (without any assumptions about the underlying data structures). We follow the paradigm of abstracting the heap structure, by capturing the relationships only between the *handles*. However, we capture coarser path relationships, which can be stored as boolean matrices. This enables faster data flow merge operation and substantially reduces the storage requirements for the analysis, while the analyses still gather *useful* information. Further we also abstract simple attributes like *shape* and *root* attributes which increase the effectiveness of our analyses. Based on this philosophy, we present a hierarchy of practical abstractions for heap analysis in the following chapters.

The heap analyses are implemented on the structured tree-based SIMPLE intermediate representation. Points-to analysis builds a complete framework to perform them efficiently and accurately in an interprocedural fashion. The overall implementation setting is shown in Figure 2.18, which can be briefly described as follows:

First, all the `.c` files for the given program are fed to the source level linker, which generates the FIRST AST for the entire program. Subsequently this AST is *simplified* and structured (i.e. `goto` statements are eliminated). The invocation graph is then constructed by identifying the functions called by each function in the program. Next, points-to analysis is performed, which calculates possible pointer targets at each program point for stack-directed pointers, and also resolves indirect calls through function pointers. Finally, various interprocedural heap analyses are conducted to estimate the relationships of heap-directed pointers.

Chapter 3

Connection Analysis

In this chapter, we describe the connection matrix abstraction, and the basic connection analysis rules associated with it. It forms the first step of our *hierarchical* approach to heap analysis. It is a simple *storeless* abstraction designed to disambiguate heap accesses at a coarse level, but in a highly efficient and cost-effective manner. We introduce and motivate this abstraction in section 3.1. In the next section we identify eight basic statements that can affect the relationships of heap-directed pointers. We then define analysis rules for these statements to clearly illustrate the basic principles of connection analysis. Using these rules as the foundation, the complete framework for connection analysis of C programs at the SIMPLE intermediate representation, is developed in chapter 4.

3.1 The Abstraction

A connection matrix C is a boolean matrix of relationships between heap-directed pointers which captures simple connectivity of heap objects. A *heap object* is defined as a memory object allocated in the heap memory, representing an instance of a valid type definition (basic or user-defined) in the program. The connection matrix abstraction is designed to disambiguate heap accesses at the data structure level. The term *data structure* in this context represents a connected region in the heap i.e. if the heap is viewed as an undirected graph with heap objects as nodes and links between them as edges, each connected component forms a separate data structure. Given any two data structures, they would not have a common heap object belonging to them. For example in Figure 3.1, the heap consists of two data structures: one pointed to by pointers `p` and `q` and the other pointed to by pointers `r`, `s` and `t`. Note that we cannot give names to these data structures. We can only refer to them as being pointed to by a given set of pointers.

With the above definitions, given any two heap-directed pointers say `p` and `q`, connection matrix abstracts the following *program-point-specific* relationships:

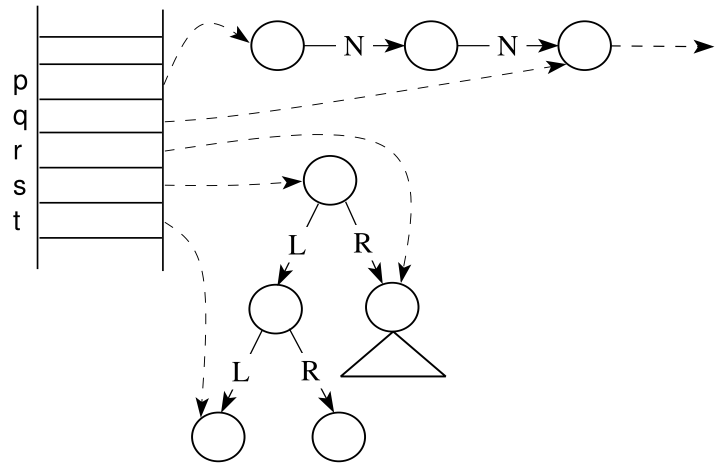
- C[p,q] = 1 : Pointers `p` and `q` *possibly* point to heap objects belonging to the same data structure. In our terminology, pointers `p` and `q` are considered to be *connected*, or to have a *connection* relationship.
- C[p,q] = 0 : The heap objects pointed to by pointers `p` and `q` *definitely* belong to different data structures. In other words, pointers `p` and `q` are *not* connected.

The useful information is the negative information. If pointers `p` and `q` are not connected, then heap accesses originating from them will always lead to disjoint heap locations, and thus not interfere. It is *safe* to report two heap-directed pointers to be connected, when they are not. However, if they can point to the same data structure, they should always be reported to be connected.

We illustrate the abstraction in Figure 3.1. Part (a) shows the structure of heap at a program point, while part (b) shows its abstraction as a connection matrix. In Figure 2.17(b) we have shown the path matrix abstraction for the same heap structure. The path matrix entries are path expressions, while connection matrix entries are simply zeros or ones. The zero in the entry C[p,r] indicates that pointers `p` and `r` point to disjoint data structures in the heap. The one in the entry C[s,r] indicates that `s` and `r` point to objects belonging to the same data structure. Note that the entry C[r,t] is set to one, despite the fact that pointers `r` and `t` point to disjoint subpieces of the same data structure. This is because connection matrix is designed to disambiguate heap accesses at the data structure level (for efficiency reasons). More sophisticated abstractions, which can distinguish between subpieces of a data structure itself, will be presented in the following chapters.

Following are some other important characteristics of the connection matrix abstraction:

- It abstracts relationships only between *stack-resident* heap-directed pointers. As all heap accesses originate from these pointers, their relationships effectively capture the structure of the heap. For example in Figure 3.1(b), the information that pointers `p` and `s` point to disjoint data structures also simultaneously implies that pointers `p->N` and `s->L` point to disjoint structures.



(a) Heap Structure

	p	q	r	s	t
p	1	1	0	0	0
q	1	1	0	0	0
r	0	0	1	1	1
s	0	0	1	1	1
t	0	0	1	1	1

(b) Connection Matrix Abstraction

Figure 3.1: An example Connection Matrix

- For each function in the program, the connection matrix abstracts relationships between all stack-resident pointers which can be heap-directed at some point in the program and are directly or indirectly (through an indirect reference) accessible from the function. Names are naturally available from the program, for directly accessible pointers. For indirectly accessible pointers, special symbolic names are generated by points-to analysis as explained in section 2.4.2. These names are reused by connection analysis. To know which pointers ever point to heap, the existing points-to information is used.

- If a pointer, say `p`, does not point to a heap location at a given program point, the connection matrix entry C[p,p] is set to zero at that program point. In this case the pointer points to `NULL` or to a stack location.

- The connection matrix relationship is symmetric i.e. for any two heap-directed pointers say `p` and `q`, we always have C[p,q] = C[q,p]. The connection relationships shown in Figure 3.1(b) illustrate this property. It is used in the actual implementation to reduce the storage requirement by half.

The connection matrix abstraction is targeted towards programs that allocate a number of disjoint data structures in the heap. Scientific applications written in C typically exhibit this feature, as they use a number of disjoint dynamically allocated arrays. We will present some empirical data in chapter 6, to demonstrate the effectiveness of this abstraction for its intended domain of applications. We now describe the basic analysis rules to compute connection matrix information.

3.2 Basic Heap Statements

Hendren and Nicolau [HN90] had identified six basic statements of a simple imperative language SIL, that access or modify heap data structures. We have added two more statements to this list to cover pointer arithmetic and use of address-operator allowed in the C-language. The complete list is given in Figure 3.2. Variables `p` and `q` and the field `f` are of pointer type, variable `k` is of integer type, and `op` denotes the `+` and `-` operations. We first give the analysis rules for these eight *basic heap* statements, *with the restriction that pointers can only point to heap objects*. These rules are simple to describe and clearly illustrate the basic principles of connection analysis. Analysis rules for the basic SIMPLE statements will then be constructed from these basic rules in chapter 4. There we will take into account the effect of stack-based points-to relationships on estimating heap relationships.

```
1. p = malloc();
2. p = q;
3. p = q->f;
4. p = &(q->f);
5. p = q op k;
6. p = NULL;
7. p->f = q;
8. p->f = NULL;
```

Figure 3.2: Basic Heap Statements

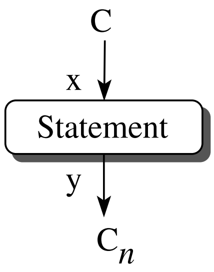
3.2.1 Analysis Rules for Basic Heap Statements

The overall structure of the analysis is shown in Figure 3.3(a). We have the connection matrix C at program point `x` before the given statment, and we wish to compute the connection matrix C_n at program point `y`. To this end, we define an analysis rule for each of the eight statements shown in Figure 3.2. Each rule will compute the following sets of relationships:

- kill_set : Set of connection relationships killed by the given statement i.e. the set of relationships which were valid before the statment (program point `x`), but are not valid after processing it (program point `y`). The entries corresponding to these relationships should be set to zero in the connection matrix C_n.
- gen_set: Set of connection relationships generated by the given statement. The entries corresponding to these relationships should be set to one in the new matrix C_n.

Let H be the set of pointers whose relationships are abstracted by the connection matrix C. Let `p`, `q`, `r` and `s` represent pointers in this set. Assume that pointers can only point to heap objects or to `NULL` (as already discussed). Further assume that updating an entry C[p,q] also implies identically updating the entry C[q,p]. This assumption is required due to the symmetric nature of connection relationships.

The new connection matrix C_n is computed as shown in Figure 3.3(b). First, the old connection matrix C is copied over to C_n. Next, the entries in the kill_set are set to zero in the matrix C_n. Finally, the entries in the gen_set are set to one in the matrix C_n, to get the complete new connection matrix.



(a)

Build the new Connection Matrix
$\forall$ r,s $\in$ H, C_n[r,s] = C[r,s]

Delete killed relationships
$\forall$ entries C[r,s] $\in$ kill_set, C_n[r,s] = 0

Add generated relationships
$\forall$ entries C[r,s] $\in$ gen_set, C_n[r,s] = 1

(b)

Figure 3.3: Computing Connection Matrix C_n from C

We now present the analysis rules for the eight basic statements shown in Figure 3.2. For each statement, we give the rules for computing their kill and gen sets. The new connection matrix can then be computed as shown in Figure 3.3(b).

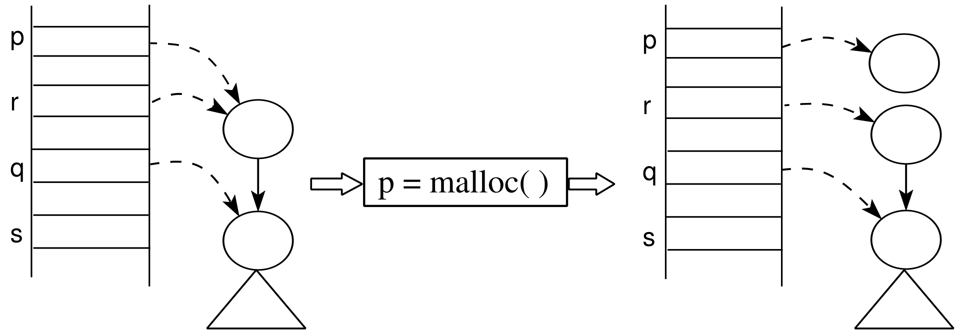
`p = malloc()` : Pointer `p` points to a newly allocated heap object. All the existing connection relationships of `p` get killed. Also as no other pointer can point to this object, `p` will only have connection relationship with itself. So we get the following rule:

$$\text{kill\_set} = \{ \ C[p,s] \mid s \in H \wedge C[p,s] \ \}$$
$$\text{gen\_set} = \{ \ C[p,p] \ \}$$

An example is shown in Figure 3.4 to illustrate this analysis rule.

Basic heap statements 2 through 5 in Figure 3.2 (`p = q`, `p = q->f`, `p = &(q->f)` and `p = NULL`), have a common attribute: all of them update the stack-resident pointer `p`, and make it point to a new data structure. They do not modify the structure of the heap itself. Their effect on connection matrix information can be summarized using a general rule, as discussed below.

`p = q` : Pointer `p` now points to the same heap object as `q`, and hence to the same data structure as `q`. All the existing relationships of `p` get killed, and `p` gets connected to all pointers connected to `q`. If `q` is presently heap-directed (C[q,q] = 1), then `p`



(a) Heap Structure

kill_set = { C[p,p], C[p,r], C[p,q] }
gen_set = { C[p,p] }

(b) Kill and Gen Sets

	p	q	r	s
p	1	1	1	0
q	1	1	1	0
r	1	1	1	0
s	0	0	0	0

(c) Connection Matrix C

	p	q	r	s
p	1	0	0	0
q	0	1	1	0
r	0	1	1	0
s	0	0	0	0

(d) Connection Matrix C_n

Figure 3.4: Analyzing Basic Heap Statement `p = malloc()`

would also be heap-directed after the statement. So the entry C[p,p] is added to the gen_set, if we have C[q,q] = 1. We present the overall rule for this statement below and illustrate it in Figure 3.5.

$$\begin{aligned}&\text{kill\_set} = \{\ C[p,s] \mid s \in H \wedge C[p,s]\ \}\\&\text{gen\_set} = \{\ C[p,s] \mid s \in H \wedge C[q,s]\ \} \cup \{\ C[p,p] \mid C[q,q]\ \}\end{aligned}$$

Note that if `q` presently points to `NULL`, `p` should also point to `NULL` after the statement. In this case all entries C[q,s] will be zero, resulting in an empty gen_set. Consequently all entries C_n[p,s] will also be zero after the statement, indicating `p` to be pointing to `NULL`, as desired. Similarly if `q` happens to be pointer `p` itself, resulting in the statement `p = p`, the gen and kill sets would be identical. In this case the connection matrix would remain unchanged, as required. Thus the above rule is general enough to take into account various special cases.

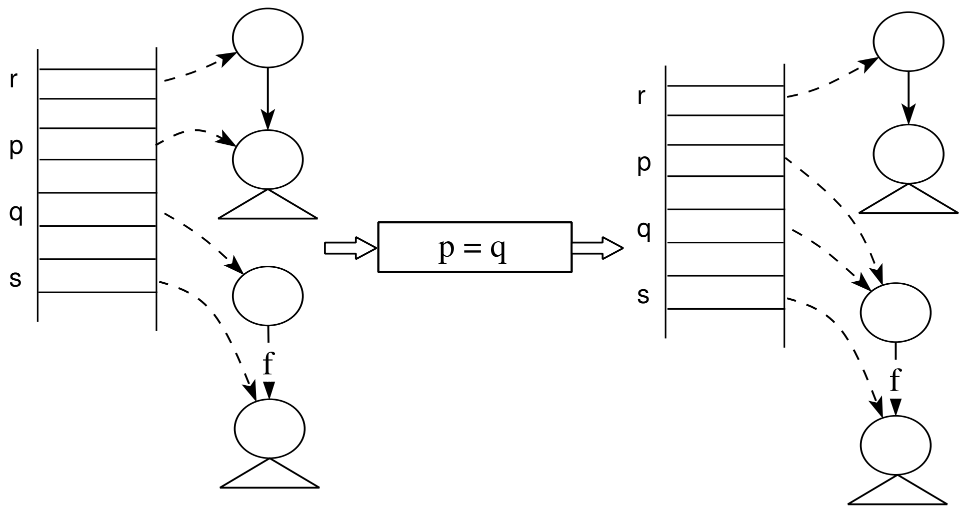
`p = q->f` : Pointer `p` now points to the heap object connected to the object pointed to by `q` through the pointer field `f`. Thus it points to the same data structure as `q`, even if not to the same heap object as `q`. So the analysis rule for this statement is same as that for the statement `p = q`. The effect of this statement on connection relationships is demonstrated in Figure 3.6. The initial heap structure for this example is same as in the example in Figure 3.5. It can be noticed that the kill and gen sets, and the output matrix C_n, are identical for the two examples.

This rule incurs some imprecision, when the pointer `q->f` points to `NULL`. In this case, pointer `p` also points to NULL after the statement. However we would report it be pointing to the same data structure as `q`. This information is *safe* but less precise. This happens because we cannot determine if `q->f` presently points to `NULL`, and not to a heap object. In other words, `q->f` is a heap-resident pointer, while connection matrix only abstracts the relationships of stack-resident pointers.

If pointers `p` and `q` are not distinct, the statement can be of the form `p = p->f`. The rule for this case is same as for the statment `p = p`, which does not change any connection relationships, as required.

`p = &(q->f)` : Pointer `p` now points to the field `f` of the heap object pointed to by `q`, as shown in Figure 3.7. For purpose of our analysis we consider a pointer pointing to a specific field of a heap object, to be pointing to the object itself. Thus, this statement is equivalent to the statement `p = q` for connection analysis.

`p = q op k` : This statement represents pointer arithmetic. After the arithmetic operation, `q` continues to point to the same heap-object, though at a different offset,



(a) Heap Structure

kill_set = { C[p,p], C[p,r] }
gen_set = { C[p,p], C[p,q], C[p,s] }

(b) Kill and Gen Sets

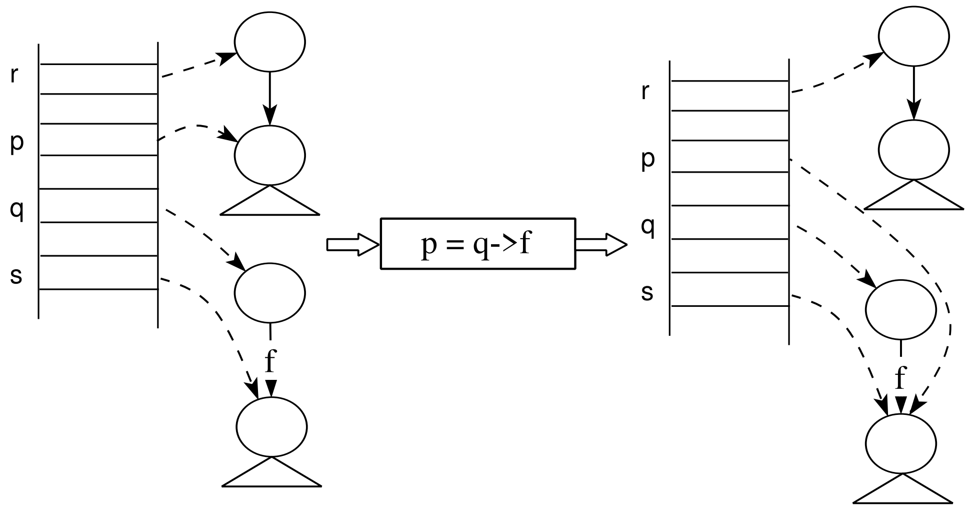
	p	q	r	s
p	1	0	1	0
q	0	1	0	1
r	1	0	1	0
s	0	1	0	1

(c) Connection Matrix C

	p	q	r	s
p	1	1	0	1
q	1	1	0	1
r	0	0	1	0
s	1	1	0	1

(d) Connection Matrix C_n

Figure 3.5: Analyzing Basic Heap Statement `p = q`



(a) Heap Structure

kill_set = { C[p,p], C[p,r] }
gen_set = { C[p,p], C[p,q], C[p,s] }

(b) Kill and Gen Sets

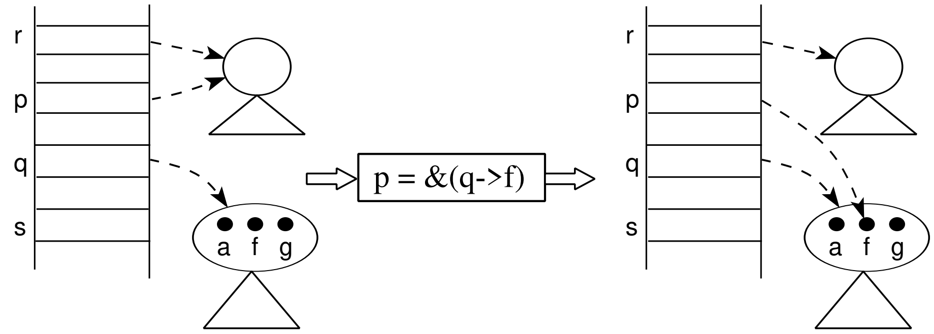
	p	q	r	s
p	1	0	1	0
q	0	1	0	1
r	1	0	1	0
s	0	1	0	1

(c) Connection Matrix C

	p	q	r	s
p	1	1	0	1
q	1	1	0	1
r	0	0	1	0
s	1	1	0	1

(d) Connection Matrix C_n

Figure 3.6: Analyzing Basic Heap Statement `p = q->f`



(a) Heap Structure

kill_set = { C[p,p], C[p,r] }
gen_set = { C[p,p], C[p,q] }

(b) Kill and Gen Sets

	p	q	r	s
p	1	0	1	0
q	0	1	0	0
r	1	0	1	0
s	0	0	0	0

(c) Connection Matrix C

	p	q	r	s
p	1	1	0	0
q	1	1	0	0
r	0	0	1	0
s	0	0	0	0

(d) Connection Matrix C_n

Figure 3.7: Analyzing Basic Heap Statement `p = &(q->f)`

as shown in Figure 3.8. We assume that a heap-directed pointer does not cross the boundary of the heap object, when pointer arithmetic is performed on it. Otherwise, it can potentially point to memory not allocated by the program, and cause an execution error on being dereferenced. With this assumption about pointer arithmetic, this statement is equivalent to the statement `p = q` for connection analysis.

`p = NULL` : Pointer `p` now does not point to any heap object allocated by the program, as shown in Figure 3.9. It does not have any connection relationship with any pointer, including itself. Thus the effect of this statement is to simply kill all the relationships of `p`, as presented below:

$$\text{kill\_set} = \{\ C[p,s] \mid s \in H \wedge C[p,s]\ \}$$
$$\text{gen\_set} = \{\}$$

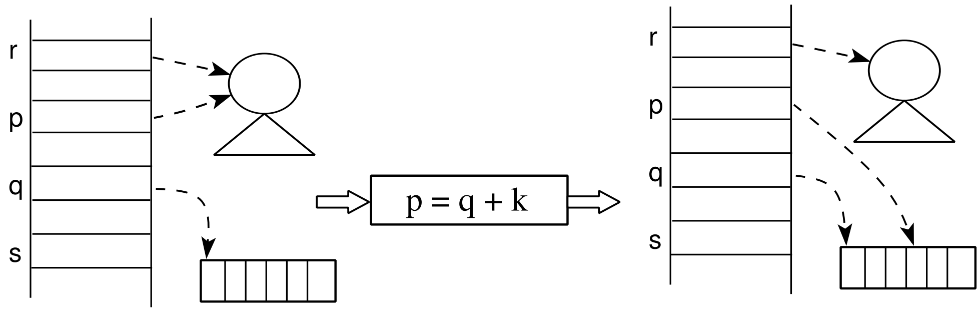
Thus after this statement we have C[p,p] = 0, indicating that `p` presently points to NULL.

The statements discussed so far update a stack-resident heap-directed pointer. The following two statements update a pointer field residing in a heap object, and hence modify the structure of the heap itself.

`p->f = NULL` : This statement sets the field `f` to `NULL`. Consequently the subpiece pointed to by the pointer `p` gets disconnected from the remaining data structure. For example in Figure 3.10, after the statement `p->f = NULL`, pointer `p` does not have connection relationship with pointers `r`, `q` and `s`. However, to obtain this kill information we need to know the following:

- Does setting the field `f` to NULL, really disconnect a subpiece from the data structure? It is possible that the data structure still remains connected due to other links. For example in Figure 3.10, if pointers `p` and `r` are also connected through a `g` link, the subpiece pointed to by `r` would not get disconnected by the statement `p->f = NULL`.
- In case a subpiece gets disconnected, which pointers point to it?

Unfortunately, connection matrix information is not sufficient to answer these questions. To answer the first question we need to have some approximation for the shape of the underlying data structure. The second question requires knowledge about the possible path relationships between the various pointers pointing to the



(a) Heap Structure

kill_set = { C[p,p], C[p,r] }
gen_set = { C[p,p], C[p,q] }

(b) Kill and Gen Sets

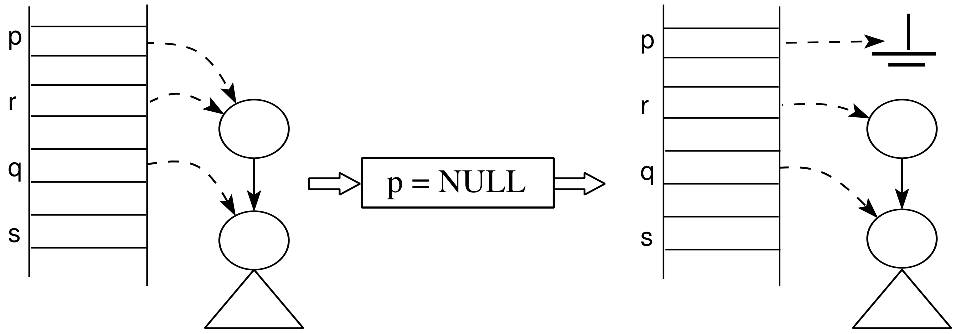
	p	q	r	s
p	1	0	1	0
q	0	1	0	0
r	1	0	1	0
s	0	0	0	0

(c) Connection Matrix C

	p	q	r	s
p	1	1	0	0
q	1	1	0	0
r	0	0	1	0
s	0	0	0	0

(d) Connection Matrix C_n

Figure 3.8: Analyzing Basic Heap Statement `p = q + k`



(a) Heap Structure

kill_set = { C[p,p], C[p,r], C[p,q] }
gen_set = {}

(b) Kill and Gen Sets

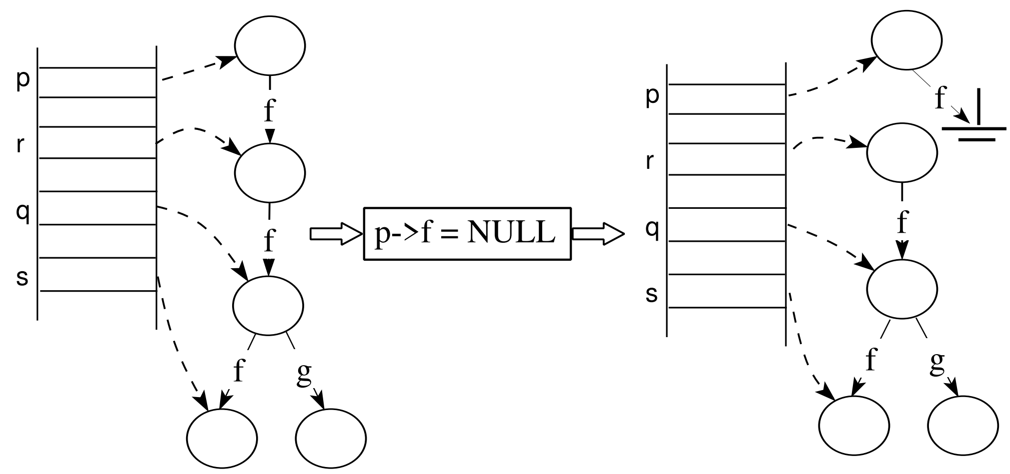
	p	q	r	s
p	1	1	1	0
q	1	1	1	0
r	1	1	1	0
s	0	0	0	0

(c) Connection Matrix C

	p	q	r	s
p	0	0	0	0
q	0	1	1	0
r	0	1	1	0
s	0	0	0	0

(d) Connection Matrix C_n

Figure 3.9: Analyzing Basic Heap Statement `p = NULL`



(a) Heap Structure

kill_set = {}
gen_set = {}

(b) Kill and Gen Sets

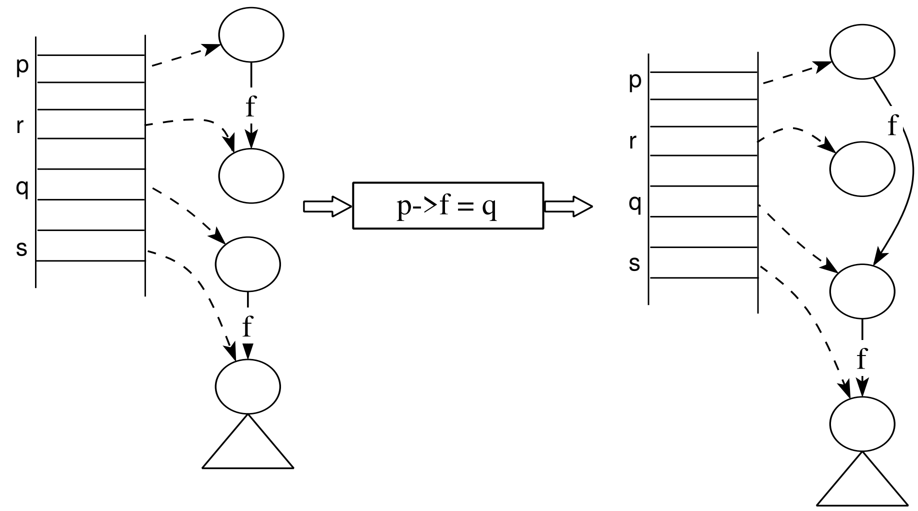
	p	q	r	s
p	1	0	1	0
q	0	1	0	1
r	1	0	1	0
s	0	1	0	1

(c) Connection Matrix C

	p	q	r	s
p	1	0	1	0
q	0	1	0	1
r	1	0	1	0
s	0	1	0	1

(d) Connection Matrix C_n

Figure 3.10: Analyzing Basic Heap Statement `p->f = NULL`



(a) Heap Structure

kill_set = {}
gen_set = { C[p,q], C[p,s], C[r,q], C[r,s] }

(b) Kill and Gen Sets

	p	q	r	s
p	1	0	1	0
q	0	1	0	1
r	1	0	1	0
s	0	1	0	1

(c) Connection Matrix C

	p	q	r	s
p	1	1	1	1
q	1	1	1	1
r	1	1	1	1
s	1	1	1	1

(d) Connection Matrix C_n

Figure 3.11: Analyzing Basic Heap Statement `p->f = q`

data structure. As such information is expensive to abstract, we do not collect it for our first abstraction for heap analysis.

In the absence of precise kill information we err conservatively, and do not kill any connection relationships for this statement. Further, this statement does not generate any new relationships. Thus both the kill and gen sets are empty for this statement, and it does not affect connection relationships.

`p->f = q` : This statement has two effects. First it potentially disconnects a subpiece of the data structure pointed to by `p`, like the previous statement `p->f = NULL`. Next, it connects the data structures pointed to by `p` and `q`. Figure 3.11 gives an illustration.

As already discussed precise kill information due to potential disconnection cannot be obtained. However new connection relationships are generated due to the interlinking of data structures pointed to by `p` and `q`. All pointers connected to `p` now get connected to all pointers connected to `q` (which include `q` itself). So we have the following analysis rule for this statement:

$$\begin{array}{l} \text{kill\_set} = \{\} \\ \text{gen\_set} = \{\ \mathrm{C}[\mathrm{r,s}] \mid \mathrm{r,s} \in \mathrm{H} \wedge \mathrm{C}[\mathrm{p,r}] \wedge \mathrm{C}[\mathrm{q,s}]\ \} \end{array}$$

This rule is illustrated in Figure 3.11. Before the statement, pointers `p` and `r` are connected to `p`, and both of them get connected to pointers `q` and `s` after the statement `p->f = q`, as shown in part (d) of the figure. Note that after the statement, all the connection relationships of pointer `r` are spurious (except $\mathrm{C}_n[\mathrm{r,r}]$). This happens because the disconnection of `r` from `p` cannot be inferred from the information available.

3.3 Summary

In this chapter, we defined and motivated the connection matrix abstraction for heap data structure analysis. We also identified eight basic statement that affect the relationships of heap-directed pointers. We developed the analysis rules for these statements, which clearly illustrate the basic principles of connection analysis. Based on these rules, one can construct a connection analysis framework for any language that supports pointer-based dynamic data structures. Our focus is on analyzing C language, using the SIMPLE intermediate representation.

Chapter 4

Interprocedural Connection Analysis for C

In this chapter, we build a complete interprocedural analysis framework to implement the connection matrix abstraction on the SIMPLE intermediate representation. The chapter is organized as follows. In section 4.1 analysis rules for basic SIMPLE statements are developed. These rules are constructed from the basic connection analysis rules presented in chapter 3. In section 4.2, analysis of control statements is described. The rules for estimating the effect of procedure calls are formulated in section 4.3. These rules are based on the interprocedural analysis framework described in chapter 2. In section 4.4, some important assumptions made by the analysis are discussed. Finally, a brief summary of the chapter is presented.

4.1 Analyzing Basic SIMPLE Statements

We have identified eight *basic heap statements* in the previous chapter (Figure 3.2), that can access or modify heap data structures. We also presented analysis rules for them. In this section, we construct the analysis rules for basic SIMPLE statements from the rules developed for basic heap statements. In this process, we remove the restriction that pointers can only point to heap objects, and take into account the presence of pointers to locations on stack. This step is crucial for applying our analysis rules to real C programs.

Instead of separately describing an analysis rule for each basic SIMPLE statement, we will develop general rules for calculating the kill and get sets, based on the variable references on the left and right hand sides of a statement. This approach enables both compact presentation of rules and clear illustration of the issues involved in their construction.

4.1.1 Identifying S-locations

In Table 4.1 we present the various types of variable references that can occur in basic SIMPLE statements relevant to heap analysis. For each variable reference we define a set of S-locations. The set of S-locations consists of the abstract stack locations (defined in section 2.3) represented by the variable reference. S-locations are represented as pairs of the form (x, D), (x, P) where x is an abstract stack location name, and D and P respectively indicate definite and possible locations. For a given variable reference say `*a`, a definite S-location (x, D) means that `*a` definitely refers to the location corresponding to the abstract stack location x, while a possible S-location (y, P) means that `*a` *possibly* refers to the location corresponding to the abstract stack location y. Further, a definite S-location represents a unique real stack location, while a possible S-location can represent more than one real stack or heap locations.

S-locations for direct variable references (Group II in Table 4.1) can be trivially determined. To determine the S-locations for variables references involving indirection, points-to information needs to be used. For example, the S-location for the variable reference `p` is simply (p, D). Now if the points-to set of variable `p` is $\{(p, r, P)$, $(p, s, P)\}$, the set of S-locations for the variable reference `*p` is $\{(r, P), (s, P)\}$.

Below we note some other important features of Table 4.1:

- The reference `(*p)[i]` is our representation of the C syntax `p[i]`, where `p` is a pointer to an array.
- The S-locations for array references are always denoted as possible locations. We use one abstract stack location to represent the whole array. This abstract location cannot be definite as it represents more than one real location.
- Points-to analysis uses one abstract location *heap* to represent all locations in the heap. The S-location corresponding to the abstract location *heap* is also always denoted as the possible location $(heap, P)$, because it represents more than one real location.

- For the variable reference `(*p).f`, the case when `p` points to *heap* is considered separately. If `p` points to *heap* then `*p` refers to a heap object, and `(*p).f` refers to the field `f` of this object. So the S-location for `(*p).f` is simply $(heap, P)$.
- An S-location representing a stack location is either of the form (p, d) or $(p.f, d)$. If it represents a heap location, it is simply $(heap, P)$.
- For some variable references, the S-location set is not defined (marked N/A). These references represent values (addresses of memory locations) and can appear only on the right hand side of a statement.

Besides S-locations, we also define the term *Root* for each variable reference. It is defined as the pointer from which the reference originates. The Root for all the references in Table 4.1 is `p` except for `p.f` and `&p.f`, for which it is `p.f`. For `malloc()` and `NULL` it is not defined.

Var Ref	S-location Set	Root	Group
`&p`	N/A	`p`	
`&p.f`	N/A	`p.f`	
`&p[i]`	N/A	`p`	I
`&(*p)[i]`	N/A	`p`	
`&(*p).f`	N/A	`p`	
`p`	$\{(p, D)\}$	`p`	
`p.f`	$\{(p.f, D)\}$	`p.f`	II
`p[i]`	$\{(p, P)\}$	`p`	
`*p`	$\{(x, d) \mid (p, x, d) \in Q\}$	`p`	
`(*p)[i]`	$\{(x, P) \mid (p, x, d) \in Q\}$	`p`	III
`(*p).f`	$\{(x.f, d) \mid (p, x, d) \in Q \wedge x \neq heap\}$	`p`	
	$\cup\{(heap, P) \mid (p, heap, P) \in Q\}$		
`NULL`	N/A	N/A	
`malloc()`	N/A	N/A	IV

Table 4.1: S-location sets relative to points-to set Q.

4.1.2 Analysis Based on S-locations

We now present the analysis of a basic SIMPLE statement, denoted as S. Let C be the input connection matrix. Let H be the set of pointers whose relationships are abstracted by the matrix C. This includes all the pointers which can be heap-directed (i.e. point to *heap*), at some point in the program, and are accessible in the procedure containing statement S. Let Q be the set of points-to relationships valid at statement S (i.e. before executing S). Let the left and right hand sides of the statement be respectively denoted as lhs(S) and rhs(S).

To be of relevance to connection analysis, statement S should be of pointer type i.e. it should perform a pointer update. This information is directly available from the SIMPLE AST. Given that the statement S satisfies this criterion, we now compute its kill and gen sets with respect to connection relationships.

Kill Set Computation

To compute the kill set only the variable reference on lhs(S) needs to be considered. We have kill information only if this variable reference represents a definite S-location. In this case the location is definitely updated and all its existing relationships get killed. For example if the variable reference on lhs(S) is `p` with S-location (p, D), all relationships of `p` get killed, as the statement would definitely update `p`. However, if the reference is `p[i]` then we cannot kill any relationships. The corresponding S-location (p, P) represents the entire array, while only one element of the array would be updated by the statement. Similarly for the variable reference `(*p).f`, if `p` definitely points to stack location `x`, its S-location would be $(x.f, D)$. So all relationships of `x.f` can be killed.

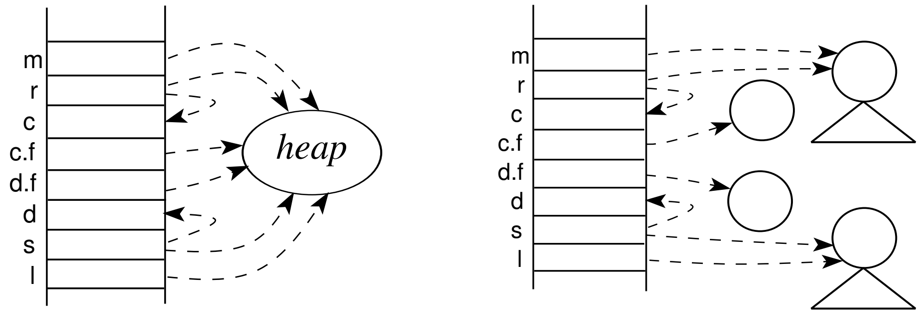
Based on the above discussion, the general rule for calculating the kill set can be expressed as follows:

$$\text{kill\_set(S)} = \{\ \text{C[p,s]} \mid (p, D) \in \text{S-locations(lhs(S))} \wedge \text{p,s} \in \text{H} \wedge \text{C[p,s]}\ \}$$

This general rule is consistent with the kill_set computation rules defined for the basic heap statements in section 3.2.1. For the basic heap statements with `p` on lhs(S), S-location(lhs(S)) is (p, D): so all the relationships of `p` get killed. For the two statements with `p->f` on lhs(S), S-location(lhs(S)) is $(heap, P)$: so no relationships can be killed.

Gen Set Computation

The set of connection relationships generated by the statement S depends on variable references on both lhs(S) and rhs(S). Every combination of S-locations represented by lhs(S) and rhs(S) needs to be considered. For example let statement S be `(*r).f = (*s).f`. Let the points-to relationships of `r` and `s` be as given in Figure 4.1(a). Since `r` possibly points to stack location `c` or to the abstract location *heap* (which represents all heap locations), we have $\{(c.f, P), (heap, P)\}$ as S-locations(lhs(S)). Similarly as `s` possibly points to stack location `d` or to *heap*, we have $\{(d, P), (heap, P)\}$ as S-locations(rhs(S)). Since the set S-locations(lhs(S)) consists of only possible locations, kill_set(S) is empty. To compute the gen_set for S, we need to consider the following assignment statements generated by the four possible combinations of S-locations for lhs(S) and rhs(S): `c.f = d.f`, `c.f = s->f` where `s` is heap-directed, `r->f = d.f` where `r` is heap-directed, and `r->f = s->f` where both `r` and `s` are heap-directed. Thus, gen_set(S) would be the union of the gen sets of these four possible assignments it represents.



(a) (b)

Points-to Set: $\{\ (r, c, P),\ (r, heap, P),\ (s, d, P),\ (s, heap, P)\ \}$
S-locations(`(*r).f`): $\{\ (c.f, P),\ (heap, P)\ \}$
S-locations(`(*s).f`): $\{\ (d.f, P),\ (heap, P)\ \}$

(c)

Figure 4.1: Example to Illustrate Identification of S-locations

It can be noticed from the above example, that every combination of S-locations generates a simple assignment statement. In a simple assignment statement, all references to stack locations are through direct variable references, while all references to heap locations are through variable references involving an indirection. This happens because points-to relationships on the stack, get factored out during calculation of S-locations. For example, the simple assignment statement `c.f = d.f` results from factoring out the points-to relationships (r, c, P) and (s, d, P). Henceforth, we refer to a simple assignment statement generated by two given S-locations, say S-lloc and S-rloc, as S-statement(S-lloc,S-rloc). When clear from the context we will simply refer to it as S-statement.

In general, a basic SIMPLE statement S can itself be an S-statement or can be represented as a *collection* of S-statements. In the latter case, the S-statements capture the different ways statement S can be represented, when program execution reaches it. So the gen set for S is computed as the union of the gen sets of the S-statements it can generate.

Thus to compute the gen set for a basic SIMPLE statements, we simply need to define the rules to compute the gen sets for its S-statements. We now identify the various types of S-statements that can occur due to various combinations of S-locations. We show that each type of S-statements corresponds to one of the eight basic heap statements discussed in section 3.2.1. A general rule is derived to compute the gen_set for each type from the rule developed for its corresponding basic heap statement. We recall that C is the input connection matrix to S, and H is the set of pointers whose relationships are abstracted by C.

Given any two S-locations S-lloc and S-rloc, each S-statement generated by their combination, denoted as T, can be analyzed as follows:

Case 1: S-lloc represents a stack location:

In this case, S-lloc is either of the form (x, d) or $(x.f, d)$, with corresponding real stack location (i.e. variable reference on lhs(T)) as `x`, `x.f` or `x[i]` (Group II in Table 4.1).

Case 1(a): If S-rloc also represents a stack location, S-statement T has the same effect as the basic heap statement `p = q`. Both of them make one stack-resident pointer to point to the data structure pointed to by another stack-resident pointer. So the gen_set for T can be computed using the rule defined for the statement `p = q` in section 3.2.1.

Case 1(b): If S-rloc is $(heap, P)$, rhs(T) can be of the form `(*x)[i]`, `*x` or `(*x).f` (Group III in Table 4.1), with `x` pointing to a heap location. Thus rhs(T) represents a heap-resident pointer. In this case, T can be analyzed in the same fashion as the basic heap statement `p = q->f`, where `p` is a stack-resident pointer and `q->f` is a heap-resident pointer.

We had noted in section 3.2.1, that the gen sets for the basic heap statements `p = q` and `p = q->f` can be computed using the same rule. The underlying assumption is that the heap-resident pointer `q->f` points to the same data structure as its origin pointer on the stack `q` (i.e. `q` = Root(`q->f`)). This assumption can be violated if `q->f` points to `NULL` or to a stack location. In either case, the connection relationships generated will be spurious but *safe*, as discussed in section 3.2.1. We further discuss the implications of a heap-resident pointer pointing to a stack location in section 4.4.

The following rule was developed in section 3.2.1 to compute the gen set for these basic heap statements:

$$\text{gen\_set} = \{ \text{C[p,s]} \mid \text{s} \in \text{H} \wedge \text{C[q,s]} \} \cup \{ \text{C[p,p]} \mid \text{C[q,q]} \}$$

Based on this rule, we derive the following general rule to compute the gen set for the S-statements with S-lloc as a stack location:

$$\text{stack\_lhs\_gen\_set(C,H,x,y)} = \{ \text{C[x,z]} \mid \text{x,y,z} \in \text{H} \wedge \text{C[y,z]} \} \cup \{ \text{C[x,x]} \mid \text{C[y,y]} \}$$

In this rule, x represents the stack-resident pointer refered on lhs(T). If rhs(T) is a stack-resident pointer, then y simply represents this pointer. If rhs(T) is a heap-resident pointer (like `(*y).f`), then y represents its origin pointer on the stack i.e. its Root. The gen_set consists of connection relationships generated by connecting x with all the pointers connected with y, and with itself if y is presently heap-directed (C[y,y] = 1). It is named stack_lhs_gen_set as it basically depends on lhs(T) being a stack location. Note that connection relationships are generated only if pointers x and y are relevant to connection analysis (i.e. belong to the set H).

Case 2: S-lloc represents a heap location:

In this case S-lloc is $(heap, P)$ and lhs(T) is a variable reference of the form `(*x).f`, `(*x)[i]` or `*x` (Group III in Table 4.1) with `x` pointing to a heap location.

Case 2(a): If S-rloc represents a stack location, S-statement T is equivalent to the basic heap statement `p->f = q` for purposes of connection analysis.

Case 2(b): If S-rloc is $(heap, P)$, the corresponding basic heap statement would be `p->f = q->f`. This statement can be analyzed in the same fashion as the statement `p->f = q`, because the pointer `q` points to the same data structure as `q->f` (as already discussed).

The following rule was developed to compute the gen set for the basic heap statement `p->f = q`:

$$\text{gen\_set} = \{ \text{C[r,s]} \mid \text{r,s} \in \text{H} \wedge \text{C[p,r]} \wedge \text{C[q,s]} \}$$

Based on this rule, we derive the following general rule to compute the gen set for the S-statements with S-lloc as a heap location:

heap_lhs_gen_set(C,H,x,y) = { C[w,z] | x,y,w,z $\in$ H $\wedge$ C[x,w] $\wedge$ C[y,z] }

In this rule, x represents the Root of the heap-resident pointer refered on lhs(T). If rhs(T) is a stack-resident pointer, then y simply represents this pointer. If rhs(T) is a heap-resident pointer then y represents its Root. The gen_set consists of connection relationships generated by connecting all pointers connected with x, with all the pointers connected with y. It is named heap_lhs_gen_set as it basically depends on lhs(T) being a heap location. Note that again connection relationships are generated only if pointers x and y are relevant to connection analysis (i.e. belong to the set H).

Special cases:

In this case rhs(T) consists of a variable reference for which the S-location set is not defined (marked N/A). Note that these variable references can occur only on the right hand side of a statement, and not on the left hand side. Below we discuss the rules for each such reference on rhs(T). We also discuss the case when rhs(S), hence rhs(T), is an arithmetic expression. We recall that T is an S-statement for the basic SIMPLE statement S. So when S-rloc is not defined, the variable reference on rhs(S) and rhs(T) will be same. We now consider these variable references:

`NULL` : If rhs(T) is `NULL` no new relationships are generated and the gen set is empty. Some relationships may be killed if lhs(S) happens to be a definite S-location. This is taken care of by kill_set computation.

`malloc()`: In this case, if S-lloc represents a heap location, the S-statement T would be of the form `p->f = malloc()`. This statement simply adds an anonymous node to the data structure pointed to by `p`, and does not generate or kill any connection relationships. So we do not need to consider this case for connection analysis. If S-lloc represents a stack location, the rule for the basic heap statment `p = malloc()`, is directly applicable to compute the gen_set for the S-statement under consideration.

This rule was:

gen_set = { C[p,p] | p $\in$ H }

We derive the following general rule:

malloc_gen_set(C,H,x) = { C[x,x] | x $\in$ H }

This rule takes any pointer denoted by x which belongs to H, and generates the relationship C[x,x] that connects x with itself. This indicates that x is now heap-directed. In C-language, there are several memory allocating routines other than `malloc()`. Further, users typically define their own allocation routines, which in turn invoke the standard library routines. We discuss all these cases in section 4.4.

Address Operation: The references with the `&` operator in Table 4.1 represent memory addresses of variables. The first three references represent addresses of variables on the stack. The next two references (`&(*p)[i]`, `&(*p).f`) can also represent addresses of variables in the heap, if `p` points to a heap location. Connection matrix only abstracts the connectivity of objects in the heap. So if the rhs(S) consists of memory address of a stack location, the SIMPLE statement S does not generate any new connection relationships.

Thus we need to consider only the references `&(*p)[i] and &(*p).f` where `p` points to a heap location. The former reference represents the address of a particular index in a heap-allocated array. The latter represents the address of a particular field in a heap-allocated structure. As noted in the rule for the basic heap statement `p = &(q->f)` in section 3.2.1: for purpose of connection analysis, a pointer pointing to a specific index or field of a heap object, is considered to be pointing to the object itself. Thus effectively for these references the S-location is $(heap, P)$. So the gen_set for the S-statement T can be computed using the stack_lhs_gen_set or heap_lhs_gen_set rules depending on the location represented by S-lloc. The arguments would be C, H, x and y, where x depends on S-lloc, and y denotes Root(rhs(S)).

Arithmetic Expressions: Finally we consider the case when the SIMPLE statement S involves pointer arithmetic. In this case lhs(S) would be a variable reference, while rhs(S) would be an arithmetic expression of the form `q op k`. Here `q` represents a pointer, `op` denotes a `+` or `-` operation, and `k` represents an integer. According to SIMPLE grammar, `q` should be a scalar pointer: the type represented by the references `p` and `p.f` in Table 4.1. As noted in the rule for the basic heap statement `p = q op k` in section 3.2.1: we assume that after pointer arithmetic, a heap-directed pointer continues to point to its present target, though at a different offset. With this assumption, effectively the S-location(rhs(S)) is (q, P): a stack location. The gen_set for the S-statement T can again be computed as per the stack_lhs_gen_set or heap_lhs_gen_set rules, depending on the location represented by S-lloc. The arguments would be S, C, H, x and y, where x depends on S-lloc and y denotes `q`.

Thus the gen set of a basic SIMPLE statement S can be computed by first computing the gen sets for its S-statements, and then unioning them. The complete rules for computing the gen set are given in Figures 4.2 and 4.3. The complete algorithm for analyzing a SIMPLE statement is presented in Figure 4.4.

We now demonstrate the analysis for the statement S `(*r).f = (*s).f` in Figure 4.1. Part (a) gives the points-to relationships of pointers `r` and `s`, part (b) shows the connection relationships between heap-directed pointers, and part(c) gives the points-to and S-location sets for the statement S. According to the figure the S-location sets for lhs(S) and rhs(S) are: $\{(c.f, P), (heap, P)\}$ and $\{(d.f, P), (heap, P)\}$

```
/* Compute the gen set for statement S with input connection
 * matrix C and H as the set of pointers abstracted by C */
fun build_gen_set(S,C,H)
  gen_set = {} /* Initialize gen set */
  if (is_null(rhs(S))) /* No new relationships are generated */
    return(gen_set)
  Let l = Root(lhs(S)) /* Root of Var Ref on lhs(S) */
  Let r = Root(rhs(S)) /* Root of Var Ref on rhs(S) */
  foreach (x, d) ∈ S−locations(lhs(S))
    if ((x, d) ≡ (heap, P)) /* S−location(lhs(S)) is a heap location */
      gen_set = gen_set ∪ build_heap_lhs_gen_set(S,C,H,l,r)
    else /* S−location(lhs(S)) is a stack location */
      gen_set = gen_set ∪ build_stack_lhs_gen_set(S,C,H,x,r)
  return(gen_set)
```

Figure 4.2: Computing Gen Set for a Basic SIMPLE Statement

with `r` as Root(lhs(S)) and `s` as Root(rhs(S)). Since all S-locations(lhs(S)) are possible locations we have kill_set(S) = {}. The combinations of S-locations give us four S-statements. We compute the gen sets for them below:

1. T1 : `c.f = d.f`. Both S-locations are on the stack. Thus gen_set(T1) = stack_lhs_gen_set(C,H,c.f,d.f) Pointer `c.f` gets connected with all pointers `d.f` is connected with. Since `d.f` is only connected with itself, so the only new connection relationships generated is C[c.f,d.f]. So we get:

   gen_set(T1) = { C[c.f,d.f] }

2. T2 : `c.f = s->f` with `s` pointing to a heap location. S-lloc is stack location while S-rloc is $(heap, P)$. Thus gen_set(T2) = stack_lhs_gen_set(C,H,c.f,s). Pointer `c.f` gets connected with all pointers `s` is connected with. Thus `c.f` gets connected with pointers `s` and `l`, and we get:

   gen_set(T2) = { C[c.f,s], C[c.f,l] }

3. T3 : `r->f = d.f` with `r` pointing to a heap location. S-lloc is $(heap, P)$ and S-rloc is a stack location. So gen_set(S3) = heap_lhs_gen_set(C,H,r,d.f). All

```
/* S: Statement, C: Connection Matrix, H: Set of pointers in C
 * (x,d): S-location(lhs(S)), r: Root(rhs(S)) */
fun build_stack_lhs_gen_set(S,C,H,x,r)
  gen_set = {} /* Initialize gen set */
  if (is_malloc(rhs(S)) /* x = malloc() */
    gen_set = malloc_gen_set(C,H,x)
  else if (is_address_op(rhs(S))) and (r, heap, P) /* x = &(r->f) */
    gen_set = stack_lhs_gen_set(C,H,x,r)
  else if (is_arith_expr(rhs(S))) /* x = r op k */
    gen_set = stack_lhs_gen_set(C,H,x,r)
  else
    foreach (y, d) ∈ S-locations(rhs(S))
      if ((y, d) ≡ (heap, P)) /* x = r->f: r is heap-directed */
        gen_set = gen_set ∪ stack_lhs_gen_set(C,H,x,r)
      else /* (y, d) is a stack location: x = y */
        gen_set = gen_set ∪ stack_lhs_gen_set(C,H,x,y)
  return(gen_set)

/* S: Statement, C: Connection Matrix, H: Set of pointers in C *
 * (heap,P): S-location(lhs(S)), l: Root(lhs(S)), r: Root(rhs(S)) */
fun build_heap_lhs_gen_set(S,C,H,l,r)
  gen_set = {} /* Initialize gen set */
  if (is_address_op(rhs(S))) and (C[r,r]) /* l->f = &(r->f) */
    gen_set = heap_lhs_gen_set(C,H,l,r)
  else if (is_arith_expr(rhs(S))) /* l->f = r op k */
    gen_set = heap_lhs_gen_set(C,H,l,r)
  else
    foreach (y, d) ∈ S-locations(rhs(S))
      if ((y, d) ≡ (heap, P)) /* l->f = r->f: l and r are heap-directed */
        gen_set = gen_set ∪ heap_lhs_gen_set(C,H,l,r)
      else /* (y, d) is a stack location: l->f = y */
        gen_set = gen_set ∪ heap_lhs_gen_set(C,H,l,y)
  return(gen_set)
```

Figure 4.3: Computing Gen Sets using S-locations

```
/* Analyze statement S with input connection matrix C and
 * H as the set of pointers abstracted by C */
fun process_basic_stmt(S,C,H) =
  if (! is_pointer_type(S) ) /* not a pointer assignment */
    return(C)
  /* Connection relationships of definite S-locations are killed */
  kill_set = { C[x,z] | (x, D) ∈ S−locations(lhs(S)) ∧ x,z ∈ H ∧ C[x,z] }
  gen_set = build_gen_set(S,C,H) /* Build the gen set */

 ∀ r,s ∈ H, C_n[r,s] = C[r,s] /* Build the new Connection Matrix */
 ∀ entries C[r,s] ∈ kill_set, C_n[r,s] = 0 /* Delete killed relationships */
 ∀ entries C[r,s] ∈ gen_set, C_n[r,s] = 1 /* Add generated relationships */
 return(C_n)
```

Figure 4.4: Analyzing a Basic SIMPLE Statement

pointers connected with `r` get connected with all pointers connected with `d.f`. Thus `r` and `m` get connected with `d.f`, and we get:

gen_set(T3) = { C[r,d.f], C[m,d.f] }

4. T4 : `r->f = s->f` with `r` and `s` pointing to heap locations. Both S-lloc and S-rloc represent $(heap, P)$. So gen_set(T4) = heap_lhs_gen_set(C,H,r,s). All pointers connected with `r` get connected with all pointers connected with `s`. Thus `r` and `m` get connected with `s` and `l`, and we get:

   gen_set(T4) = { C[r,s], C[r,l], C[m,s], C[m,l] }

We get gen_set(S) by unioning the gen sets of the above four S-statements:

gen_set(S) = { C[c.f,d.f], C[c.f,s], C[c.f,l], C[r,d.f],
C[m,d.f], C[r,s], C[r,l], C[m,s], C[m,l] }

We finally note that assignments involving structures are handled by breaking them down into assignments between individual fields. However only fields of pointer type are considered for connection analysis.

4.2 Analyzing Compositional Control Statements

In this section we present the analysis of control statements. SIMPLE supports the following control statements: `if`, `for`, `while`, `do-while`, `switch`, `continue`, and `break`. Thus only compositional control statements are supported, as `goto` statements are eliminated during the program structuring phase [EH94].

The analysis of control statements builds upon two fundamental concepts: (i) merge operation for the flow information, and (ii) fixed-point computation.

The merge operation is required to approximate the data flow information at control flow join points in the program. For example to obtain the output information for an `if` statement, one needs to merge the output information from its if-part and else-part. Fixed-point computation is needed to approximate the flow information for loop statements. In this context, a fixed-point is reached when two successive approximations of a loop do not result in any new information.

Merge Operator : We now define the merge operator for connection matrix information. Since the information abstracted by connection matrix is binary in nature, the merge operator turns out to be simply the logical OR operation. Two connection matrices C and C_n can be merged as follows, with C_n as the resulting matrix:

$$\mathrm{Merge}(C,C_n) \Rightarrow \forall\ r,s \in H,\ C_n[r,s] = C_n[r,s] \vee C[r,s]$$

Thus if a connection relationship exists in either of the matrices, it exists in the resulting matrix. It should be noted that having an efficient merge operation was one of the major design criteria for connection matrix abstraction. In section 1.3 we had shown that most heap analysis techniques are rendered expensive due to complex merge operation.

To simplify the explanation, we first consider compositional control statements without the presence of `break` and `continue` statements. Next we discuss how these statements are accommodated in the analysis framework. The complete framework for analyzing compositional control statements was developed in [Sri92, Ema93]. We simply adapt this framework for connection analysis.

4.2.1 Analysis without `break` and `continue` Statements

<u>`if` statement</u>:

Figure 4.5 gives the algorithm as well as a pictorial representation of the analysis of `if` statement. The input connection matrix is C. If the condition `cond` does a pointer equality check (e.g. `p == NULL`), it is considered as an assignment. The input matrix C is modified to take this into account, and the resulting matrix C0 is propagated to the then-body. Similarly if the condition `cond` does a pointer inequality check (e.g. `p != NULL`), its negation is considered to obtain the modified matrix C1 for the else-body. Next the output matrices C2 and C3 from the then-body and else-body are merged to obtain the output matrix for the `if` statement. In case the else-body is empty, its output will be same as its input matrix C1.

```
/* C : Input connection matrix,
 * H : Set of pointers abstracted by C,
 * ign : Current invocation graph node */
fun process_if(cond,then_body,else_body,C,H,ign) =
  C0 = process_basic_stmt(cond,C,H);
  C1 = process_basic_stmt(!cond,C,H);
  C2 = process_stmt(then_body,C0,H,ign);
  C3 = process_stmt(else_body,C1,H,ign);
  return(Merge(C1,C2));
```

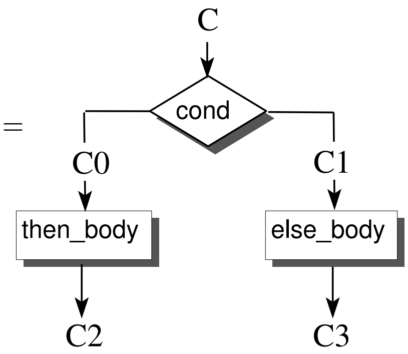


Figure 4.5: Analyzing an `if` Statement

<u>`while` statement</u>:

The algorithm for analyzing a `while` loop is shown in Figure 4.6 along with a pictorial representation of the analysis. The input connection matrix is C which is also the first approximation as the output matrix for the loop. It is modified to C1 to take into account the condition `cond` if it happens to be a pointer equality test. The matrix C1 is then propagated through the loop body to get the matrix C2. We merge C and C2 to obtain the new approximation. This process is repeated until the previous and current approximations turn out to be identical i.e. a fixed-point is reached.

Other loop constructs like `do-while` and for statements are analyzed in a similar way, using the analysis framework described in [Sri92, Ema93].

4.2.2 Analysis with `break` and `continue` Statements

We first recall the semantics of `break` and `continue` statements. Execution of a `break` statement terminates the execution of the closest `while`, `do-while`, `for` or `switch` statement. The control flow is then immediately transfered to the point just after the body of the corresponding statement. A `continue` statement terminates the execution of the body of the closest `while`, `do-while`, or `for` statement. The control flow is immediately transferred to the beginning of the loop body and the execution continues from that point with a re-evaluation of the loop condition.

```
/* C : Input connection matrix,
 * H : Set of pointers abstracted by C,
 * ign : Current invocation graph node */
fun process_while(cond,body,C,H,ign) =
  do
    prevC = C;
    C1 = process_basic_stmt(cond,C,H);
    C2 = process_stmt(body,C1,H,ign);
    C = Merge(C,C2);
  while (C != prevC);
  return(C);
```

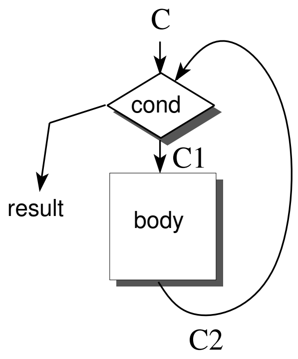


Figure 4.6: Analyzing a `while` Statement

To handle these statements we use two structures called the `break-list` and `continue-list`. On encountering a `break` or `continue` statement, the current connection matrix is stored in the `break-list`/`continue-list`, and $\perp$ (BOTTOM) is passed as output, where $\perp$ denotes no information. Propagating $\perp$ corresponds to taking paths in the program that will never occur in any execution. Any statement with the input $\perp$ produces $\perp$ as output. The merge operation for $\perp$ and a connection matrix C is as follows:

$$\text{Merge}(C,\perp) = \text{Merge}(\perp,C) = C$$

This rule is based on the fact, that a path with $\perp$ as its output is an impossible execution path, and its output can be ignored during the merge.

We now explain how the information stored in the structures `break-list` and `continue-list` is used in our analysis. Let us first consider the `continue-list`. Since a `continue` statement takes the program control back to the beginning of the corresponding loop, the following three matrices should be merged to get a new approximation for the loop, each time it is analyzed (each of these matrices can form a new input to the loop):

- The matrix representing the previous approximation for the loop.
- The output matrix obtained by analyzing the loop-body with the previous approximation as the input.
- The matrices stored in the `continue-list`. Note that each of these matrices is a potential input to the loop corresponding to some path in the loop body terminated by a `continue` statement.

This process is repeated until a fixed-point is reached.

Unlike the `continue-list`, the `break-list` does not participate in the fixed-point calculation. Each matrix in the `break-list` represents a potential output of the enclosing loop or `switch` statement. So matrices stored in this list are simply merged with the approximation obtained for the statement under analysis, to get its final approximation.

In the actual implementation, the whole list of matrices is not maintained. Every time a `break` or `continue` statement is encountered, the new matrix is simply merged with the one existing in the corresponding list.

In Figures 4.7 we present the algorithm to analyze a `while` statement in the presence of `break` and `continue` statements. The framework for analyzing other statements like `do-while`, `for`, and `switch` statements in this context, is presented in [Sri92, Ema93], and is similarly adapted for connection analysis.

4.3 Interprocedural Analysis

In section 2.4 we had described a framework for context-sensitive interprocedural analysis. This framework is built by points-to analysis, and its salient features include: (i) the invocation graph representation, which precisely captures the invocation structure of the program, (ii) context-sensitive map information deposited on invocation graph nodes, and (iii) accurate handling of indirect calls through function pointers.

We now extend connection analysis to handle procedure calls using this framework.

```
/* C : Input connection matrix,
 * H : Set of pointers abstracted by C,
 * ign : Current invocation graph node */
fun process_while(cond,body,C,H,ign) =
  do
    prevC = C;
    C1 = process_basic_stmt(cond,C,H);
    C2 = process_stmt(body,C1,H,ign);
    /* cont_lst denotes continue-list */
    C3 = Merge(C2, cont_lst);
    C = Merge(C,C3);
  while (prevC != C);
  /* break_lst denotes break-list */
  result = Merge(C,break_lst);
  return(result);

fun process_break(C,break_lst) =
  break_lst = Merge(C,break_lst);
  return ⊥;

fun process_break(C,cont_lst) =
  cont_lst = Merge(C,cont_lst);
  return ⊥;
```

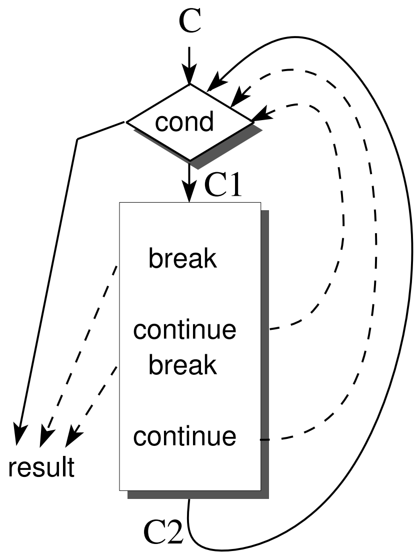


Figure 4.7: Analyzing a `while` Statement with `break` and `continue` Statements

4.3.1 An Approach Based on Invocation Graphs

The overall strategy for interprocedural analysis is depicted in Figure 4.8, and the complete rules are given in Figures 4.9 and 4.10. The general idea is that, first, the connection matrix C_m at the call-site is *mapped* to prepare the input connection matrix C_e for the called procedure. Next, the body of the procedure is analyzed with this input matrix and the output matrix obtained (C_x) is *unmapped* and the resulting matrix C_n is returned to the call-site. In effect, this strategy leads to a depth first traversal of the invocation graph. Every time a procedure call is analyzed for some call-chain, there exists an invocation graph node corresponding to it.

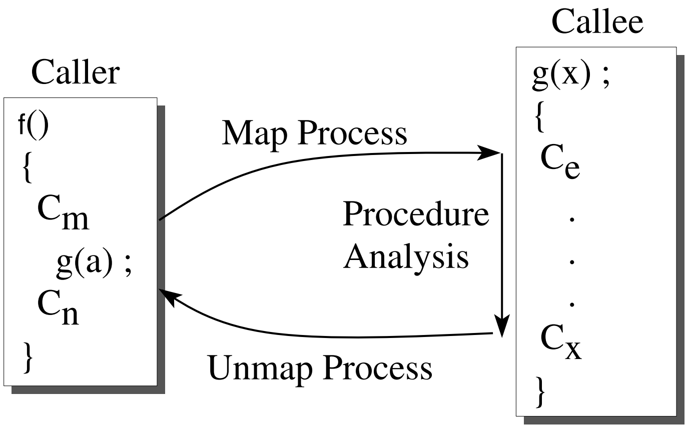


Figure 4.8: Interprocedural Startegy

With this strategy, connection relationships induced by one call-site are never returned to another call-site, and similarly connection relationships arriving from different call-sites are never simultaneously used to generate new relationships. However, the worst-case cost of this approach is exponential in the number of call-sites in the program. It may not scale for programs with a large number of call-sites for procedures having large invocation (sub)graphs. Empirical results [EGH94] indicate that this scheme is efficient for a broad range of programs. Presently we use simple memoization to avoid potential explosion, as shown in the rules for interprocedural analysis, where memoization can be turned on by setting a flag. More advanced techniques for memoization are currently being developed.

With the above approach, when a procedure is analyzed for the first time, the connection matrix valid at each statement (i.e. the matrix valid before processing the statement), is deposited in the corresponding statement node in the SIMPLE tree. For subsequent passes, the current matrix is merged with the one deposited in the tree, and the resulting matrix is stored in the SIMPLE node. Thus, the final matrix in each statement node captures the connection relationships that may arise due to all possible invocations of the procedure.

With the overall strategy being clear, we first explain our approach to handle recursive procedure calls, indirect procedure calls and `return` statements. Then we describe in detail the process of mapping and unmapping connection matrices.

4.3.2 Handling Recursive Procedure Calls

The cases of approximate and recursive procedure calls shown in Figure 4.10 work together to implement a safe and accurate fixed-point computation for recursion. As we have explained in section 2.4.1, all possible unrollings for call-chains involving recursion are approximated by introducing matched pairs of recursive and approximate nodes in the invocation graph. Each recursive node marks a place where a fixed-point computation must be performed, while each approximate node marks a place where the current stored approximation for the function should be used (instead of evaluating the call, the stored output matrix is used directly).

At each recursive node we store an input matrix, an output matrix, and a list of pending input matrices. The input and output matrices can be thought of as approximating the effect of the call associated with the recursive function (let us call it `f`). The fixed-point computation generalizes the stored input matrix until it finds an input matrix that summarizes all invocations of `f` in any unrolled call tree starting at the recursive node for `f`. Similarly, the output matrix is generalized to find a summary for the output for any unrolling of the call tree starting in the recursive node for `f`. The generalizations of the input and output matrices may alternate, with a new generalization of the output matrix causing the input matrix to change.

Let us consider the rule for the approximate node in Figure 4.10. In this case, the current input matrix is compared to the stored input matrix of the matching recursive node. If the current input matrix is contained in the stored input matrix, then we use the stored output matrix as the result. Otherwise, the result is not yet known for this input matrix, so the input matrix is put on the pending list, and $\perp$ is returned as the result. Note that an approximate node never evaluates the body of a function, it either uses the stored result, or returns $\perp$.

Now consider the recursive rule. In this case we have an iteration that only terminates when the input matrix is sufficiently generalized (the pending list of input matrices is empty) and the output matrix is sufficiently generalized (the result of evaluating the call doesn't add any new information to the stored output matrix).

```
/* Analyze the procedure call corresponding to the invocation graph
 * node ign. funcBody: Body of the called function, C: Input matrix,
 * H: set of pointers abstracted  by the input matrix,
 * flgMemo: flag to set memoization on */
fun process_call(funcBody,C,H,actualList,formalList,ign,flgMemo) =
  case type(ign) of
    < Ordinary > =>
      funcInput  = cn_map(C,H,actualList,formalList,ign);
      /* if already computed */
      if ((flgMemo) and (funcInput == ign.storedInput))
        return(cn_unmap(C,H,ign.storedOutput,ign));
      else /* compute output,  store input and output */
        /* funcInput.pSet is the set of pointers abstracted by
         * the connection matrix funcInput */
        funcOutput = process_stmt(funcBody,funcInput,funcInput.pSet,ign);
        ign.storedInput = funcInput;
        ign.storedOutput = funcOutput;
        return(cn_unmap(C,H,funcOutput,ign));

    < Approximate > =>
      return(process_call_approx(funcBody,C,H,actualList,
                                  formalList,ign,flgMemo));

    < Recursive > =>
      return(process_call_recur(funcBody,C,H,actualList,
                                 formalList,ign,flgMemo));
```

Figure 4.9: Compositional Interprocedural Rules for Connection Analysis

```
/* Rules to analyze approximate and recursive calls */
fun process_call_approx(funcBody,C,H,actualList,formalList,ign,flgMemo) =
   funcInput  = cn_map(C,H,actualList,formalList,ign);
   recIgn = ign.recEdge;  /* get partner recursive node in inv. graph */
   /* if this input is contained in  stored input, use stored ouput */
   if isSubsetOf(funcInput,recIgn.storedInput)
     return(cn_unmap(C,H,recIgn.storedOutput,ign);
   else  /* put this input in the pending list, and return ⊥ */
     addToPendingList(funcInput,recIgn.pendingList);
     return ⊥;

fun process_call_recur(funcBody,C,H,actualList,formalList,ign,flgMemo) =
   funcInput  = cn_map(C,H,actualList,formalList,ign);
   if ((flgMemo) and (funcInput == ign.storedInput))  /* already computed */
     return(cn_unmap(C,H,ign.storedOutput,ign));
   else
     /* initial input estimate */       /* initial output estimate */
     ign.storedInput  = funcInput;       ign.storedOutput = ⊥;
     ign.pendingList  = {};              flgMemo = done = false;
     do /* process the function body */
        sInput = ign.storedInput;
        funcOutput = process_stmt(funcBody,sInput,sInput.pSet,ign,flgMemo);
        /* if there are unresolved inputs, merge inputs and restart */
        if (ign.pendingList != {})
          ign.storedInput = Merge(ign.storedInput,pendingListInputs);
          ign.pendingList = {};   ign.storedOutput = ⊥;
        /* check to see if the new output is included in old output */
        else if isSubsetOf(funcOutput,ign.storedOutput)
           done = true;
        else /* merge outputs and try again */
           ign.storedOutput = Merge(ign.storedOutput,funcOutput);
     while (not done);
     /* reset stored input to initial input for future memoization */
     ign.storedInput = funcInput;
     /* return the fixed-point after unmapping */
     return(cn_unmap(C,H,ign.storedOutput,ign));
```

Figure 4.10: Compositional Interprocedural Rules for Connection Analysis

An important point to note from Figure 4.10 is that the memoization flag is set to False while handling a recursive procedure call. This is done to avoid the possible reuse of an incompletely computed output for an incompletely computed input at approximate nodes.

4.3.3 Handling Indirect Procedure Calls

Indirect procedure calls through function pointers are easily incorporated in the analysis. The points-to analysis has resolved all the functions possibly pointed to by the function pointer, and one just analyzes each possibility, merging the output matrices. The exact rule is given in Figure 4.11.

```
/* Analyze the indirect procedure call corresponding to the invocation
 * graph  node ign. C: Input matrix, H: Set of pointers  abstracted by
 * the input matrix, flgMemo: flag to set memoization on */
fun process_indirect_call(C,H,actualList,ign,flgMemo) =
  ignSet = childNodesOf(ign) /* set of functions invocable */
  funcOutput = {}
  foreach igNode in (ignSet)
    /* get the output matrix for each invocable function */
    igNodeOutput = process_call(igNode.funcBody,C,H,actualList,
                                igNode.formalList,igNode,flgMemo);
    /* merge the output matrices */
    funcOutput = Merge(funcOutput,igNodeOutput);
  return(funcOutput);
```

Figure 4.11: Handling Indirect Procedure Calls

4.3.4 Return Statement

Function calls of the form `x = f(args)` are handled in the same way as normal function calls, with a slight modification. In this case the function `f` should have at least one occurrence of the `return` statement.

For each function `f` returning a pointer type variable, we define a global variable `return_f` with the same type as `f`. Using this newly defined variable, we treat `return(var)` as:

```
return_f = var;

return;
```

and we treat `x = f(args)` as:

```
f(args);

x = return_f;
```

The `return` statement is handled in the same way as `break` statement. Another structure called `return-list` is maintained to store connection matrices reaching the `return` statements in the function, and $\perp$ is passed as the output of each `return` statement. At function exit, the current connection matrix is merged with the matrices in the `return-list`, to obtain the output matrix for the function.

4.3.5 Mapping and Unmapping Connection Matrices

Mapping involves preparing the input connection matrix C_e for the called procedure from the connection matrix C_m valid at the call-site. The mapping process proceeds in three steps : (i) identifying pointers abstracted by C_m, whose connection relationships can be modified by the procedure call, and representing them in the matrix C_e, (ii) computing the connection relationships generated by assigning actual parameters to their corresponding formals, and (iii) building the connection matrix C_e from C_m using the information from steps (i) and (ii).

Let the set of pointers abstracted by C_m and C_e be respectively denoted as H_m and H_e. We first identify the pointers belonging to H_m, whose connection relationships can be changed by the procedure call. They are as follows:

- **Pointers which are global in scope**. They are directly accessible to the callee, and hence their connection relationships can be arbitrarily modified by the procedure call.
- **Pointers indirectly accessible in the callee**. These pointers are local to the caller, but can be accessed by the callee through an indirect reference. Hence their connection relationships can also be easily modified by the procedure call.

- **Inaccessible local pointers**. These pointers are local to the caller and are also not indirectly accessible in the callee. However, they are connected either (i) with a global pointer, or (ii) with an indirectly accessible pointer, or (iii) with a pointer passed as a parameter. These pointers cannot be accessed or updated by the callee. However, their connection relationships can be modified by the callee, through the accessible pointers connected with them.

  In Figure 4.12 global pointer `p` and local pointer `q` are connected at the call-site `foo()`. After the call, both `p` and `q` remain connected, but only `p` remains accessible in the called procedure. In procedure `foo()`, statement `r = p` connects global pointer `r` with `p`. Next the statement `p = NULL` kills all connection relationships of `p`. On returning from the procedure, pointer `q` becomes visible again. But now it is connected with `r` and is no longer connected with `p`. Thus the connection relationships of `q` are modified by the call, through the global pointer `p`. Similarly if `q` is connected with an indirectly accessible pointer `x` or to an actual parameter $\mathtt{a}_i$, the callee can modify its connection relationships by indirectly referencing `x` or by connecting the corresponding formal $\mathtt{f}_i$ to some pointer visible in the caller.

```
bar *r, *p;
main()
{
   bar *q;
   q = (bar*) malloc();
   p = (bar*) malloc();
   q->f = p;
   /* Cm[p,p], Cm[q,q], Cm[p,q] */
   foo();
   /* Cn[q,q], Cn[r,r], Cn[q,r] */
   q->i = 5;
}

foo()
{
   ...
   /* Ce[p,p] */
   r = p;
   /* Ce[p,p], Ce[r,r], Ce[r,p] */
   p = NULL;
   /* Cx[r,r] */
}
```

Figure 4.12: Procedure Call Affects Relationships of Inaccessible Pointers

To enable accurate estimation of the effect of a procedure call on their connection relationships, the above three types of pointers should participate in the analysis of the called procedure. Hence they should be abstracted by the connection matrix C_e, and should be represented by some name in the set H_e. Global pointers can be simply represented by their name, as in the set H_m. The other two types of pointers are local to the caller, and do not have natural names in the callee. So we represent them using special compiler-generated symbolic names. If a pointer `p` is represented by a symbolic name say `1-x`, `p` is considered to be *mapped* to the name `1-x`.

Representing Out of Scope Pointers

To represent indirectly accessible pointers, we simply reuse the symbolic names generated by points-to analysis. As discussed in section 2.4.2, points-to analysis generates special symbolic names to represent all indirectly accessible variables. The symbolic names themselves are context-independent. For a given calling context, each indirectly accessible variable is mapped to one of these symbolic names, and this map information is stored in the corresponding invocation graph node.

For connection analysis, we extract the points-to map information from the invocation graph node for the current calling context. To represent indirectly accessible pointers in connection matrix C_e, we use the symbolic names they are mapped to, as per this map information. It should be noted that points-to analysis maps each indirectly accessible variable to a unique symbolic name, but can map more than one variable to a single symbolic name.

To represent inaccessible local pointers, we generate additional symbolic names, in the same fashion as points-to analysis (section 2.4.2). For each pointer that can be heap-directed at some point in the program, and is either global in scope or is a formal parameter, a unique symbolic name is generated by prefixing its name with the string '0+'. Further, if a symbolic name generated by points-to analysis happens to represent a heap-directed pointer in some calling context, another symbolic name is generated by prefixing its name with the string '0+'. Points-to analysis prefixes variable names with strings of the form 'i+' and 'i-' where i ≥ 1, to generate symbolic names. Our choice of the string '0+' thus avoids possible name clashes. Otherwise the choice is completely arbitrary.

Now if an inaccessible local pointer is connected with a global pointer, it would be mapped to the '0+'-prefixed symbolic name corresponding to the global pointer. We demonstrate the mapping of names through the example program in Figure 4.13(a). At the call-site `foo()` in `main`, local pointer `q` is connected with global pointer `p`. So it is mapped to the symbolic name `0+p`. Further, pointer `r` is connected with itself and is passed as a parameter to the formal `fr`. So it is mapped to the symbolic name `0+fr`. Finally, pointer `s` is indirectly accessible in the callee via the indirect reference `*fs`. Points-to analysis maps it to the symbolic name `1-fs`, and we reuse this mapping. Since local pointer `l` is connected with `s`, it gets mapped to the symbolic name `0+1-fs`. The complete map associations are shown in part (b) of Figure 4.13.

Each inaccessible local pointer is mapped to at most one symbolic name. So if a pointer has already been mapped, it is not mapped again. However more than one pointer may be mapped to a symbolic name. In this case, the connection relationships of the symbolic name are a merge of the relationships of the pointers it represents. This introduces imprecision. So we try to minimize the number of pointers mapped to a symbolic name, using a simple greedy strategy: if a pointer can be mapped to more than one symbolic name, we choose the one with least number of pointers mapped to it. More complicated schemes can be developed, but empirical results indicate that our simple scheme works well for real programs.

Finally, the mapping associations of both indirectly accessible and inaccessible local pointers, are recorded in the invocation graph as connection map information(cn_map_info). This information is retrieved and used while unmapping.

The complete algorithm for mapping local pointers is shown in Figure 4.15. The function cn_mapped_name is also defined. For any pointer `x`, cn_mapped_name(`x`,ign) gives the name that represents `x` in the called procedure for the calling context corresponding to the invocation graph node ign.

Building the Connection Matrix at Procedure Entry

Due to call-by-value semantics of C-language, parameter passing results in assigning actual arguments to the corresponding formal parameters. If an actual argument $\mathtt{a}_i$ is presently heap-directed, the corresponding formal $\mathtt{f}_i$ inherits its connection relationships. Thus if $\mathtt{a}_i$ is connected with a pointer $\mathtt{x}_i$ as per connection matrix C_m, we have $\mathtt{f}_i$ connected with the pointer $\mathtt{y}_i$ = cn_mapped_name($\mathtt{x}_i$,ign) in connection matrix C_e. Besides if $\mathtt{a}_i$ is connected with some pointer $\mathtt{a}_j$, which is passed as a parameter to the formal $\mathtt{f}_j$, $\mathtt{f}_i$ and $\mathtt{f}_j$ get connected in C_e. A special instance of this case is when $\mathtt{a}_i$ itself is passed as an argument to the two formals.

Only parameter assignments generate new connection relationships for a procedure call. Other relationships are simply copied over from C_m to C_e, taking into account the mapping of pointers in C_m to possibly new names in C_e. Consider a connection matrix entry C_m[r,s]. Let `y` = cn_mapped_name(`r`,ign) and `z` = cn_mapped_name(`s`,ign). If either `y` or `z` is not defined, the entry C_m[r,s] can be ignored. In this case the relationship C_m[r,s] neither generates any relationship in C_e nor would it itself be

```
bar *p;
main()
{
   bar *q, *r, *s;
   bar *l, **temp;
   p = (bar*) malloc();
   r = (bar*) malloc();
   s = (bar*) malloc();
   q = p; l = s;
   temp_0 = &s;
   /* C_m */
   foo(temp_0,r);
   /* C_n */
   q->i = 5;
}
```

```
foo(bar **fs, *fr)
{
   /* C_e */
   ...
   p->f = fr;
   temp_1 = *fs;
   fr->f = *temp_1;
   p = NULL;
   /* C_x */
}
```

(a)

Map Associations: { s $\Rightarrow$ 1-fs, l $\Rightarrow$ 0+1-fs, q $\Rightarrow$ 0+p, r $\Rightarrow$ 0+fr }

(b)

Figure 4.13: An Interprocedural Example

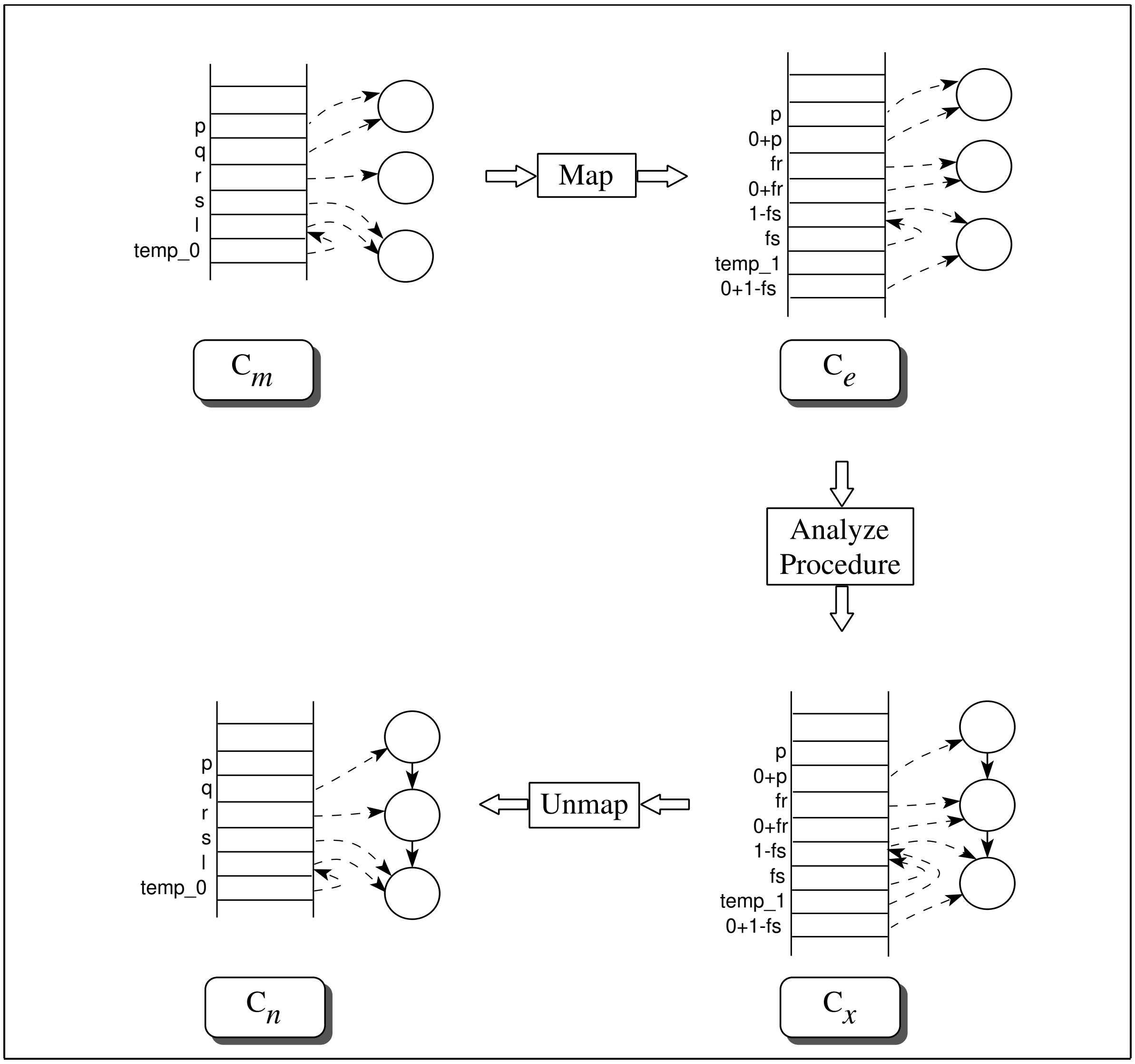


Figure 4.14: Connection Relationships for the Interprocedural Example

```
/* Functions to map names in the matrix C_m at call-site, to names in
 * the matrix C_e at procedure entry, for the call corresponding to the
 * invocation graph node ign */
fun cn_map_names(C_m,H_m,actualList,formalList,ign)
  foreach r ∈ H_m
    /* Find the name r should be mapped to in the called procedure */
    x = find_mapped_name(r,C_m,H_m,actualList,formalList,ign);
    /* r is mapped to the name denoted by x for this invocation */
    cn_mapped_name(r,ign) = x;
  return;

fun find_mapped_name(r,C_m,H_m,actualList,formalList,ign)
  if is_defined(cn_mapped_name(r,ign)) /* already mapped */
    return;
  if is_global(r) /* global pointers are mapped to themselves */
    return r;
  if is_defined(ign.ptMapInfo(r)) /* already mapped by points-to analysis */
    return (ign.ptMapInfo(r));
  /* r is an inaccessible local. Determine the set of globals, indirectly
   * accessible pointers, and actual arguments, r is connected with */
  cn_set = {};
  foreach s ∈ H_m
    if (C_m[r,s]) /* r is connected with s */
      if ((is_global(s)) or (is_defined(ign.ptMapInfo(s)) or
          (s ∈ actualList))
        /* Find the '0+'-prefixed symbolic name corresponding to s */
        s_sym = cn_symbolic_var(s);
        cn_set = cn_set ∪ {s_sym};
  if (is_empty(cn_set))
    return undefined; /* r need not be mapped */
  /* find the variable in cn_set with minimum number of vars mapped to it */
  x = min_mapped_var(cn_set);
  return (x);
```

Figure 4.15: Mapping Names From Caller to Callee

modified by the procedure call. Otherwise if `y` and `z` are already connected, they remain connected. This can happen because more than one name in C_m can be mapped to names in C_e. If C_e[y,z] is presently zero, it is set to one if C_m[r,s] is presently one. So we have: $C_e[y,z] = C_m[r,s] \vee C_e[y,z]$. The rest of the entries in C_e relating local pointers of the callee are initialized to zero.

The complete algorithm to construct the matrix C_e at the procedure entry is shown in Figure 4.16. An example is shown in Figure 4.14, for the program in Figure 4.13.

Unmap Strategy

Once the called procedure is analyzed with the input connection matrix C_e we get the output matrix C_x at the procedure exit. Next we need to unmap C_x to obtain the output matrix C_n valid after the call-site in the caller. This is a simple process. For each entry C_x[r,s] we find `y` = cn_mapped_name(`r`,ign) and `z` = cn_mapped_name(`s`,ign). If either `y` or `z` is not defined, it implies that this entry is not modified by the procedure call. So it is simply copied over to C_n, and we have $C_n[r,s] = C_m[r,s]$. Otherwise `r` and `s` would have the same relationship after the call-site, as `y` and `z` at the procedure exit, and we have $C_n[r,s] = C_x[y,z]$. The complete unmap algorithm is shown in Figure 4.17. An example unmapping is shown in Figure 4.14, for the program in Figure 4.13.

4.4 Some Important Observations

Connection analysis relies upon some important assumptions. We discuss them below.

4.4.1 Memory-Allocating Functions

Our analysis needs to know the functions that can allocate a new heap object. Since the analysis is implemented for C programs, we have identified various library functions that allocate heap memory: `malloc`, `calloc`, `valloc`, `memalign`, `mallopt`, and `alloca`. The function `alloca` allocates memory on the stack but since the allocation is dynamic we treat it as heap allocation. The function `realloc(ptr, size)` needs to be considered separately. The statement `p = realloc(q,size)` makes `p` point to the memory object pointed to by `q`, but at an offset of size `size`. For purpose of

```
/* Functions to compute input matrix C_e at procedure entry from the
 * matrix C_m at call−site corresponding to invocation graph node ign */
fun cn_map_matrix(C_m,H_m,actualList,formalList,ign) =
  /* Map names in C_m to names in C_e */
  cn_map_names(C_m,H_m,actualList,formalList,ign);
  /* Compute relationships generated by parameter assignments */
  param_gen_set = {}; /* initializing param_gen_set */
  foreach (a_i ∈ actualList) and (f_i ∈ formalList)
    if (C[a_i,a_i]) /* actual is presently heap−directed */
      /* gen_set for the assignment f_i = a_i */
      gen_set = cn_param_assign(C_m,H_m,a_i,f_i,actualList,ign);
      param_gen_set = param_gen_set ∪ gen_set;
  /* Entries in param_gen_set are set to one in C_e */
  foreach entry C[r,s] ∈ param_gen_set, C_e[r,s] = 1;
  /* Map entries in C_m to entries in C_e */
  foreach pair r,s ∈ H_m
    y = cn_mapped_name(r,ign); /* r is mapped to y */
    z = cn_mapped_name(s,ign); /* s is mapped to z */
    if ((is_defined(y)) and (is_defined(z))) /* Both r and s are mapped */
      /* C_e[y,z] could already be set to one due to
       * a previous mapping. So merge is done */
      C_e[y,z] = C_m[r,s] ∨ C_e[y,z];
  return C_e; /* Matrix at procedure entry */

fun cn_param_assign(C_m,H_m,a_i,f_i,actualList,ign)
  gen_set = {};
  foreach r ∈ H_m
    y = cn_mapped_name(r,ign); /* r is mapped to y */
    if (C_m[a_i,r]) /* r is connected with a_i at call−site */
      /* f_i gets connected with y at procedure entry */
      gen_set = gen_set ∪ {C[f_i,y]};
      foreach a_j ∈ actualList
        if (r ≡ a_j) /* a_i is connected with another actual a_j */
          gen_set = gen_set ∪ {C[f_i,f_j]}; /* f_i gets connected with f_j */
  return gen_set;
```

Figure 4.16: Mapping a Connection Matrix

```
/* Given the matrix C_m valid at the call-site, and the matrix C_x
 * valid at the procedure exit, compute the matrix C_n valid after
 * the call-site. ign is the invocation graph node for the given call */
fun cn_unmap_matrix(C_m,H_m,C_x,ign) =
  foreach pair r,s ∈ H_m
    y = cn_mapped_name(r,ign); /* r is mapped to y */
    z = cn_mapped_name(s,ign); /* s is mapped to z */
    if ((is_defined(y)) and (is_defined(z)))
      /* Both r and s were mapped to C_e. So simply copy from C_x */
      C_n[r,s] = C_x[y,z];
    else
      /* At least one of them is not mapped. So relationship remains same */
      C_n[r,s] = C_m[r,s];
  return C_n;
```

Figure 4.17: Unmapping a Connection Matrix

connection analysis we consider `p` to be pointing to the same object as `q`. Hence the analysis rule for the above statement is same as that for the statement `p = q`.

Sometimes, due to efficiency reasons, programmers allocate a big chunk of memory using one of the above library functions. Then they do their own memory management, typically using pointer arithmetic. In this case our analysis is able to identify only one heap object. All heap-directed pointers point at different offsets of this object. Hence all of them are reported to be connected with each other, providing essentially no useful information. However, even in this case, programmers typically define their own function to allocate memory from the pre-allocated chunk of memory. If the analysis is informed about this function, it can treat it same as a `malloc()` call.

Presently, our analysis recognizes only the library functions mentioned above as memory allocators. We plan to extend it to recognize user-defined memory-allocating functions, with appropriate feedback from the programmer.

It should be noted that if the call to a memory-allocating function is embedded inside another function, our analysis does not lose any precision because of its interprocedural nature. For example, suppose the user defines a function `my_malloc`:

```
void *my_malloc(int size)
{
  void *temp;
  temp = (void*) malloc(size);
  if (temp == 0)
    fatal_error("Virtual memory exhausted.");
  else
    return temp;
}
```

Now the statement `p = my_malloc(size)` will be analyzed as:

```
my_malloc(size);

p = return_my_malloc;
```

In the function call `my_malloc()`, statement `return temp` will be analyzed as:

```
return_my_malloc = temp;

return;
```

Thus after the function call, the global variable `return_my_malloc` will be pointing to the new heap object allocated by the call to `malloc()` in the function. The assignment `p = return_my_malloc` will make `p` also point to this object. Thus the statement `p = my_malloc(size)` has the same effect on connection relationships of `p` as would the statement `p = malloc(size)`.

We also review our assumptions about pointer arithmetic. They may not be valid under certain circumstances, specially when user does his own memory management. We provide a flag to the user to indicate when the assumptions may not be valid. In this case, for a statement like `p = q + k`, `p` is connected with every other pointer to ensure the *safety* of our approximation.

4.4.2 Pointers from Heap To Stack

While defining the basic analysis rules, we had assumed that heap-resident pointers do not point to locations on the stack. With this assumption, if `p` points to a heap data structure, `p->f` should also point to the same data structure. Without this assumption, `p->f` can also point to a stack location. The two cases are shown in Figure 4.18(with `N` denoting the field `f`). In part (b) of the figure, pointers `p` and `q` point to disjoint heap data structures from connection matrix point of view, as they are not linked by a heap-resident pointer. However, starting from pointer `p` one can access pointer `q`, and hence the data structure pointed to by `q`. On the contrary, we want that when `p` and `q` are not connected, `p` should not be able to access any heap location accessible from `q`, and vice versa.

Note that heap-resident pointers pointing to stack locations (henceforth we term these stack locations as *heap-pointed* locations), as such do not affect the correctness of connection analysis. Their presence just requires more careful interpretation of connection matrix information. Presently, we detect all heap-pointed locations of pointer type by using points-to information: any pointer `x`, involved in points-to relationships of the form $(heap, x, P)$, falls into this category. In the presence of heap-pointed pointers, we ensure that any analysis or transformation using connection matrix information, makes the following conservative assumption: if a heap-pointed pointer is heap-directed, the data structure pointed to by it can be potentially accessed by any other heap-directed pointer.

To enable more accurate assumptions, we plan to abstract another relationship called stack-connection. Pointer `p` is considered to be stack-connected with pointer `q`, if some heap object belonging to the data structure pointed to by `p`, has a pointer field pointing to `q` i.e. the pointer field contains the address of `q`. Thus, in Figure 4.18(b), pointer `p` is stack-connected with pointer `q`. With this abstraction, we can state the following: Two heap-directed pointers cannot access a common heap location, if they are neither connected nor stack-connected.

To accurately capture stack-connection relationships, we need to build a stack-connection matrix for each function. This matrix abstracts relationships between heap-directed pointers and pointers which are reported to be heap-pointed by points-to analysis. A pointer becomes explicitly stack-connected with another pointer due to the statement `p->link = &q`. Further if a pointer `p` is stack-connected with another pointer `q`, all pointers connected with `p` also get stack-connected with `q`. Using these two basic rules, stack-connections can be computed in the same fashion as connection relationships, both intraprocedurally and interprocedurally.

In our experimental study of a collection of C programs presented in chapter 6, we found some programs to have heap-pointed stack locations. However, none of these locations turned out to be of pointer type. We plan to analyze a larger set of programs to evaluate, how much improvement can be obtained by abstracting stack-connection relationships.

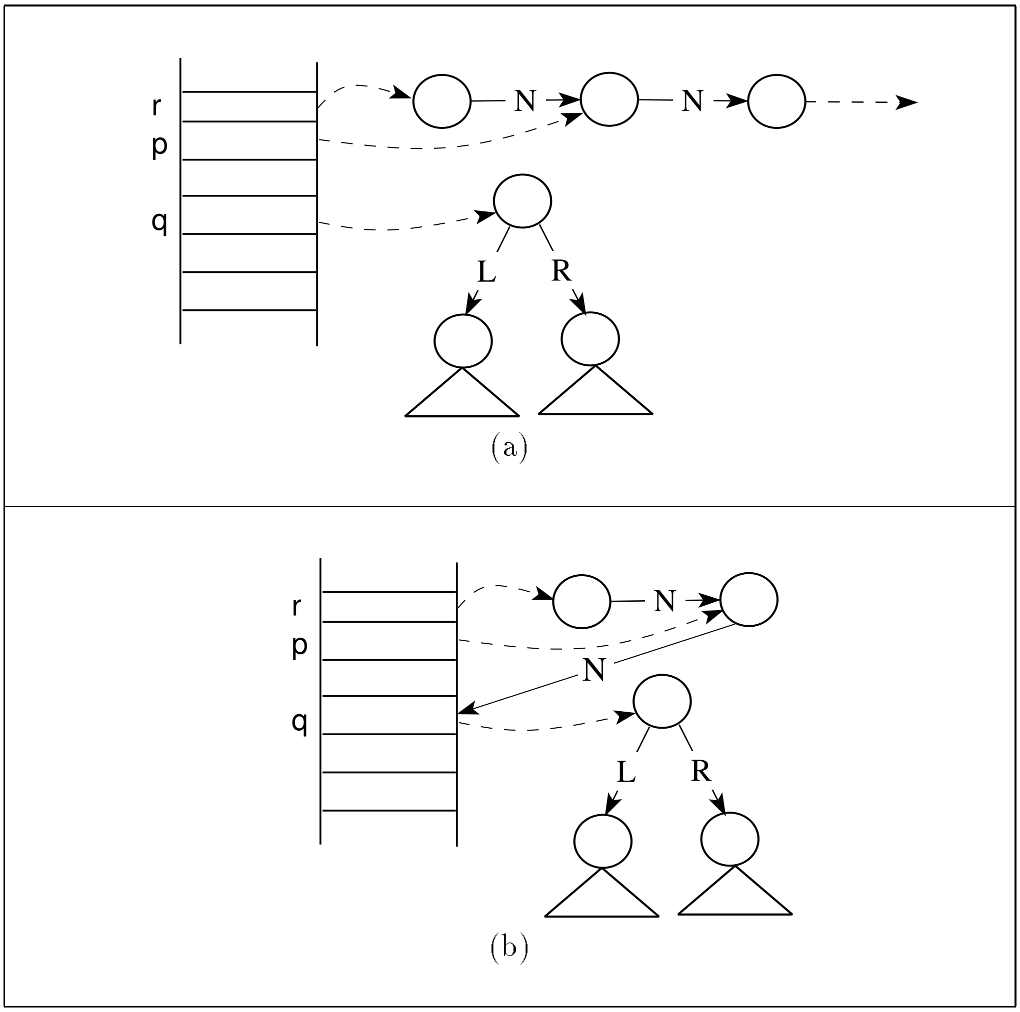


Figure 4.18: Handling Stack-connection Relationships

```
/* Given a statement S, an input matrix C which abstracts the set of
 * pointers H, and an invocation graph node ign: analyze it in the
 * calling context corresponding to ign, and return the output
 * connection matrix, flgMemo: flag to set memoization on */
fun process_stmt (S,C,H,ign,flgMemo) =
  if basic_stmt(S)
    return(process_basic_stmt(S,C,H));
  else
    case S of
      <SEQ(S1,S2)> : return(process_stmt(S2,process_stmt(S1,C,H,ign),H,ign));
      <IF(cond,thenS,elseS)> : return(process_if(cond,thenS,elseS,C,H,ign));
      <WHILE(cond,bodyS)> : return(process_while(cond,bodyS,C,H,ign));
      ...
      <(*f)(args)> : return(process_indirect_call(C,H,ign,flgMemo));
      <f(args)> : return(process_call(f.body,C,H,f.actualList,
                                      f.formalList,ign,flgMemo));
```

Figure 4.19: Overview of Analyzing a SIMPLE Statement

4.5 Summary

In this chapter we described the complete interprocedural connection analysis. A storeless approach was developed to identify if two heap-directed pointers point to the same data structure i.e. the same connected region in the heap. The analysis has been implemented on the SIMPLE intermediate representation, and an overview of the analysis is shown in Figure 4.19.

Other more sophisticated heap analysis techniques discussed in section 1.3 can also obtain this information. However connection analysis should be viewed as the first abstraction in a hierarchy of abstractions for heap analysis. Hence it cannot be directly compared with other methods. In chapter 6, we provide empirical data to demonstrate that it provides useful information in a cost-effective way, for its target domain of applications.

Chapter 5

Shape Analysis

In this chapter, we present a new heap data structure analysis called shape analysis. It follows the connection analysis in our hierarchy of heap data structure analyses. Connection analysis helps disambiguate heap accesses to completely disjoint data structures. When programs use huge aggregate structures like trees, it becomes essential to disambiguate accesses to disjoint subpieces of the same data structure. Shape analysis is designed to disambiguate heap accesses at this level, and it uses four simple abstractions which work together towards this goal. These abstractions include: (i) direction matrix, (ii) interference matrix, (iii) shape attribute, and (iv) root attribute.

The chapter is organized as follows. We introduce and motivate these abstractions in section 5.1. In the next section, we define the rules to analyze basic heap statements for collecting shape analysis information. In sections 5.3 and 5.4, we extend these rules to analyze C programs, using the SIMPLE intermediate representation. In section 5.5, the interprocedural analysis framework presented in section 2.4, is adapted to shape analysis. Finally, a brief summary of the chapter is presented.

5.1 The Abstractions

Like connection matrices, both direction and interference matrices are boolean matrices of relationships between heap-directed pointers. Direction matrix abstracts the *path existence* relationship between heap-directed pointers. Interference matrix abstracts the accessibility of a common heap object from two given heap-directed pointers.

Given any two heap-directed pointers `p` and `q`, direction matrix D captures the following program-point-specific relationships between them:

- D[p,q] = 1 : An access path *possibly* exists in the heap, *from* the heap object pointed to by pointer `p`, *to* the heap object pointed to by pointer `q`.
- D[p,q] = 0 : No access path exists *from* the heap object pointed to by `p`, *to* the heap object pointed to by `q`.

Henceforth, if D[p,q] is one, we will simply state that pointer `p` has a path to pointer `q`, and vice versa.

Given any two heap-directed pointers `p` and `q`, interference matrix I captures the following program-point-specific relationships between them:

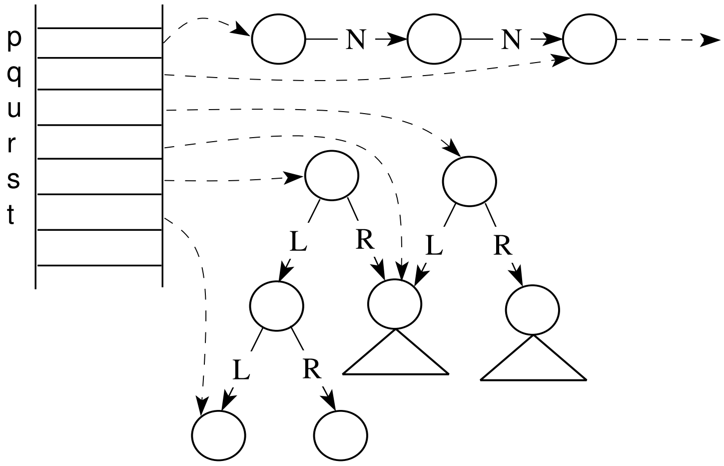
- I[p,q] = 1 : A common heap object can be *possibly* accessed, starting from pointers `p` and `q`. In this case, we state that `p` and `q` can *interfere*.
- I[p,q] = 0 : No common heap object can be accessed, starting from pointers `p` and `q`. In this case, we state that `p` and `q` do not *interfere*.

We illustrate the two abstractions in Figure 5.1. Part (a) shows the heap structure at a program point, while parts (b) and (c) show the direction and interference matrices for it. In Figures 2.17 and 3.1, we had shown the path matrix and connection matrix abstractions for a similar heap structure.

The zero in the entry D[p,r] indicates that no access path exists from the heap object pointed to by `p` to the heap object pointed to by `r`. The one in the entry D[s,t] indicates that an access path exists from pointer `s` to pointer `t`. Note that in the path matrix abstraction in Figure 2.17, the entry P[s,t] is the path expression L^2. Thus direction matrix only abstracts the *path existence* relationship between heap-directed pointers, as opposed to the precise path expressions between them, for efficiency reasons.

In Figure 5.1(a), pointers `r` and `t` point to disjoint subpieces of the same data structure. So the entry D[r,t] is zero, while in Figure 3.1 the connection matrix entry C[r,t] is set to one. Thus direction matrix collects more sophisticated information than the connection matrix abstraction.

Pointers `s` and `u` in Figure 5.1(a) point to the same data structure. However, no access path exists between them. So both the entries D[s,u] and D[u,s] are zero. But



(a) Heap Structure

	p	q	r	s	t	u
p	1	1	0	0	0	0
q	0	1	0	0	0	0
r	0	0	1	0	0	0
s	0	0	1	1	1	0
t	0	0	0	0	1	0
u	0	0	1	0	0	1

(b) Direction Matrix

	p	q	r	s	t	u
p	1	1	0	0	0	0
q	1	1	0	0	0	0
r	0	0	1	1	0	1
s	0	0	1	1	1	1
t	0	0	0	1	1	0
u	0	0	1	1	0	1

(c) Interference Matrix

Figure 5.1: Example Direction and Interference Matrices

starting from both s and u, the heap object pointed to by r can be accessed. To indicate this, the interference matrix entry I[s,u] is set to one. Note that in this case the connection matrix entry C[s,u] would also be one. However, the entry I[u,t] is zero, whereas C[u,t] would be one. Thus the interference matrix abstraction captures more precise information than the connection matrix abstraction.

Below, we note some other important characteristics of the two abstractions:

- Direction relationships are not symmetric. If an entry D[p,q] is one, it only implies the existence of an access path *from* p *to* q and not vice versa. For example, in Figure 5.1 the entry D[s,t] is one, while the entry D[t,s] is zero.
- Interference relationships like connection relationships are symmetric. If an entry I[p,q] is one, the entry I[q,p] is also one. Both of them imply that a common heap object can be accessed starting from pointers p and q. The interference relationships shown in Figure 5.1(c), illustrate this property. The symmetric property reduces the storage requirements for interference matrix by half, in the actual implementation.
- Interference relationships form a superset of direction relationships. If an access path exists from pointer p to q, then p and q can both access the heap object pointed to by q. Thus, if the entry D[p,q] is one, the entries I[p,q] and I[q,p] are also one. As already discussed, the converse is not necessarily true. In Figure 5.1 I[s,u] is one, while both D[s,u] and D[u,s] are zero.
- If the entry D[p,p] is set to one, it implies that pointer p is presently heap-directed.

The main motivation behind estimating direction relationships, is to estimate the shape of the data structures built by the program. The shape of a data structure can be *Tree*, *Dag* or *Cycle*. A data structure is considered to be a *Tree*, if there is a unique (possibly empty) (access) path between any two nodes (heap objects) belonging to it. It is considered to be a *Dag* (directed acyclic graph), if there can be more than one path between any two nodes in the data structure, but there is no path from a node to itself (i.e. it is acyclic). If the data structure contains a node having a path to itself, its shape is considered to be *Cycle*. Note that, as lists are a special case of tree data structures, their shape is also considered as Tree.

We now demonstrate how direction relationships help estimate the shape of heap data structures. A *shape* attribute is associated with each heap-directed pointer. For a pointer p, the shape of the data structure *accessible* from it, is abstracted as *p.shape*.

In Figure 5.2, initially we have both p.shape and q.shape as Tree. Further we have D[p,q] as one, as there exists a path between `p` and `q` through the `next` link. The statement `q->prev = p`, sets up a path from `q` to `p` through the `prev` link, and creates a cycle between heap objects pointed to by `p` and `q`. After the statement, we have D[p,q] = 1, D[q,p] = 1, p.shape = Cycle and q.shape = Cycle.

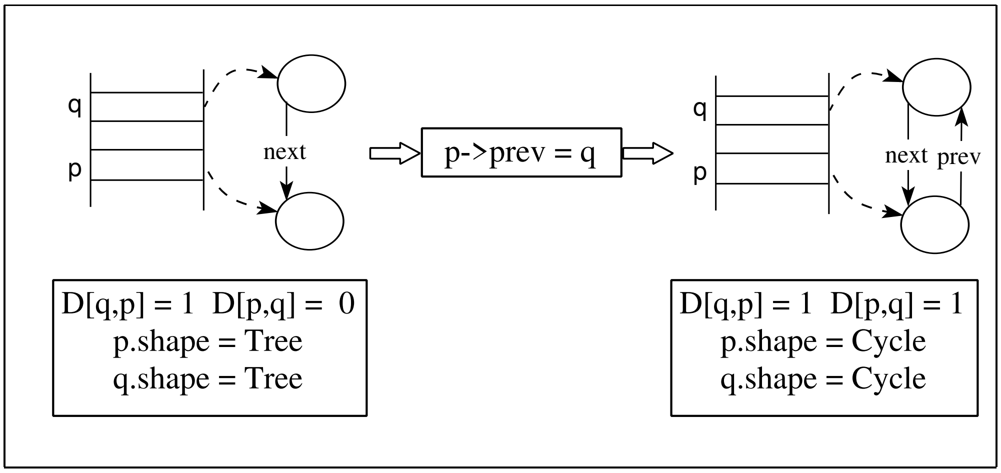


Figure 5.2: Example Demonstrating Shape Estimation

It should be noted that for a heap-directed pointer `p`, p.shape only abstracts the shape of the data structure *accessible* from `p` and not the overall shape of the data structure pointed to by `p`. For example, in Figure 5.3, the overall shape of the data structure pointed to by `p` and `q` is Dag. However, if only the part of the data structure accessible from `p` or `q` is considered, its shape is Tree. So we have both p.shape and q.shape as Tree.

Knowledge about the shape of the data structure accessible from a heap-directed pointer, provides crucial information for disambiguating heap accesses originating from it. For a pointer `p`, if p.shape is Tree, then any two accesses of the form `p->f` and `p->g` will always lead to disjoint subpieces of the tree (assuming `f` and `g` are distinct fields). If p.shape is Dag, then two distinct field accesses `p->f` and `p->g` can lead to a common heap object, as in Figure 5.4. However, if the data structure is traversed using only one link, for example using only the `f` link in Figure 5.4, a distinct heap object will be visited by each access. Moreover, if a dag-like structure is traversed using a given sequence of links, every subsequence would visit a distinct node. This information can be used to disambiguate heap accesses across different iterations of a loop traversing such a data structure. Finally, if p.shape happens to be

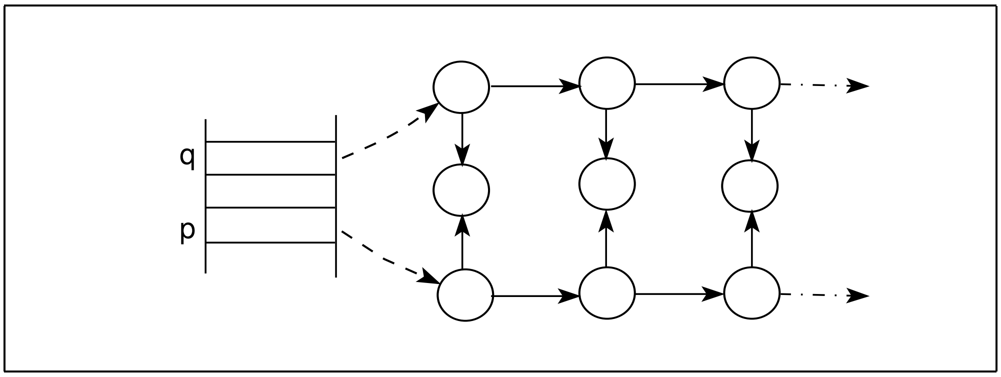


Figure 5.3: Estimating Shape with *accessibility* Criterion

Cycle, we have effectively no information to disambiguate heap accesses originating from `p`.

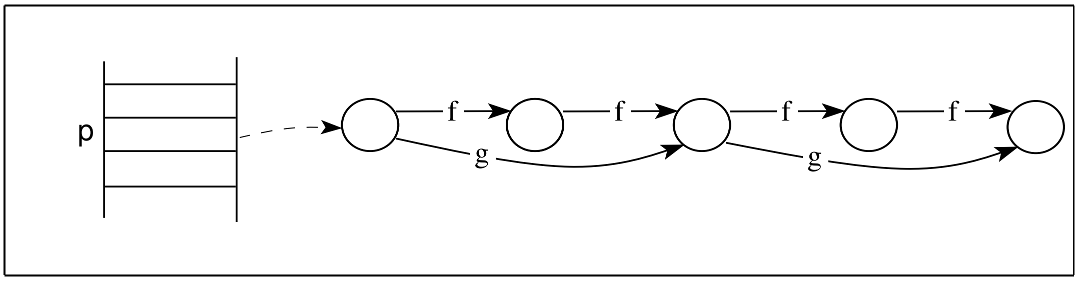


Figure 5.4: Acyclicity of Dag Data Structures

Thus, our goal is to identify tree-like and dag-like data structures, and to retain this information as long as possible, during the analysis. To be able to collect more accurate shape information, we also abstract a *root* attribute for each heap-directed pointer [1]. We have *p.root* as True, when the heap object pointed to by `p` has no incoming links into it. Otherwise, it is assigned False. Note that a stack-resident pointer pointing to a heap object, is not considered as an incoming link. In Figure 5.1 the root attribute of pointers `p`, `u` and `s` is True, while for pointers `q`, `r` and `t` it is False.

[1]It turned out that the root attribute does not prove to be of direct help in shape analysis. However, we still abstract it as it can be useful for other purposes like program understanding and debugging.

The interference matrix information is used to improve the precision of shape analysis, as shown in the next section. However, to accurately estimate interference relationships themselves, direction relationships are required. So the two abstractions need to be computed simultaneously. Interference information by itself, can be used to determine if heap accesses originating from two heap-directed pointers can lead to a common heap object. This information is more accurate than connection information, as the latter only identifies if two heap accesses can lead to a common data structure.

Having defined and motivated the abstractions, we now present the analysis rules to compute them.

5.2 Analyzing Basic Heap Statements

In this section, we present the rules to estimate the effect of the eight basic heap statements shown in Figure 3.2, on direction and interference matrix information. The overall structure of the analysis is shown in Figure 5.5(a). We have the direction and interference matrices D and I at program point `x`, before the given statement. We wish to compute the matrices D_n and I_n at program point `y`. Additionally, we have the attribute matrix A, where for a pointer `p`, A[p.shape] gives its shape attribute, and A[p.root] gives its root attribute. The attribute matrix after the statement is represented as A_n.

For each statement, we compute the sets of direction and interference relationships it kills and generates. Using these sets, the new matrices D_n and I_n are computed as shown in Figure 5.5(b). We also compute the sets of heap-directed pointers H_s and H_r, whose shape and root attributes can be changed by the given statement. Another attribute matrix A_c is used to store the changed attributes of pointers belonging to the sets H_s and H_r. The attribute matrix A_n is then computed using the matrices A and A_c as shown in Figure 5.5(b).

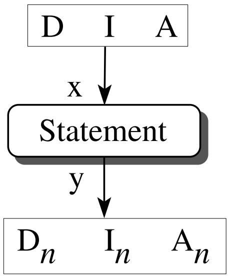
Let H be the set of pointers relevant to heap analysis, for the procedure containing the given statement. This is the set of pointers whose relationships/attributes are abstracted by matrices D, I and A. For the analysis of basic heap statements, we assume that pointers can only point to heap objects or to `NULL`, as we assumed for connection analysis. We also assume that updating an interference matrix I[p,q], implies identically updating the entry I[q,p]. This assumption is rendered valid due to the symmetric nature of interference relationships.

We now present the analysis rules for the eight basic heap statements:

`p = malloc()` : Pointer `p` points to a newly allocated heap object. All its existing relationships get killed. Pointer `p` has an empty path to itself, and it also interferes with itself. This statement can change the attributes of only pointer `p`. Since the newly allocated object pointed to by `p` has no incoming or outgoing links, it is both a root and a Tree.

These observations can be summarized in rule format as follows:

D_kill_set = { D[p,s] | s $\in$ H $\wedge$ D[p,s] } $\cup$ { D[s,p] | s $\in$ H $\wedge$ D[s,p] }
I_kill_set = { I[p,s] | s $\in$ H $\wedge$ I[p,s] }
malloc_gen_set = { D[p,p], I[p,p] }
H_s = { p } H_r = { p } A_c[p.shape] = Tree A_c[p.root] = True



(a)

Build the new matrices
$\forall$ r,s $\in$ H, D_n[r,s] = D[r,s], I_n[r,s] = I[r,s]
$\forall$ s $\in$ H, A_n[s.shape] = A[s.shape]
$\forall$ s $\in$ H, A_n[s.root] = A[s.root]
Delete killed relationships
$\forall$ entries D[r,s] $\in$ kill_set, D_n[r,s] = 0
$\forall$ entries I[r,s] $\in$ kill_set, I_n[r,s] = 0
Add generated relationships
$\forall$ entries D[r,s] $\in$ gen_set, D_n[r,s] = 1
$\forall$ entries I[r,s] $\in$ gen_set, I_n[r,s] = 1
Update attributes of affected pointers
$\forall$ s $\in$ H_r, A_n[s.root] = A_c[s.root]
$\forall$ s $\in$ H_s, A_n[s.shape] = A_c[s.shape]

(b)

Figure 5.5: The Overall Structure of the Analysis

Note that having D[p,p] in the gen set here simply implies that `p` presently points to a heap object. It does not imply that a cyclic data structure becomes accessible from `p` after this statement. In that case, we should also have A_c[p.shape] = Cycle.

The next five basic heap statements (`p = q`, `p = q->f`, `p = &(q->f)`, `p = q op k` and `p = NULL`) update the stack-resident pointer `p`, and make it point to a new heap object. They kill all existing relationships of `p`, and only change the attributes of pointer `p`. So the kill set and the sets H_s and H_r for all these statements, are same as that for the statement `p = malloc()`. Below, we present the rules to calculate the gen set and the matrix A_c for these five statements.

`p = q` : Pointer `p` now points to the same heap object as `q`. So it would have paths from/to all the pointers `q` has paths from/to. Similarly, it would interfere with all the pointers `q` interferes with. Finally, the same data structure would be accessible from `p` as from `q`. Thus, all the existing relationships of pointer `p` get killed, and it simply inherits the relationships and attributes of pointer `q`. In case `q` presently points to `NULL`, `p` would also point to `NULL` after the statement. So we have D[p,p] and I[p,p] in the gen set, only if D[q,q] and I[q,q] are presently set to one (implying that `q` does not point to `NULL`). Thus, we have the following rule for this statement:

$$
\begin{aligned}
&\text{D\_gen\_set\_1} = \{\ D[p,s] \mid s \in H \wedge D[q,s]\ \} \\
&\text{D\_gen\_set\_2} = \{\ D[s,p] \mid s \in H \wedge D[s,q]\ \} \cup \{\ D[p,p] \mid D[q,q]\ \} \\
&\text{I\_gen\_set} = \{\ I[p,s] \mid s \in H \wedge I[q,s]\ \} \cup \{\ I[p,p] \mid I[q,q]\ \} \\
&\text{stack\_lhs\_stack\_rhs\_gen\_set} = \text{D\_gen\_set\_1} \cup \text{D\_gen\_set\_2} \cup \text{I\_gen\_set} \\
&A_c[\text{p.shape}] = A_n[\text{q.shape}] \quad A_c[\text{p.root}] = A_n[\text{q.root}]
\end{aligned}
$$

Since this statement simply copies one stack-resident pointer to another stack-resident pointer, we name its overall gen_set as stack_lhs_stack_rhs_gen_set.

As discussed in section 3.2, the statements `p = q op k` and `p = &(q->f)` are equivalent to the statement `p = q`, for purposes of heap analysis. So these statements are analyzed using the same rules as developed for the statement `p = q`. The statement `p = NULL` kills all relationships of `p` and does not generate any new relationships. Since `p` points to `NULL` after the statement, shape and root attributes are not relevant to it. As a default case, its attributes are set as Tree and True respectively. We now present the analysis rules for the statement `p = q->f`.

`p = q->f` : Pointer `p` now points to the heap object accessible from pointer `q` through the link `f`, as shown in Figure 5.6.[2] All the existing relationships of `p` get killed. All pointers having a path to `q` (which includes `q` itself), now have a path to `p` through the link `f`. In Figure 5.6, pointers `u`, `v` and `q` will have a path to `p` after the statement. This set of newly generated direction relationships, can be summarized as follows:

$$\text{D\_gen\_set\_1} = \{\ D[s,p] \mid s \in H \wedge D[s,q]\ \}$$

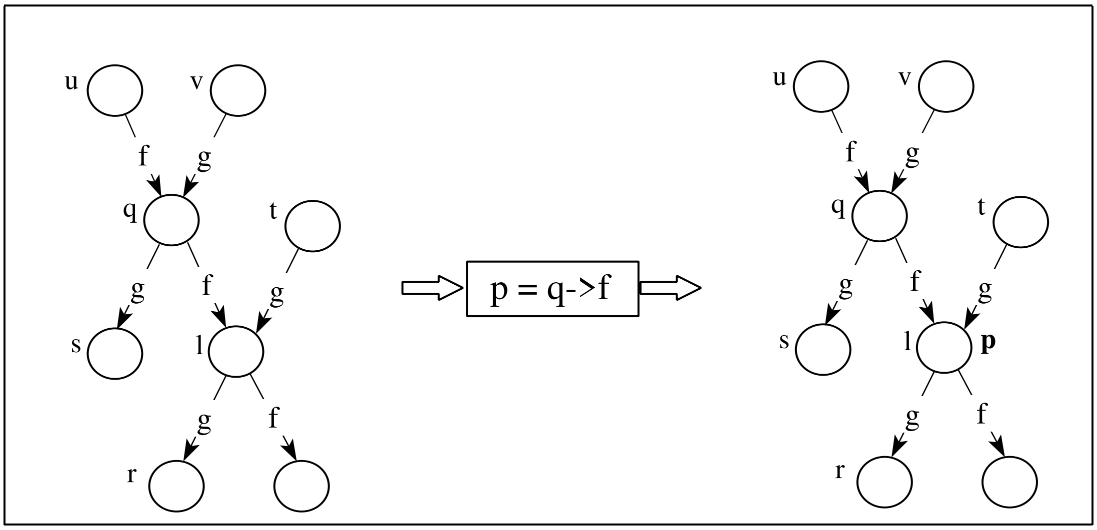


Figure 5.6: Analyzing Basic Heap Statement `p = q->f`

Some pointers `q` has paths to, may also have paths back to `p`, after the statement `p = q->f`. In Figure 5.6, before the statement, A[q.shape] is Tree and `q` has paths to pointers `l`, `r` and `s`. After the statement, `r` and `s` will not have paths back to `p`, while `l` will have an empty path to `p`. Further, in Figures 5.7(a) and (b), where A[q.shape] is respectively Dag and Cycle, `q` has a path to `x` before the statement, and `x` will have a path to `p` after the statement `p = q->f`. Thus, to ensure the correctness of the analysis we err conservatively, and assume all pointers `q` has paths to, to have a path to `p` in the matrix D_n. The set of new direction relationships generated for this case can be summarized as follows:

$$\text{D\_gen\_set\_2} = \{ \text{ D[s,p]} \mid \text{s} \in \text{H} \wedge \text{D[q,s]} \}$$

In Figure 5.6, pointer `t` has a path to pointer `p` after the statement. Now pointer `t` does not have any direction relationships with pointer `q` (Figure 5.8(a)), but it has an interference relationship with `q`, as shown in in the interference matrix I in Figure 5.8(b). Thus, any pointer that interferes with `q` can potentially have a path to `p`, giving the following gen_set:

$$\text{D\_gen\_set\_3} = \{ \text{ D[s,p]} \mid \text{s} \in \text{H} \wedge \text{I[s,q]} \}$$

[2]In this Figure, for sake of clarity we have simply labeled each node with the stack-resident pointer that points to it, instead of explicitly representing the stack.

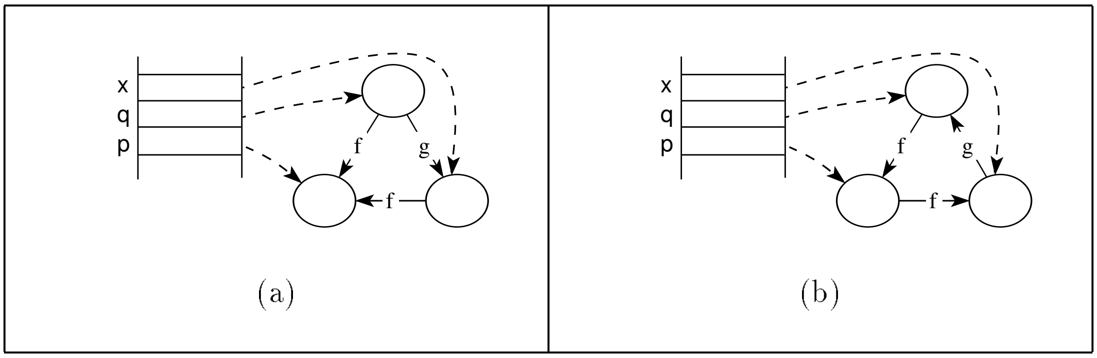


Figure 5.7: Shape Attribute and Direction Relationships

Now, all pointers `q` has paths from/to, also interfere with `q`. Thus, D_gen_set_1 and D_gen_set_2 are subsets of D_gen_set_3, and the set of direction relationships generated because of pointer `p` having paths *from* other pointers, is simply D_gen_set_3.

Pointer `p` will have a path to all pointers `q` has a path to, via the link `f`. From the direction matrix, we can find *all* the pointers `q` has a path to, but cannot identify the pointers `q` has a path to via a specific link. So, we assume `p` to have a path to all the pointers `q` has a path to. This makes the information collected conservative, but ensures its correctness. In Figure 5.6, after the statement, pointer `p` is reported to be having paths to pointers `l`, `r` and `s`, where the path from `p` to `s` is spurious.

Note that `q` has a path to itself , so `p` should also be reported to have a path to `q` after the statement. However, this would not be true, if the the data structure accessible from `q` is acyclic (i.e. A[q.shape] = Tree or Dag). In this case, any path originating from `q` cannot return to it, and `q` can only have an empty path to itself. In Figure 5.6, A[q.shape] is Tree, hence `p` is not reported to be having a path to `q`. Further, in this case if another pointer say `m` points to the same heap object as `q`, `p` should not be reported to be having a path to `m` either. To enable this, we need to keep track of pointers definitely pointing to the same heap object. We track them in the implementation, but do not consider them here for sake of clarity.

Thus, the set of direction relationships generated due to pointer `p` having paths *to* other pointers, can be summarized as follows:

$$\begin{aligned} \text{D\_gen\_set\_4} = \{\ & \text{D[p,s]} \mid \text{s} \in \text{H} \land \text{s} \neq \text{q} \land \text{D[q,s]}\ \} \cup \\ \{\ & \text{D[p,q]} \mid \text{A[q.shape]} = \text{Cycle}\ \} \end{aligned}$$

After the statement, pointer `p` can potentially interfere with all the pointers `q` presently interferes with. So we have the following set of newly generated interference relationships:

$$\text{I\_gen\_set} = \{\ \text{I[p,s]} \mid \text{s} \in \text{H} \wedge \text{I[q,s]}\ \}$$

Finally, `p` is reported as heap-directed (i.e D[p,p] and I[p,p] are set to one), only if `q` is presently heap-directed. To ensure this we have the following set of relationships:

$$\text{D\_gen\_set\_5} = \{\ \text{D[p,p]} \mid \text{D[q,q]}\ \} \cup \{\ \text{I[p,p]} \mid \text{I[q,q]}\ \}$$

The overall gen set for this statement, named stack_lhs_heap_rhs_gen_set as the statement copies a heap-resident pointer to a stack-resident pointer, is as follows:

$$\text{stack\_lhs\_heap\_rhs\_set} = \text{D\_gen\_set\_3} \cup \text{D\_gen\_set\_4} \cup \text{D\_gen\_set\_5} \cup \text{I\_gen\_set}$$

The new direction and interference matrices D_n and I_n valid after the analysis of the statement `p = q->f` with the heap structure illustrated in Figure 5.6, are shown in parts (c) and (d) of Figure 5.8. It can be noticed that many spurious direction and interference relationships can be generated during the analysis of this statement. To avoid them, more sophisticated path relationships like path expressions [HN90] need to be captured, incurring greater cost in analysis. Thus, there is a trade-off between the cost of the analysis and the quality of information it collects.

However, as already mentioned, the main focus of shape analysis, is to identify the shape of the underlying data structures. The statement `p = q->f`, despite potentially introducing several spurious direction/interference relationships, does not affect the shape of the data structure accessible from `q`. It only gives a new name to one of the heap objects belonging to this data structure. Now, since the data structure accessible from `p` is a subpiece of the data structure accessible from `q`, `p` simply inherits the shape attribute of `q`.

It is possible that A[q.shape] is Cycle, while the shape of the structure accessible from `q` via the `f` link is Tree. However, we cannot detect this from the information available, and have A_n[p.shape] also as Cycle. But if A[q.shape] is Tree, we do not lose the tree attribute, and if it is Dag we still preserve the acyclic property of the data structure accessible from `p`. Note that, if we simply deduce the shape attribute of `p` from its direction relationships after the statement, we may lose its Tree or Dag attribute. Thus, separately abstracting the shape attribute, proves to be critical in identifying tree-like and dag-like data structures. Finally, the node (heap object) pointed to by `p`, cannot be a *root* node, as the link `f` from `q` leads into it.

Thus, we have the following attribute matrix A_c:

$$A_c[\text{p.shape}] = A[\text{q.shape}] \quad A_c[\text{p.root}] = \text{False}$$

	l	p	q	r	s	t	u	v
l	1	0	0	1	0	0	0	0
p	0	0	0	0	0	0	0	0
q	1	0	1	1	1	0	0	0
r	0	0	0	1	0	0	0	0
s	0	0	0	0	1	0	0	0
t	1	0	0	1	0	1	0	0
u	1	0	1	1	1	0	1	0
v	1	0	1	1	1	0	0	1

(a) Direction Matrix D

	l	p	q	r	s	t	u	v
l	1	0	1	1	0	1	1	1
p	0	0	0	0	0	0	0	0
q	1	0	1	1	1	1	1	1
r	1	0	1	1	0	1	1	1
s	0	0	1	0	1	0	1	1
t	1	0	1	1	0	1	1	1
u	1	0	1	1	1	1	1	1
v	1	0	1	1	1	1	1	1

(b) Interference Matrix I

	l	p	q	r	s	t	u	v
l	1	1	0	1	0	0	0	0
p	1	1	0	1	1	0	0	0
q	1	1	1	1	1	0	0	0
r	0	1	0	1	0	0	0	0
s	0	1	0	0	1	0	0	0
t	1	1	0	1	0	1	0	0
u	1	1	1	1	1	0	1	0
v	1	1	1	1	1	0	0	1

(c) Direction Matrix D_n

	l	p	q	r	s	t	u	v
l	1	1	1	1	0	1	1	1
p	1	1	1	1	1	1	1	1
q	1	1	1	1	1	1	1	1
r	1	1	1	1	0	1	1	1
s	0	1	1	0	1	0	1	1
t	1	1	1	1	0	1	1	1
u	1	1	1	1	1	1	1	1
v	1	1	1	1	1	1	1	1

(d) Interference Matrix I_n

Figure 5.8: Matrices For the Heap Structure Shown in Figure 5.6

`p->f = NULL` : This statement breaks the link `f` emanating from the heap object pointed to by `p`. Thus, after the statement, `p` should no longer have paths to pointers, it presently has paths to, exclusively via the link `f`. As already discussed, this information cannot be obtained from direction/interference matrices. So no relationships can be killed. Further, the statement does not generate any new relationships.

The shape attribute of pointer `p` may change, if this statement disconnects the subpiece of the data structure, due to which A[p.shape] is Dag or Cycle. Similarly, the heap object into which the `f` link presently leads from `p`, may become a root again once the link is broken. But the direction/interference information does not suffice to detect such cases, and we err conservatively leaving the attributes unchanged. Note that due to the lack of precise kill information for this statement, if a tree-like structure temporarily becomes dag-like or cyclic, and becomes a tree again (e.g. when swapping the children of a tree), our analysis would continue to report its shape as Dag or Cycle.

`p->f = q` : This statement first breaks the link `f`, and then resets it thereby linking the heap object pointed to by `p`, to the heap object pointed to by `q`, as shown in Figure 5.9. As already discussed, the relationships killed on breaking the link `f`, cannot be obtained with the information available. However, resetting the link `f` results in generating some new relationships and modifying the attributes of several pointers, as discussed below.

All pointers having a path to `p` (including `p` itself), will now have a path to `q` via the link `f`. Further, these pointers will have paths to all pointers `q` has paths to. In Figure 5.9, pointers `u`, `v` and `p` will have paths to pointers `q`, `r` and `s` after the statement. Thus, the set of direction relationships generated can be summarized as follows:

$$\text{D\_gen\_set} = \{\ \text{D[r,s]} \mid \text{r,s} \in \text{H} \wedge \text{D[r,p]} \wedge \text{D[q,s]}\ \}$$

In Figure 5.9, pointer `q` interferes with pointer `t`, before the statement. After the statement, pointers `u` and `v` will also interfere with `t`. This demonstrates that all pointers having a path to `p`, can potentially interfere with all pointers `q` interferes with.

Thus, we get the following set of new interference relationships:

$$\text{I\_gen\_set} = \{\ \text{I[r,s]} \mid \text{r,s} \in \text{H} \wedge \text{D[r,p]} \wedge \text{I[q,s]}\ \}$$

The overall gen set for this statement, named heap_lhs_stack_rhs_gen_set as the statement copies a stack-resident pointer to a heap-resident pointer, is as follows:

$$\text{heap\_lhs\_stack\_rhs\_gen\_set} = \text{D\_gen\_set} \cup \text{I\_gen\_set}$$

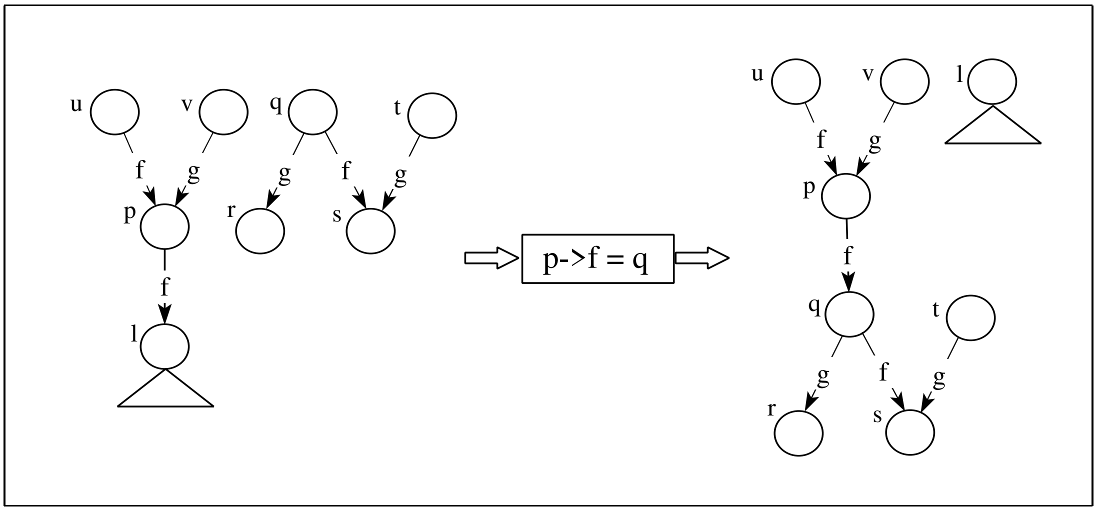


Figure 5.9: Analyzing Basic Heap Statement `p->f = q`

This statement can considerably affect the shape attribute of pointers, which have direction relationships with pointers `p` and `q`. We can have the following situations, depending on the current attributes and direction relationships of pointers `p` and `q`:

<u>Pointer `q` already has a path to `p` (D[q,p] = 1)</u> : After the statement `p->f = q`, `p` will also have a path back to `q`. Thus, a cycle will be generated between `p` and `q`, as shown in Figure 5.10(a). This cycle will be accessible from all pointers that presently have a path to `p` or `q` (including `p` and `q` themselves), and the shape attribute of all these pointers will become Cycle. In Figure 5.10(a), the shape attribute of pointers `u`, `v`, `p`, and `q` becomes Cycle after the statement. We summarize this case as follows:

$$\mathrm{H}_s = \{ \mathrm{s} \mid \mathrm{s} \in \mathrm{H} \wedge (\mathrm{D[s,q]} \vee \mathrm{D[s,p]}) \}$$
$$\forall \mathrm{s} \in \mathrm{H}_s,\ \mathrm{D[q,p]} \Rightarrow \mathrm{A}_c[\mathrm{s.shape}] = \mathrm{Cycle}$$

If the above situation does not arise, we have the following possibilities:

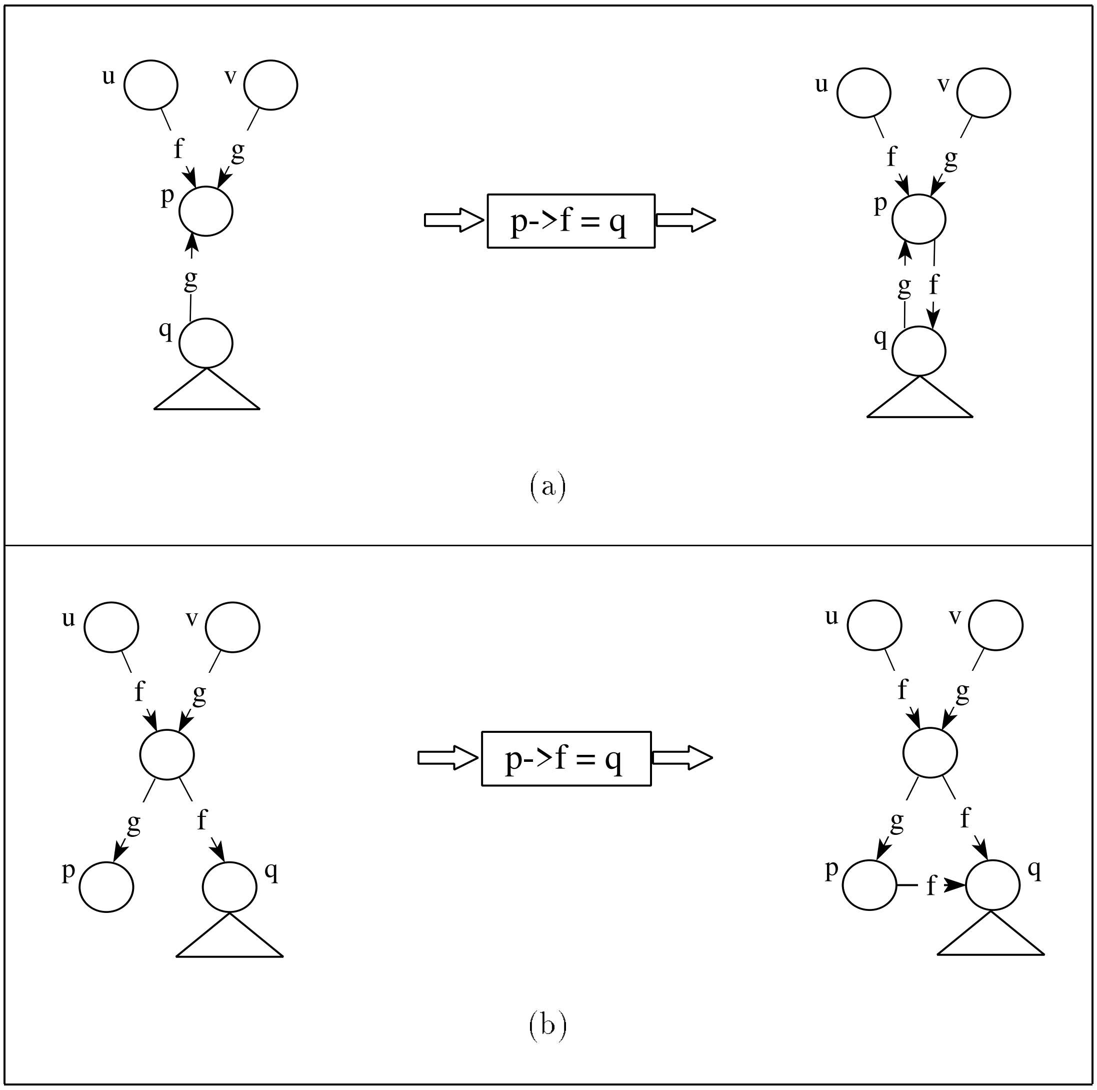


Figure 5.10: Direction Relationships Impacting Shape Attribute

A[q.shape] = Tree : In this case another tree-like structure becomes accessible from all the pointers that presently have a path to `p`. If the data structures pointed to by `p` and `q` are initially completely disjoint, then the statement simply connects a tree substructure to the data structure pointed to by `p` and does not affect the shape attribute of any pointer. Figure 5.9 illustrates this case. Otherwise the shape attribute of pointers that initially have a path to `p` and also interfere with `q`, becomes Dag (if it is presently Tree). Pointers `u` and `v` in Figure 5.10(b) fall in this category. These pointers presently have paths to both `p` and `q`. After the statement, they will have additional paths to `q` via `p`. Thus, the shape attribute of `u` and `v` will become Dag, if it is presently Tree. The same would hold true, if pointers `u` and `v` have paths to `p`, and to some pointer to which `q` has a path to (i.e. `u` and `v` have a path to `p` and interfere with `q`).

Finally, if the shape attribute of such a pointer is already Dag or Cycle, it remains unchanged. In other words, the shape attribute of these pointers, becomes the merge of their current attribute and the Dag attribute, where the merge operator $\bowtie$ for the shape attribute is defined as follows:

$\bowtie$	*Tree*	*Dag*	*Cycle*
Tree	*Tree*	*Dag*	*Cycle*
Dag	*Dag*	*Dag*	*Cycle*
Cycle	*Cycle*	*Cycle*	*Cycle*

This case can be formally summarized as follows:

$$\mathrm{H}_s = \{\ \mathrm{s} \mid \mathrm{s} \in \mathrm{H} \wedge (\mathrm{I[s,q]} \wedge \mathrm{D[s,p]})\ \}$$
$$\forall\ \mathrm{s} \in \mathrm{H}_s,\ ((\neg \mathrm{D[q,p]}) \wedge (\mathrm{A[q.shape]} = \mathrm{Tree})) \Rightarrow$$
$$\mathrm{A}_c\mathrm{[s.shape]} = \mathrm{A[s.shape]} \bowtie \mathrm{Dag}$$

A[q.shape] $\neq$ Tree : In this case, the shape attribute of all pointers that have path to `p` is merged with the shape attribute of `q`. This is required because the data structure accessible from `q`, will also become accessible from all these pointers after the statement. We summarize the case as follows:

$$\mathrm{H}_s = \{\ \mathrm{s} \mid \mathrm{s} \in \mathrm{H} \wedge \mathrm{D[s,p]}\ \}$$
$$\forall\ \mathrm{s} \in \mathrm{H}_s,\ ((\neg \mathrm{D[q,p]}) \wedge (\mathrm{A[q.shape]} \neq \mathrm{Tree})) \Rightarrow$$
$$\mathrm{A}_c\mathrm{[s.shape]} = \mathrm{A[s.shape]} \bowtie \mathrm{A[q.shape]}$$

Finally, the heap object pointed to by `p` cannot be a root anymore, as the link `f` from `p` will lead into it. So, the root attribute of `q` becomes False after the statement. Further, any other pointer that points to the same heap object as `q`, should also have its root attribute as False, after the statement. To capture such pointers, we make the root attribute of all pointers `q` has paths to, as False. Note that this does not introduce any imprecision, as no pointer to which another pointer can have a path, can have its root attribute as True. Thus, we have the following rule:

$$H_r = \{ \text{s} \mid \text{s} \in \text{H} \wedge \text{D[q,s]} \}$$
$$\forall \text{ s} \in H_r,\ A_c[\text{s.root}] = \text{False}$$

This completes the analysis of the statement `p->f = q`, and the analysis of basic heap statements.

From the rules presented above, it can be noticed that a considerable number of spurious direction and interference relationships can be introduced during the analysis. This happens because of the following two reasons:

- We only abstract boolean relationships between heap-directed pointers, as opposed to precise path relationships between them [HN90]. This makes the analysis more efficient, but less precise.
- Our analysis is *not* restricted to analyze programs that only build tree and dag-like data structures, as in [HN90]. Due to the requirement to handle more complex data structures, it tends to become more conservative.

However, as already mentioned, the main focus of our analysis is to be able to identify tree and dag-like data structures built by the program under analysis. Direction and interference relationships are basically computed to achieve this goal. Empirical results presented in chapter 6, indicate that our analysis provides effective information for a broad range of programs.

5.3 Analyzing Basic SIMPLE Statements

Shape analysis of a basic SIMPLE statement S, is also based on the S-locations represented by the variable references on lhs(S) and rhs(S) (similar to connection analysis). The overall algorithm for analyzing a basic SIMPLE statement is presented in Figure 5.11. In the following paragraphs, we describe in detail the rules for computation of kill and gen sets and the estimation of new attributes.

```
/* Analyze statement S with input matrices D, I and A *
 * with H as the set of pointers abstracted by them */
fun process_basic_stmt(S,D,I,A,H) =
  if (! is_pointer_type(S) ) /* not a pointer assignment */
    return([D,I,A])
  /* All relationships of definite locations are killed */
  kill_set = { D[x,z], D[z,x], I[x,z] | (x, D) ∈ S−locations(lhs(S)) ∧ x,z ∈ H }
  gen_set = build_gen_set(S,D,I,A,H) /* Build the gen set */
  [H_r,H_s,A_c] = find_mod_attr(S,D,I,A,H) /* Estimate new attributes */

 /* Build the new Matrices */
 ∀ r,s ∈ H, D_n[r,s] = D[r,s], I_n[r,s] = I[r,s]
 ∀ s ∈ H, A_n[s.shape] = A[s.shape], A_n[s.root] = A[s.root]
 /* Delete killed relationships */
 ∀ entries D[r,s] ∈ kill_set, D_n[r,s] = 0
 ∀ entries I[r,s] ∈ kill_set, I_n[r,s] = 0
 /* Add generated relationships */
 ∀ entries D[r,s] ∈ gen_set, D_n[r,s] = 1
 ∀ entries I[r,s] ∈ gen_set, I_n[r,s] = 1
 /* Reset the attributes of definitely updated pointers to default values */
 ∀ (x, D) ∈ S−locations(lhs(S)), A_n[x.shape] = Tree, A_n[x.root] = True
 /* Update Attributes of Affected Pointers */
 ∀ s ∈ H_s, A_n[s.shape] = A_c[s.shape] ⋈ A_n[s.shape]
 ∀ s ∈ H_r, A_n[s.root] = A_c[s.root] ⋈ A_n[s.root]

 return([D_n,I_n,A_n])
```

Figure 5.11: Analyzing a Basic SIMPLE Statement

5.3.1 Computing Kill Set

If the set S-locations(lhs(S)) consists of a (single) definite S-location (x, D), all the direction and interference relationships of `x` are killed. If this set consists of possible S-locations, no relationships can be killed. Thus, we have the following kill set for any given SIMPLE statement S (also shown in Figure 5.11):

kill_set(S) = { D[x,z], D[z,x], I[x,z] | $(x, D) \in$ S-locations(lhs(S)) $\wedge$ x,z $\in$ H }

5.3.2 Computing Gen Set

To compute the gen set for the SIMPLE statement S, we need to consider all the S-statements (section 4.1.2) it can generate. The gen set for S would be the union of the gen sets of its S-statements. We had noted during connection analysis (chapter 4), that each S-statement corresponds to one of the eight basic heap statements, depending upon the S-locations it represents. The gen set for an S-statement can thus be computed using the gen set computation rule for its corresponding basic heap statement. Based on this strategy, the complete rules for computing the gen set of a basic SIMPLE statement are formulated in Figures 5.12 and 5.13.

Given any two S-locations S-lloc and S-rloc (for the SIMPLE statement S), the gen set for the S-statement T generated by their combination, can be computed depending on the location of S-lloc and S-rloc in the memory organization, as follows:

Case 1: S-lloc represents a stack location: If S-rloc is a stack location, the basic heap statement corresponding to T is `p = q`. If S-rloc is a heap location, the corresponding basic heap statement is `p = q->f`. The general rules to compute the gen sets for these two cases, are derived by simply parameterizing the gen sets stack_lhs_stack_rhs_gen_set and stack_lhs_heap_rhs_gen_set defined for the two basic heap statements, as shown in Figure 5.13.

Case 2: S-lloc represents a heap location:

Case 2(a): If S-rloc is a stack location, the basic heap statement corresponding to T is `p->f = q`. The rule to compute the gen set for this case is obtained by appropriately parameterizing the set heap_lhs_stack_rhs_gen_set as shown in Figure 5.13.

Case 2(b): If S-rloc is a heap location, the basic heap statement corresponding to T would be `p->f = q->f`. Our analysis breaks down this statement as the following sequence of basic heap statements: (`temp = q->f; p->f = temp`). First, the statement `temp = q->f` is analyzed with the input matrices. Using the resulting matrices,

```
/* Compute the gen set for statement S with input matrices *
 * D, I and A. H is the set of pointers abstracted by them */
fun build_gen_set(S,D,I,A,H)
  gen_set = {} /* Initialize gen set */
  if (is_null(rhs(S))) /* No new relationships are generated */
    return(gen_set)
  Let l = Root(lhs(S)) /* Root of Var Ref on lhs(S) */
  Let r = Root(rhs(S)) /* Root of Var Ref on rhs(S) */
  foreach (x, d) ∈ S−locations(lhs(S))
    if ((x, d) ≡ (heap, P)) /* S−location(lhs(S)) is a heap location */
      gen_set = gen_set ∪ build_heap_lhs_gen_set(S,D,I,A,H,l,r)
    else /* S−location(lhs(S)) is a stack location */
      gen_set = gen_set ∪ build_stack_lhs_gen_set(S,D,I,A,H,x,r)
  return(gen_set)
```

Figure 5.12: Computing Gen Set for a Basic SIMPLE Statement

```
/* S : Statement, D, I, A : Matrices, H : Set of pointers *
 * abstracted, (x,P) : S−location(lhs(S)) r : Root(rhs(S)) */
fun build_stack_lhs_gen_set(S,D,I,A,H,x,r)
  gen_set = {}
  if (is_malloc(rhs(S)) /* x = malloc() */
    gen_set = malloc_gen_set(D,I,H,x)
  else if (is_address_op(rhs(S))) and (r, heap, P) /* x = &(r−>f) */
    gen_set = stack_lhs_stack_rhs_gen_set(D,I,H,x,r)
  else if (is_arith_expr(rhs(S))) /* x = r op k */
    gen_set = stack_lhs_stack_rhs_gen_set(D,I,H,x,r)
  else
    foreach (y, d) ∈ S−locations(rhs(S))
      if ((y, d) ≡ (heap, P)) /* x = r−>f : r is heap-directed */
        gen_set = gen_set ∪ stack_lhs_heap_rhs_gen_set(D,I,A,H,x,r)
      else /* (y, d) is a stack location : x = y */
        gen_set = gen_set ∪ stack_lhs_stack_rhs_gen_set(D,I,H,x,y)
  return(gen_set)

/* S : Statement, D, I, A : Matrices, H : Set of pointers abstracted *
 * (heap,P) : S−location(lhs(S)), l : Root(lhs(S)), r : Root(rhs(S)) */
fun build_heap_lhs_gen_set(S,D,I,A,H,l,r)
  gen_set = {}
  if (is_address_op(rhs(S))) and (r, heap, P) /* l−>f = &(r−>f) */
    gen_set = heap_lhs_stack_rhs_gen_set(D,I,A,H,l,r)
  else if (is_arith_expr(rhs(S))) /* l−>f = r op k */
    gen_set = heap_lhs_stack_rhs_gen_set(D,I,A,H,l,r)
  else
    foreach (y, d) ∈ S−locations(rhs(S))
      if ((y, d) ≡ (heap, P)) /* l−>f = r−>f : l and r are heap-directed */
        gen_set = gen_set ∪ heap_lhs_heap_rhs_gen_set(D,I,A,H,l,r)
      else /* (y, d) is a stack location : l−>f = y */
        gen_set = gen_set ∪ heap_lhs_stack_rhs_gen_set(D,I,A,H,l,y)
  return(gen_set)
```

Figure 5.13: Computing Gen Sets using S-locations

the gen set for the statement `p->f = temp` is computed, which forms the gen set for the statement `p->f = q->f`. We summarize the gen set computation for this case as follows:

heap_lhs_heap_rhs_gen_set(D,I,A,H,x,y) =
  heap_lhs_stack_rhs_gen_set(D_m,I_m,A_m,H,x,temp)
  where [D_m,I_m,A_m] = process_basic_stmt(`temp = y->f`,D,I,A,H)

The function process_basic_stmt is defined in Figure 5.11.

Special cases: In this case, rhs(S) (S is the basic SIMPLE statement) can be: (i) `NULL`, (ii) a call to `malloc`, (iii) an address operation, or (iv) an arithmetic operation. If rhs(S) is `NULL`, the gen set is empty. If it is a call to `malloc`, and statement S is of the form `x = malloc()`, the gen set can be obtained by appropriately parameterizing the set malloc_gen_set defined for the basic heap statement `p = malloc()`. If statement S is of the form `x->f = malloc()`, we generate different S-statments for it depending on the points-to relationships of `x`. If `x` is heap-directed, this statement simply adds an anonymous node to the data structure pointed to by `x`, and does not kill or generate any relationships. Finally, if rhs(S) is an arithmetic or an address operation, S-rloc is simply considered as the stack location Root(rhs(S)), and the appropriate gen set rule is used depending on S-lloc, as shown in Figure 5.13.

5.3.3 Estimating New Attributes

The effect of the SIMPLE statement S, on the shape and root attributes of heap-directed pointers, is also estimated by considering all the S-statements it can generate. For each S-statement, we calculate: (i) the set of pointers H_r whose root attribute is affected, (ii) the set of pointers H_s whose shape attribute is affected, and (iii) the matrix A_c which stores the new attributes of pointers in sets H_r and H_s. The overall H_r and H_s sets for the SIMPLE statement S, are obtained by taking the union of the individual H_r and H_s sets of its S-statements. Similarly, the matrix A_c for statement S is obtained by merging the individual A_c matrices of its S-statements. The complete rules to estimate the modified attributes are presented in Figures 5.14, 5.15 and 5.16. It can be noticed from Figure 5.16, that the computation of sets H_r, H_s and the matrix A_c for any S-statement, again depends on the S-locations (S-lloc and S-rloc) it represents.

```
/* Estimate the attributes modified by statement S with input *
 * matrices as D, I and A. H is the set of pointers abstracted. *
 * H_r is the set of pointers whose root attribute is modified. *
 * H_s is the set of pointers whose shape attribute is modified. *
 * A_c contains the new attributes of pointers in H_r and H_s */
fun find_mod_attr(S,D,I,A,H)
  [H_r,H_s,A_c] = {} /* Initialization */
  Let l = Root(lhs(S)) /* Root of Var Ref on lhs(S) */
  Let r = Root(rhs(S)) /* Root of Var Ref on rhs(S) */
  foreach (x,d) ∈ S−locations(lhs(S))
    if ((x,d) ≡ (heap,P)) /* S−location(lhs(S)) is a heap location  */
     [pH_r,pH_s,pA_c] = find_heap_lhs_mod_attr(S,D,I,A,H,x,r)
    else /* S−location(lhs(S)) is a stack location */
     [pH_r,pH_s,pA_c] = find_stack_lhs_mod_attr(S,D,I,A,H,x,r)
    [H_r,H_s,A_c] = merge_attr(H_r,H_s,A_c,pH_r,pH_s,pA_c)
  return([H_r,H_s,A_c])

/* Merge the attribute information in matrices A_c and pA_c */
fun merge_attr(H_r,H_s,A_c,pH_r,pH_s,pA_c)
  foreach s ∈ pH_r
    if (s ∈ H_r)
      A_c[s.root] = A_c[s.root] ⋈ pA_c[s.root]
    else
      A_c[s.root] = pA_c[s.root]
  foreach s ∈ pH_s
    if (s ∈ H_s)
      A_c[s.shape] = A_c[s.shape] ⋈ pA_c[s.shape]
    else
      A_c[s.shape] = pA_c[s.shape]
  /* Obtain the new H_r and H_s sets */
  H_r = H_r ∪ pH_r
  H_s = H_s ∪ pH_s
  return([H_r,H_s,A_c])
```

Figure 5.14: Estimating Attributes Modified by a Basic SIMPLE Statement

```
/* S : Statement, D, I, A : Matrices, H : Set of pointers *
 * abstracted, (x,P) : S-location(lhs(S)) r : Root(rhs(S)) */
fun find_stack_lhs_mod_attr(S,D,I,A,H,x,r)
  [H_r,H_s,A_c] = {} /* Initialization */
  if (is_malloc(rhs(S)) or (is_null(rhs(S)) /* x = malloc() or x = NULL */
    [H_r,H_s,A_c] = malloc_mod_attr(A,x)
  else if (is_address_op(rhs(S))) and (r, heap, P) /* x = &(r->f) */
    [H_r,H_s,A_c] = stack_lhs_stack_rhs_mod_attr(A,x,r)
  else if (is_arith_expr(rhs(S))) /* x = r op k */
    [H_r,H_s,A_c] = stack_lhs_stack_rhs_mod_attr(A,x,r)
  else
    foreach (y, d) ∈ S-locations(rhs(S))
      if ((y, d) ≡ (heap, P)) /* x = r->f : r is heap-directed */
        [pH_r,pH_s,pA_c] = stack_lhs_heap_rhs_mod_attr(A,x,r)
      else /* (y, d) is a stack location : x = y */
        [pH_r,pH_s,pA_c] = stack_lhs_stack_rhs_mod_attr(A,x,y)
      [H_r,H_s,A_c] = merge_attr(H_r,H_s,A_c,pH_r,pH_s,pA_c)
  return([H_r,H_s,A_c])

 /* (heap,P) : S-location(lhs(S)), l : Root(lhs(S)), r : Root(rhs(S)) */
fun find_heap_lhs_mod_attr(S,D,I,A,H,l,r)
  [H_r,H_s,A_c] = {} /* Initialization */
  if (is_address_op(rhs(S))) and (r, heap, P) /* l->f = &(r->f) */
    [H_r,H_s,A_c] = heap_lhs_stack_rhs_mod_attr(D,I,A,H,l,r)
  else if (is_arith_expr(rhs(S))) /* l->f = r op k */
    [H_r,H_s,A_c] = heap_lhs_stack_rhs_mod_attr(D,I,A,H,l,r)
  else
    foreach (y, d) ∈ S-locations(rhs(S))
      if ((y, d) ≡ (heap, P)) /* l->f = r->f : l and r are heap-directed */
        [pH_r,pH_s,pA_c] = heap_lhs_heap_rhs_mod_attr(D,I,A,H,l,r)
      else /* (y, d) is a stack location : l->f = y */
        [pH_r,pH_s,pA_c] = heap_lhs_stack_rhs_mod_attr(D,I,A,H,l,y)
      [H_r,H_s,A_c] = merge_attr(H_r,H_s,A_c,pH_r,pH_s,pA_c)
  return([H_r,H_s,A_c])
```

Figure 5.15: Estimating Attributes using S-locations

```
fun malloc_mod_attr(A,x)
  H_r = { x }    H_s = { x }
  A_c[x.shape] = Tree    A_c[x.root] = True
  return ([H_r,H_s,A_c])

fun stack_lhs_stack_rhs_mod_attr(A,x,y) =
  H_r = { x }    H_s = { x }
  A_c[x.shape] = A[x.shape]    A_c[x.root] = A[y.root]
  return ([H_r,H_s,A_c])

fun stack_lhs_heap_rhs_mod_attr(A,x,y) =
  H_r = { x }    H_s = { x }
  A_c[x.shape] = A[x.shape]    A_c[x.root] = False
  return ([H_r,H_s,A_c])

fun heap_lhs_stack_rhs_mod_attr(D,I,A,H,x,y) =
  [H_r,H_s,A_c] = {}
  if (D[y,x])
    H_s = { s | s ∈ H ∧ (D[s,x] ∨ D[s,y]) }
    ∀ s ∈ H_s, A_c[s.shape] = Cycle
  else if (A[y.shape] == Tree)
    H_s = { s | s ∈ H ∧ (D[s,x] ∧ I[s,y]) }
    ∀ s ∈ H_s, A_c[s.shape] = A[s.shape] ⋈ Dag
  else /* A[y.shape] is Dag or Cycle */
    H_s = { s | s ∈ H ∧ D[s,x] }
    ∀ s ∈ H_s, A_c[s.shape] = A[s.shape] ⋈ A[y.shape]
  H_r = { s | s ∈ H ∧ D[y,s] }
  ∀ s ∈ H_r, A_c[s.root] = False
  return ([H_r,H_s,A_c])

fun heap_lhs_heap_rhs_mod_attr(D,I,A,H,x,y) =
  [D_m,I_m,H_m] = process_basic_stmt(temp = y->f,D,I,A,H)
  return (heap_lhs_stack_rhs_mod_attr(D_m,I_m,A_m,H,x,temp))
```

Figure 5.16: Calculating New Attributes

5.3.4 An Example

We now demonstrate shape analysis on a basic SIMPLE statement S. Let statement S be `r = s->f` with the following points-to relationships: $\{(r, heap, P), (s, heap, P), (s, d, P)\}$. The heap structure before the statement is shown in Figure 5.17(a). The set S-locations(lhs(S)) consists of the definite S-location (r, D), so all relationships of `r` get killed, resulting in the following kill set:

kill_set(S) = { D[r,u],D[u,r],I[u,r] }

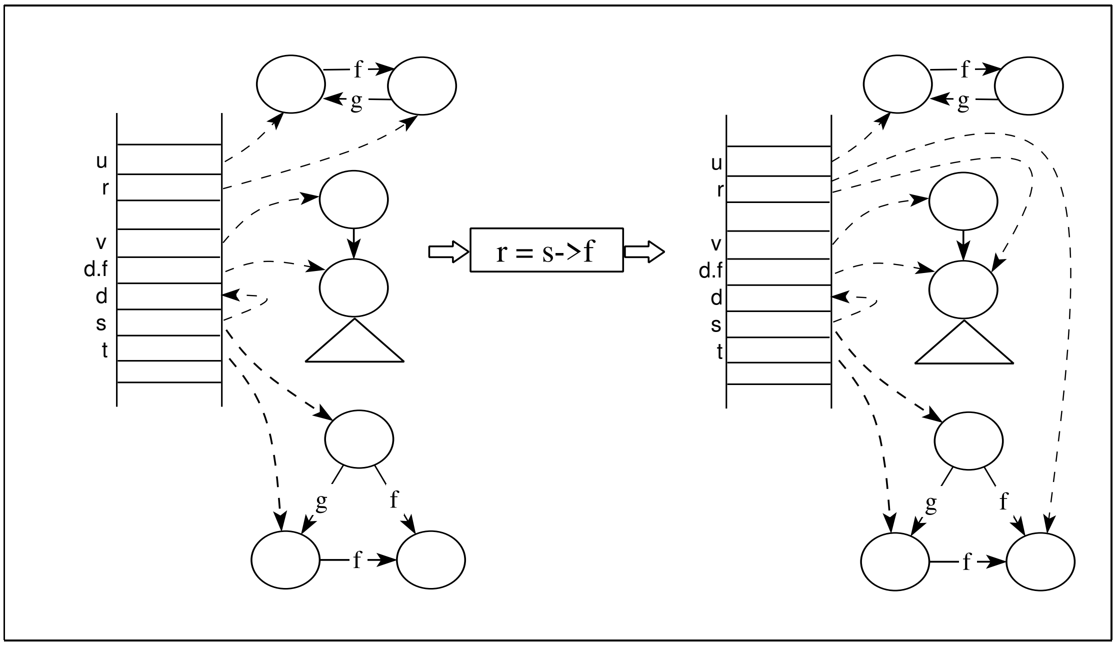


Figure 5.17: Analyzing Basic SIMPLE Statement `r = s->f`

With the above points-to relationships, the statement S generates the following two S-statements: (i) T1: `r = d.f` and (ii) T2: `r = s->f` where `s` points to a heap location. We get the following gen sets for the S-statements T1 and T2, using the stack_lhs_stack_rhs_gen_set and stack_lhs_heap_rhs_gen_set rules respectively:

gen_set(T1) = { D[r,d.f],D[d.f,r],D[v,r],I[r,d.f],I[r,v] }
gen_set(T2) = { D[s,r],D[r,t],D[t,r],I[r,s],I[r,t] }
gen_set(S) = gen_set(T1) ∪ gen_set(T2)

The statements T1 and T2 can change the attributes of only pointer `r`. So we have both H_r and H_s as { r }. For T1, we follow the stack_lhs_stack_rhs_mod_attr rule and have A_c[r.shape] = A[(d.f).shape] and A_c[r.root] = A[(d.f).root]. For T2, we follow the stack_lhs_heap_rhs_mod_attr rule and thus have A_c[r.shape] = A[s.shape] and A_c[r.root] = False. The matrix A_c for statement S is obtained by merging the matrices for its S-statements T1 and T2. So for the SIMPLE statement S, we have A_c[r.shape] = A[(d.f).shape] $\bowtie$ A[s.shape] $\Rightarrow$ Tree $\bowtie$ Dag $\Rightarrow$ Dag. Similarly, we have A_c[r.root] = A[(d.f).root] $\bowtie$ False $\Rightarrow$ False $\bowtie$ True $\Rightarrow$ False.

The current matrices D, I and A are copied over to the new matrices D_n, I_n and A_n. The relationships in kill_set(S) are then deleted from the matrices D_n and I_n. Next, the relationships in gen_set(S) are added to the two matrices. Since the set S-locations(lhs(S)) consists of the definite S-location (r, D), the attributes of `r` are set to default values: A_n[r.shape] = Tree, A_n[r.root] = True. Finally, the attributes of pointers belonging to the sets H_r and H_s, are merged with their attributes in the matrix A_c. Since `r` is the only pointer in sets H_r and H_s, we have A_n[r.shape] = A_n[r.shape] $\bowtie$ A_c[r.shape] $\Rightarrow$ Tree $\bowtie$ Dag $\Rightarrow$ Dag. Using the same strategy, we have A_n[r.root] = False. The attributes of other pointers remain same as in matrix A. The heap structure after the statement S is shown in Figure 5.17(b).

5.4 Analyzing Compositional Control Statements

The overall strategy for analyzing control statements is same as that described for connection analysis. However, now the fixed-point computation has to take into account three matrices as opposed to only one. The merge operator for direction and interference matrices is also simply the logical OR operation as for connection matrix. We have already defined the merge operator $\bowtie$ for the attribute matrix. Thus we have the following merge rules:

$$\begin{aligned}
&\text{Merge}(D_n,D) \Rightarrow \forall\ r,s \in H,\ D_n[r,s] = D_n[r,s] \vee D[r,s]\\
&\text{Merge}(I_n,I) \Rightarrow \forall\ r,s \in H,\ I_n[r,s] = I_n[r,s] \vee I[r,s]\\
&\text{Merge}(A_n,A) \Rightarrow \forall\ s \in H,\ A_n[s.shape] = A_n[s.shape] \bowtie A[s.shape],\\
&\qquad\qquad\qquad\qquad\qquad\quad A_n[s.root] = A_n[s.root] \bowtie A[s.root]
\end{aligned}$$

We demonstrate the analysis of control statements, by presenting the algorithm for analyzing the `while` statement in Figure 5.18.

```
/* D,I,A : Input matrices, H : Set of pointers abstracted,
 * ign : Current invocation graph node */
fun process_while(cond,body,D,I,A,H,ign) =
  do
    prevD = D; prevI = I; prevA = A;
    [D1,I1,A1] = process_basic_stmt(cond,D,I,A,H);
    [D2,I2,A2] = process_stmt(body,D1,I1,A1,H,ign);
    D = Merge(D,D2); I = Merge(I,I2); A = Merge(A,A2);
  while ((D != prevD) and (I != prevI) and (A != prevA));
  return([D,I,A]);
```

Figure 5.18: Analyzing a `while` Statement

5.5 Interprocedural Analysis

The overall interprocedural strategy for shape analysis is similar to that for connection analysis presented in section 4.3. However, now three matrices (D, I and A) need to be handled as opposed to one. All three of them are simultaneously mapped and unmapped. Further, all of them participate in the fixed-point approximation for recursive procedure calls, in the same manner as for the `while` statement analysis shown in Figure 5.18.

The only aspect where the interprocedural scheme for shape analysis basically differs from that for connection analysis, is in the mapping of *inaccessible* local pointers in the caller to symbolic names in the callee. Recall that inaccessible local pointers are those pointers in the caller, which are neither directly nor indirectly accessible in the callee, but which have relationships with pointers accessible in the callee. Since these relationships can be changed by the call, inaccessible pointers need to be mapped to special symbolic names, as explained in section 4.3.5.

Connection analysis maps inaccessible pointers to special '0+'-prefixed symbolic names in the callee. For example, if an inaccessible local pointer `r` is connected with a global pointer `s` at the call-site, then `r` is mapped to the symbolic name `0+s` and inside the callee we have `0+s` connected with `s`. Unlike connection relationships, direction relationships are not symmetric. Considering the above example in the context of direction relationships, we would like to map `r` to different symbolic names depending upon if `r` has a path *to* `s` or a path *from* `s`.

If all inaccessible pointers having direction relationships with `s` are mapped to the same symbolic name, unnecessary imprecision can be introduced. For example, consider two inaccessible pointers `u` and `v` having direction relationship with global pointer `s`. Assume that `u` has a path to `s` while `v` has a path from `s`. Now if both `u` and `v` are mapped to the symbolic name `0+s`, we will have `0+s` having paths both to and from `s`. Assuming that the direction relationships between `s` and `0+s` are not affected by the callee, on returning from the callee we will have both `u` and `v` having paths to as well as from `s`, thereby introducing two spurious direction relationships.

To avoid this imprecision, for each '0+'-prefixed symbolic name, we generate another '0-'-prefixed symbolic name. In the above example, we will have two symbolic names corresponding to `s`: `0+s` and `0-s`. Inaccessible pointers having paths *to* `s` (like `u`) will be mapped to `0-s`, while those having paths *from* `s` (like `v`) will be mapped to `0+s`.

The inaccessible pointers are first mapped based on their direction relationships. Now a pointer can also fall into the inaccessible category, because of having an interference relationship alone. So, if an inaccessible pointer has not already been mapped due to a direction relationship, it is mapped based on its interference relationships. It should be noted that more than one pointer can be mapped to a symbolic name, but not vice versa. When a symbolic name represents more than one pointer, its relationships and attributes become the merge of the relationships and attributes of these pointers.

The complete algorithm for mapping names in caller to appropriate names in callee is given in Figure 5.19. More complex algorithms can be devised to improve the accuracy of mapping. However, our experimental results indicate that this simple scheme suffices for real C benchmark programs. The algorithms for mapping and unmapping matrices, and for handling recursive, approximate and indirect procedure calls are similar to that for connection analysis, except that here three matrices are simultaneously handled. So we do not describe them again.

5.6 Summary

In this chapter we presented a new heap data structure analysis called shape analysis. We demonstrated how it uses relatively simple abstractions to identify of tree and dag-like data structures built by the program. We also introduced the idea of associating root and shape attributes with each heap-directed pointer. This enables abstraction of the properties of the subpiece of the data structure *accessible* from the given pointer, as opposed to those of the entire data structure it points to.

```
/* Functions to map names in matrices D_m and I_m at call-site, to names in
 * matrices D_e and I_e at procedure entry for the call corresponding to the
 * invocation graph node ign */
fun di_map_names(D_m,I_m,H_m,actualList,formalList,ign)
  foreach r ∈ H_m
    /* Find the name r should be mapped to in the called procedure */
    x = find_mapped_name(r,D_m,I_m,H_m,actualList,formalList,ign);
    /* r is mapped to the name denoted by x for this invocation */
    di_mapped_name(r,ign) = x;
  return;

fun find_mapped_name(r,D_m,I_m,H_m,actualList,formalList,ign)
  if is_defined(di_mapped_name(r,ign)) /* already mapped */
    return;
  if is_global(r) /* global pointers are mapped to themselves */
    return r;
  if is_defined(ign.ptMapInfo(r)) /* already mapped by points-to analysis */
    return (ign.ptMapInfo(r));
  /* r is an inaccessible local. Determine the set of globals, indirectly
   * accessible pointers, and actual arguments, r has relationships with */
  di_set = {};
  foreach s ∈ H_m
    if ((is_global(s)) or (is_defined(ign.ptMapInfo(s)) or
          (s ∈ actualList))
      if (D_m[r,s]) /* r has a path to s */
        /* Find the '0-'-prefixed symbolic name corresponding to s */
        s_sym = di_symbolic_var_1(s);
        di_set = di_set ∪ {s_sym};
      else if (D_m[s,r] or I_m[r,s]) /* r has a path from or interferes with s */
        /* Find the '0+'-prefixed symbolic name corresponding to s */
        s_sym = di_symbolic_var_2(s);
        di_set = di_set ∪ {s_sym};
  if (is_empty(di_set))
    return undefined; /* r need not be mapped */
  /* return the variable in di_set with minimum number of vars mapped to it */
  return(min_mapped_var(di_set));
```

Figure 5.19: Mapping Names From Caller to Callee

Since we only abstract boolean relationships like *path existence* and *interference* (for efficiency reasons), we are not able to precisely handle destructive updates where a tree data structure temporarily becomes dag-like or cyclic, and then again becomes a tree. Analysis techniques that capture more precise path relationships between heap-directed pointers as path expressions [HN90], symbolic access paths [Deu94] or storage shape graphs [CWZ90], can handle such cases to some extent, but they also incur considerable cost. Our future research will focus on designing practical abstractions to handle destructive updates more accurately. We briefly present one such abstraction in chapter 7.

Chapter 6

Experimental Results

In this chapter, we present empirical data to demonstrate the effectiveness of connection and shape analyses presented in chapters 4 and 5. To perform this study, we implemented these analyses in the framework of our McCAT C compiler, and analyzed a collection of C programs written for both scientific and non-scientific applications. We present the results for connection analysis in section 6.1, and for shape analysis in section 6.2.

6.1 Connection Analysis Results

In this section, we present the experimental results obtained from connection analysis of a set of 13 C programs. We chose programs that use a significant amount of dynamic allocation, as benchmarks for our study. Below we give a brief description of each benchmark program, and the principal data structures it uses:

- **genetic:** It implements a genetic algorithm to test sorting. The principal data structures used by this program are three global dynamically allocated arrays of type `int`, which are also passed as parameters to various functions. Henceforth, we will refer to dynamically allocated arrays as simply dynamic arrays.
- **sim:** This is a benchmark from computational biology that computes k-best non-intersecting alignments within a single DNA sequence or between two DNA sequences, using dynamic programming. The main data structures used by this program are dynamic arrays of type `long`. It also uses dynamic arrays of pointers to structures. It allocates two types of structures: one with no pointer fields and one with a recursive pointer field.

- **blocks2:** This is another benchmark from computational biology that computes multiple aligned blocks from a given family of pairwise alignments for DNA sequences. It mainly uses dynamic arrays of type `long`. It also builds a constraint graph data structure using dynamic arrays of pointers to recursive structures.

- **ear:** This is a SPECINT92 benchmark that implements a model of acoustic propagation and detection in the human cochlea. It uses dynamic arrays and structures with non-recursive pointer fields.

- **assembler:** It implements an assembler and its principal data structures include: dynamic arrays and a linked list implementation for the symbol table.

- **loader:** It implements a loader, and uses the same data structures as the benchmark *assembler*. Both of these benchmarks are part of William Landi's test suite [LR92], and have been obtained from him.

- **cholesky:** It performs Cholesky factorization of a sparse positive definite matrix. It is part of the SPLASH [SWG91] benchmark suite from Stanford. It implements the sparse matrix using structures with non-recursive pointer fields. These pointers point to dynamic arrays of type `int`.

- **mp3d:** This is another benchmark from the SPLASH suite related to rarefied fluid flow simulation used in aerospace research. It dynamically allocates structures with no pointer fields or with one non-recursive pointer field, and arrays of type `int` and `float`.

- **water:** It solves the molecular dynamics N-body problem to evaluate forces and potentials in a system of water molecules in the liquid state, using spatial data structures. It is part of the new SPLASH benchmark suite called SPLASH-2, and we use the sequential version. The primary data structures used by this program are linked lists and dynamically allocated arrays of pointers pointing to linked lists.

- **volrend:** This benchmark renders a three-dimensional volume onto a two-dimensional plane using an optimized ray casting technique. It is also a part of the SPLASH-2 benchmark suite, and we analyze its sequential version. It dynamically allocates a number of bit vectors to store, manipulate and render the image. Its principal data structure is an array of pointers on the stack, which point to bit vectors allocated in the heap.

- **chomp:** It implements a game tree and uses two recursive data structures: a binary tree and a linked list, besides dynamic arrays.

- **sparse:** It builds a large and random sparse matrix using two-dimensional linked lists, then scales, factors and solves it. The sparse matrix data structure is a cyclic structure with nodes having links to nodes in the previous and next rows as well as columns.

- **pug:** This program triangulates an unstructured grid using control volume finite element method. It uses a single complex cyclic data structure with nodes linked to one another through various pointers.

In Table 6.1, we give further information about the the benchmark programs. The following characteristics are presented for each program in the given order:

- Source lines including comments, counted using the *wc* utility.

- Number of statements in the SIMPLE intermediate representation. This number gives a good estimate of program size from the analysis point of view.

- Minimum, maximum and average number of variables abstracted by the connection matrices of various functions in the program (this includes symbolic variables introduced by our analysis). These numbers indicate the size of the abstraction and the memory requirements of the analysis for a given program.

- Total number of indirect references in the program, and the number of indirect references where the dereferenced pointer can point to a stack location, to a heap location and to both a stack and a heap location. We are able to determine the possible targets of indirect references, as we collect the above statistics after performing points-to analysis [Ema93, EGH94].

  The number of indirect references in a program, provides a measure for the relevance of pointer analysis to its optimization. The number of indirect references refering to stack and heap locations, respectively represent the significance of stack-based points-to analysis and heap-based data structure analyses for the given program.

The number of SIMPLE statements for the given benchmark set varies from 476 for *chomp* to 4909 for *volrend*, with an overall average of 2028 statements per program. The maximum number of variables abstracted by the connection matrix of a function is 133 for *pug*, followed by 114 for *cholesky*. The maximum of the average number of variables abstracted, is 89 for *cholesky* followed by 43 for *sim*. We will estimate the space requirements of the analysis using this data in appendix A on implementation details.

Program	Source Lines	SIMPLE stmts	Min vars	Max vars	Avg vars	Ind Refs	To Stack	To Heap	Stack/ Heap
genetic	506	479	6	14	7	54	28	30	4
sim	1422	1760	38	69	43	374	34	340	0
blocks2	876	1070	28	54	33	373	98	275	0
ear	4953	3476	38	51	39	290	143	147	0
assembler	3361	3071	12	26	14	718	666	52	0
loader	1539	1055	7	20	10	170	106	64	0
cholesky	1899	2217	76	114	89	488	22	466	0
mp3d	1687	1849	18	28	20	490	25	465	0
water	2703	2418	8	65	27	581	32	549	0
volrend	4207	4909	18	45	20	190	63	128	1
chomp	430	476	20	27	22	127	45	82	0
sparse	2859	1495	12	40	18	468	3	465	0
pug	2400	2089	16	133	30	822	147	688	13

Table 6.1: Characteristics of Connection Analysis Benchmarks

All the benchmarks have substantial number of indirect references, with maximum 822 for *pug* followed by 718 for *assembler*. Further, all of them have indirect references refering to both stack and heap locations, with majority of the indirect references refering to heap locations (except for the two benchmarks: *assembler* and *loader*). This makes the given benchmark set well-suited for evaluating a heap analysis.

While discussing the connection analysis of SIMPLE statements in Chapter 4, we had noticed that the analysis tends to become conservative, when a pointer can point to both heap and stack locations. However, the data in the last column of Table 6.1 shows that this does not happen very frequently in real C programs: pointers used to point to dynamically allocated memory, are not commonly used to also point to stack locations. We inspected the analysis output for programs *genetic*, *volrend* and *pug*, to detect the indirect references where it happens. We found that these indirect references mostly dereference formal parameters (of pointer type), to which both heap-directed and stack-directed pointers are passed as actuals, from different call-sites of the given function.

6.1.1 Measurements for Heap Related Indirect References

In Table 6.2, we present empirical measurements for connection analysis of the above benchmarks. Our measurements focus on indirect references in the program that refer to heap locations, as connection matrix information is computed to effectively resolve them at compile time. We motivate our measurements using the following example program:

```
main()
  { ...
    p = my_malloc(N);
    q = my_malloc(M);
    ...
S:  *p = INIT_VAL;
T:  *q = INIT_VAL;
    ...
  }
```

This program allocates two disjoint heap structures and then initializes them. Before connection analysis, the only information available from points-to analysis is: both the indirect references `*p` and `*q` (at statements S and T respectively) refer to the location *heap*, and thus the statements S and T interfere. After connection analysis, we know that the data structures pointed to by `p` and `q` are never connected (are disjoint), and hence the statements S and T do not interfere.

Our experimental measurements attempt to quantify the improvement in resolution of heap data structures provided by connection matrix information over that obtained from the conservative approximation of points-to analysis. With only points-to analysis one must assume that each heap-directed pointer is possibly connected with all other other heap-directed pointers, while with connection analysis one can identify a more precise set.

Thus, the effectiveness of connection analysis can be evaluated by comparing the total number of heap-directed pointers at an indirect reference (the conservative estimate provided by points-to analysis), with the total number of pointers connected with the dereferenced pointer (the more precise estimate available from connection analysis). For example, in the above program, at statement S, the total number of heap-directed pointers is two (both `p` and `q` are heap-directed), while the number of pointers connected with the dereferenced pointer `p` is only one (`p` itself). The same situation holds at statment T.

Following this strategy, we have calculated the following metrics for each benchmark program (presented in Table 6.2):

- **Refs:** Total number of indirect references in the program that can refer to heap locations.

- **cavg:** Average number of pointers that are connected with the dereferenced pointer at an indirect reference. This average is calculated as follows. At each indirect reference we determine the total number of pointers connected with the dereferenced pointer. Let us call this number as `cn_tot_i` for the ith indirect reference in the program (as per lexical order). We do not include symbolic variables in this count as we generate them only to facilitate interprocedural mapping, and they cannot be accessed or dereferenced by the program. Further if the dereferenced pointer is only connected with itself, the count `cn_tot_i` will be one for the given indirect reference.

  We then sum up the numbers `cn_tot_i` for all indirect references, and divide this sum total denoted as `cn_sum_tot` by the total number of heap related indirect references in the program (**Refs**), to obtain the average **cavg**. We cannot have `cavg` less than 1.0 (unless there are no heap related indirect references in the program and we have **Refs** as zero), as each heap-directed pointer is at least connected with itself.

- **havg:** Average number of pointers that are heap-directed at an indirect reference. This average is calculated in the same fashion as **cavg**. First, at each indirect reference the total number of heap-directed pointers is calculated as `heap_tot_i`. Next, this number is summed up for all indirect references, and the sum total `heap_sum_tot` is divided by **Refs** to obtain the average **havg**. Again symbolic variables are not considered in computing this average.

- **Impr:** A measure to approximate the percentange improvement provided by connection matrix information over points-to information, in effectively resolving heap related indirect references in the program. It is calculated using the following formula: ((`heap_sum_tot` - `cn_sum_tot`) * 100.0)/(`heap_sum_tot`), where as described above, `heap_sum_tot` gives the sum total of heap-directed pointers, and `cn_sum_tot` gives the sum total of connection relationships at indirect references in the program.

  Without connection analysis, the conservative approximation for the number `cn_sum_tot` would be simply `heap_sum_tot`, resulting in zero percentage improvement. With connection analysis, the more precise is the analysis, the fewer will be the number of connection relationships reported. This results in

Program	*a / (*a).b				a[i]			
	Refs	cavg	havg	% Impr	Refs	cavg	havg	% Impr
genetic	0	0.0	0.0	0.00	30	1.7	5.2	67.74
sim	96	3.4	23.2	85.55	244	1.6	20.4	92.41
blocks2	119	8.8	22.9	61.36	156	5.2	22.3	83.74
ear	42	2.7	3.8	27.22	105	2.4	7.1	66.26
assembler	45	4.4	7.8	42.98	7	6.0	9.4	36.36
loader	55	5.1	6.5	21.07	9	1.0	4.1	75.66
cholesky	82	14.9	34.3	56.46	384	3.7	20.7	82.27
mp3d	391	2.5	8.6	70.42	74	1.9	7.1	73.86
water	250	15.4	31.2	50.69	299	14.7	24.1	38.94
volrend	96	7.4	22.2	66.73	32	9.8	18.8	47.59
chomp	56	5.2	7.2	27.65	26	1.6	3.9	59.00
sparse	384	9.3	10.1	7.23	0	0.0	0.0	0.00
pug	514	36.8	36.9	0.30	174	47.6	47.7	0.11

Table 6.2: Empirical Measurements for Connection Analysis Results

a small `cn_sum_tot` and hence a greater percentage improvement. Thus, the metric **Impr** provides a reasonable measure for the effectiveness of connection analysis. For our small example program (given above): **Refs** is 2, `cn_sum_tot` is 2 and hence **cavg** is 1.0; `heap_sum_tot` is 4, `havg` is 2.0 and **Impr** is (((4 - 2) * 100.0) / 4) or 50%.

In Table 6.2, we present these measurements separately for indirect references of the type `*a/(*a).b`, and of the type `a[i]` where `a` is of pointer type. We discuss the results presented in this table below:

Indirect References of type `a[i]`: The percentage improvement (**Impr**) is in general higher for indirect references of this type. This happens because most of these references represent stack-based pointers that point to dynamically allocated memory and access it as an array (of non pointer type). For example, the statement `a = (int *) malloc(8 * sizeof(int))` dynamically allocates an array of eight integers. Now such array structures are in general not pointed to by many other pointers. In SIMPLE, the above statement is *simplified* as `temp_0 = malloc(8 * sizeof(int)); a = (int *) temp_0`, resulting in both `a` and `temp_0` pointing to the allocated structure.

In case the allocation is done through a user-defined routine (for example `a = my_malloc(size)`), the temporary variable is not generated, and pointer `a` alone points to the allocated structure. So the number of connection relationships of pointers like `a` tends to be close to 2.0 on an average. In Table 6.2, **cavg** for indirect array references is in the range of 1.0 to 3.7 for most of the benchmarks. For some benchmarks **cavg** tends to be much larger. We analyze them below.

The benchmarks *volrend* and *blocks2* use arrays of pointers. Since we represent the entire array by the array name, connection relationships of pointers representing different indices of the array get merged. This results in large number of relationships for the single name representing them in the connection matrix.

The benchmarks *assembler*, *water* and *pug* have pointers to arrays as fields of dynamically allocated structures (as opposed to being located on the stack). These pointers are reported to be connected with all other pointers that point to the given data structure. This results in larger overall **cavg** for these benchmarks. Actually **cavg** for *pug* is almost same as **havg**, as it builds only a single complex data structure, providing effectively no improvement.

Indirect References of type `*a/(*a).b`: For indirect references of this form, the percentage improvement is in general not as high as for indirect array references. Such indirect references commonly access big aggregate data structures that consist of a large number of heap objects, specially if the data structure is recursive. Several pointers point to any such data structure, and all of them have connection relationships with each other.

In our benchmark set, *sim* and *mp3d* primarily use structures with no pointer fields. The percentage improvement for them is quite high, as these structures are also stand-alone entities in the heap, like dynamic arrays of non pointer type.

The benchmarks *ear* and *cholesky* primarily allocate structures with non-recursive pointer fields. For *ear*, **cavg** is quite small, though the percentage improvement is not very high as not many pointers are heap-directed in this program. For *cholesky* we have more than 50 percent improvement.

The benchmark *volrend* allocates integers and floats in the heap and accesses them through indirect references of the form `*a`. The percentage improvement for it could be even higher, but it uses arrays of pointers to point to the heap-allocated integers and floats. The benchmark *blocks2* allocates several disjoint arrays of pointers to dynamically allocated objects of type `int` and user-defined structure types with both recursive and non-recursive pointer fields. So it has higher **cavg**, but shows substantial percentage improvement.

Program	Refs	cavg	havg	% Impr
genetic	30	1.7	5.2	67.74
sim	340	2.1	21.2	90.29
blocks2	275	6.8	22.5	69.86
ear	147	2.5	6.1	59.40
assembler	52	4.6	8.0	41.93
loader	64	4.5	6.1	26.21
cholesky	466	5.7	23.1	75.53
mp3d	465	2.4	8.4	70.88
water	549	15.0	27.3	45.05
volrend	128	8.0	21.3	62.52
chomp	82	4.1	6.2	33.86
sparse	384	9.3	10.1	7.23
pug	688	39.5	39.6	0.24

Table 6.3: Overall Connection Analysis Results

The benchmarks *assembler* and *loader* use two disjoint linked list data structures, *chomp* uses a linked list and a tree structure, while *water* uses arrays of linked lists several of which are disjoint at different points in the program. The percentage improvement statistics for these benchmarks show the following expected trend: the greater is the number of disjoint data structures used by a program, the better are the connection matrix results for it.

Finally, the programs *sparse* and *pug* use a single complex recursive data structure, and all heap-directed pointers point to it. Consequently, connection analysis provides negligible improvement for them.

In Table 6.3, we present the overall measurements for all the benchmark programs. The percentage improvement is highest for programs that primarily use dynamic arrays (of non pointer type) and structures without pointer fields (*sim*, *cholesky* and *mp3d*). For some programs (*genetic* and *ear*) the percentage improvement is not very high, but **cavg** is quite small which indicates that connection analysis provides effective information for them. Overall, the results show that if the given program uses disjoint data structures, connection analysis can always provide more accurate information for resolving heap related indirect references (as compared to the information provided by points-to analysis). Thus, the connection matrix abstraction works well for its target domain of applications.

To give a clearer picture of the connection matrix results, we present scatter plots of connection relationships for ten of our thirteen benchmarks, in Figures 6.1 to 6.5. Each `+` mark in a scatter plot represents an indirect reference in the given program. Its `x`-coordinate (horizontal axis) represents the total number of heap-directed pointers at the program point where the indirect reference is made. Its `y`-coordinate (vertical axis) represents the number of pointers connected with the pointer being dereferenced by the indirect reference.

The dotted line represents the (`x = y`) plot. A `+` mark falling on this line represents an indirect reference at which the number of total heap-directed pointers is equal to the number of pointers connected with the dereferenced pointer. A `+` mark on the bottom right hand corner of a plot represents an indirect reference where a large number of pointers are heap-directed but which has a few connection relationships. We cannot have any `+` mark above the dotted line, as for no indirect reference can the number of connected pointers be greater than the number of total heap-directed pointers.

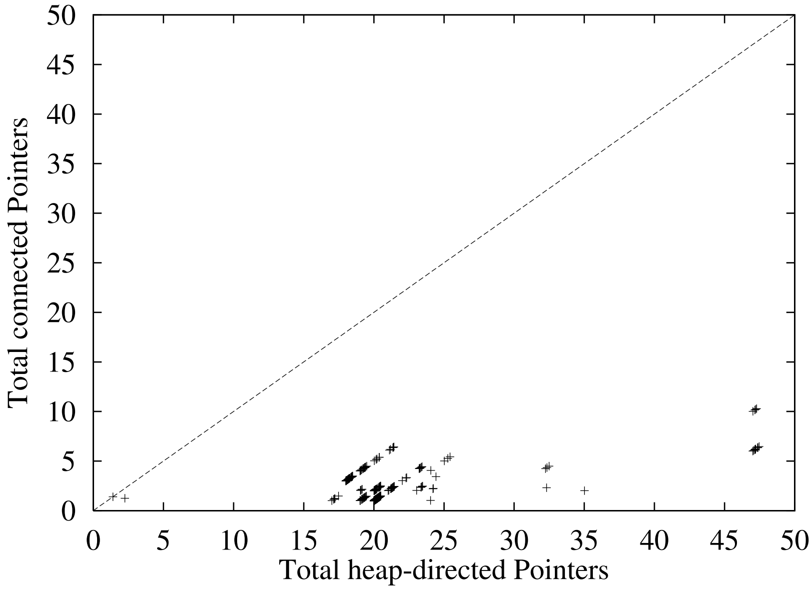
The effectiveness of connection analysis for any program can be evaluated by identifying the regions in its scatter plot, where the majority of its indirect references are represented. For example, for *sim*, *mp3d* and *cholesky*, most of the indirect references fall close to the horizontal axis, indicating that they have very few connection relationships. For the program *sparse*, majority of the indirect references fall on the dotted line, indicating that connection analysis provides negligible improvement for it. The scatter plots for other benchmarks can be interpreted accordingly.

While collecting the data, we noticed that scatter plot coordinates for many indirect references in a given program, turn out to be identical. Thus, we end up having one `+` mark representing several indirect references. To avoid this situation, we add a randomly generated fraction in the range [0.00,0.49] to both `x` and `y` coordinates of each point to be plotted. This strategy helps provide proper density effects in the plots, with negligible modification of the original data.

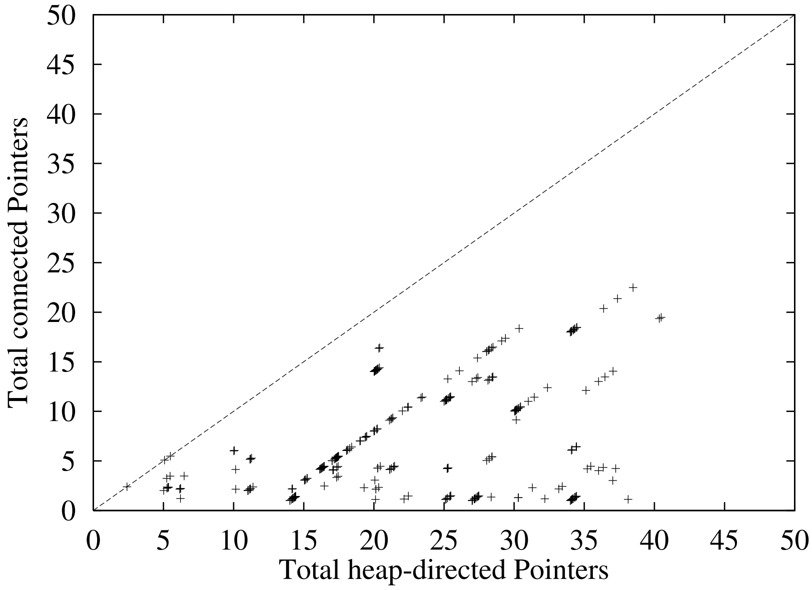
6.1.2 Interprocedural Measurements

Connection analysis is a context-sensitive interprocedural analysis. In Tables 6.4 and 6.5 we present some measurements demonstrating the interprocedural characteristics of the analysis, using the same set of benchmarks as listed in Table 6.1.

In Table 6.4, we provide some *static* interprocedural characteristics of the benchmarks. The first three columns in this table, respectively give the total number of

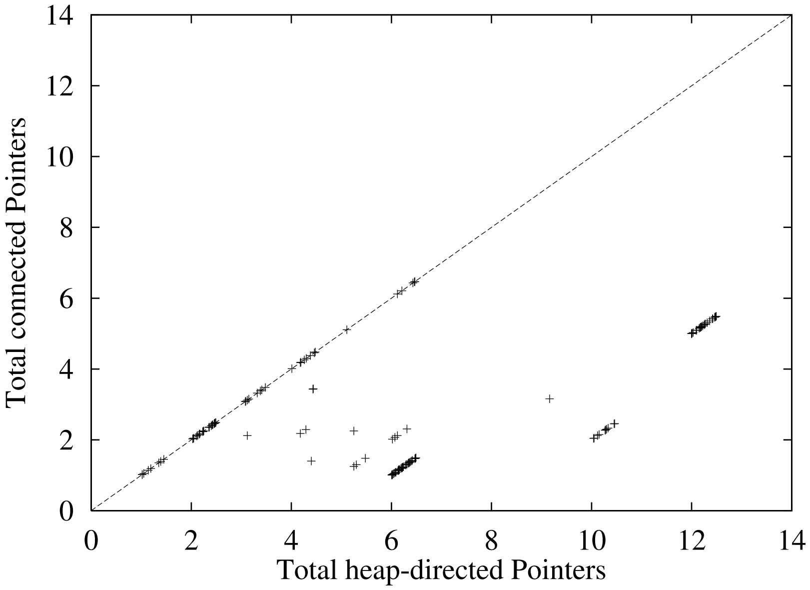


(a) *sim*

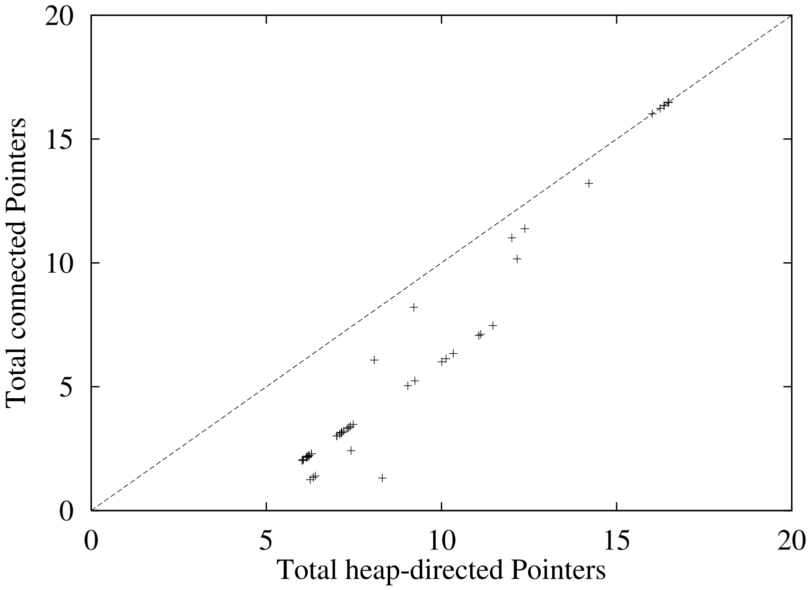


(b) *blocks2*

Figure 6.1: Connection Relationships at Indirect References

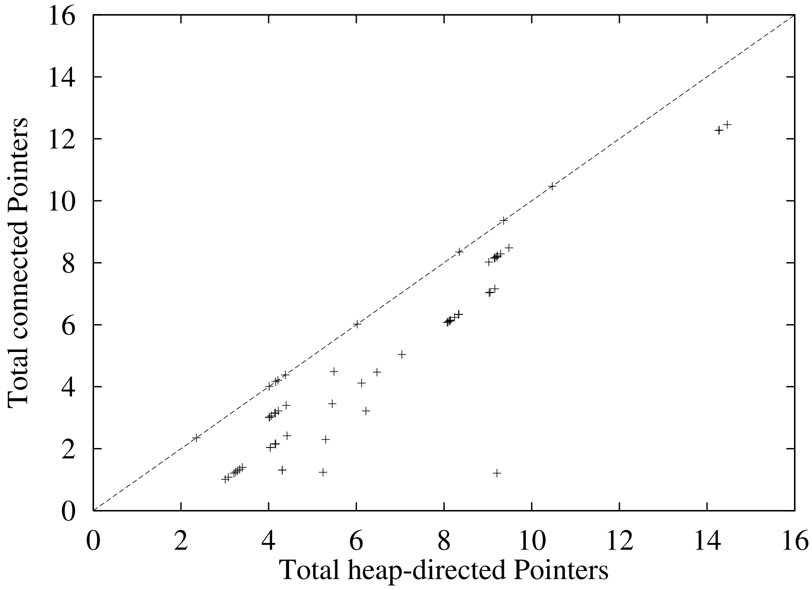


(a) *ear*

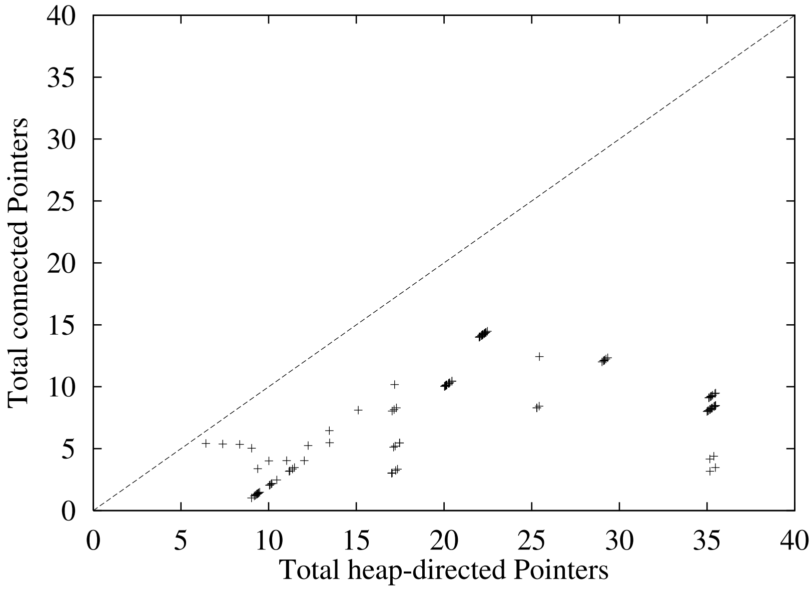


(b) *assem*

Figure 6.2: Connection Relationships at Indirect References

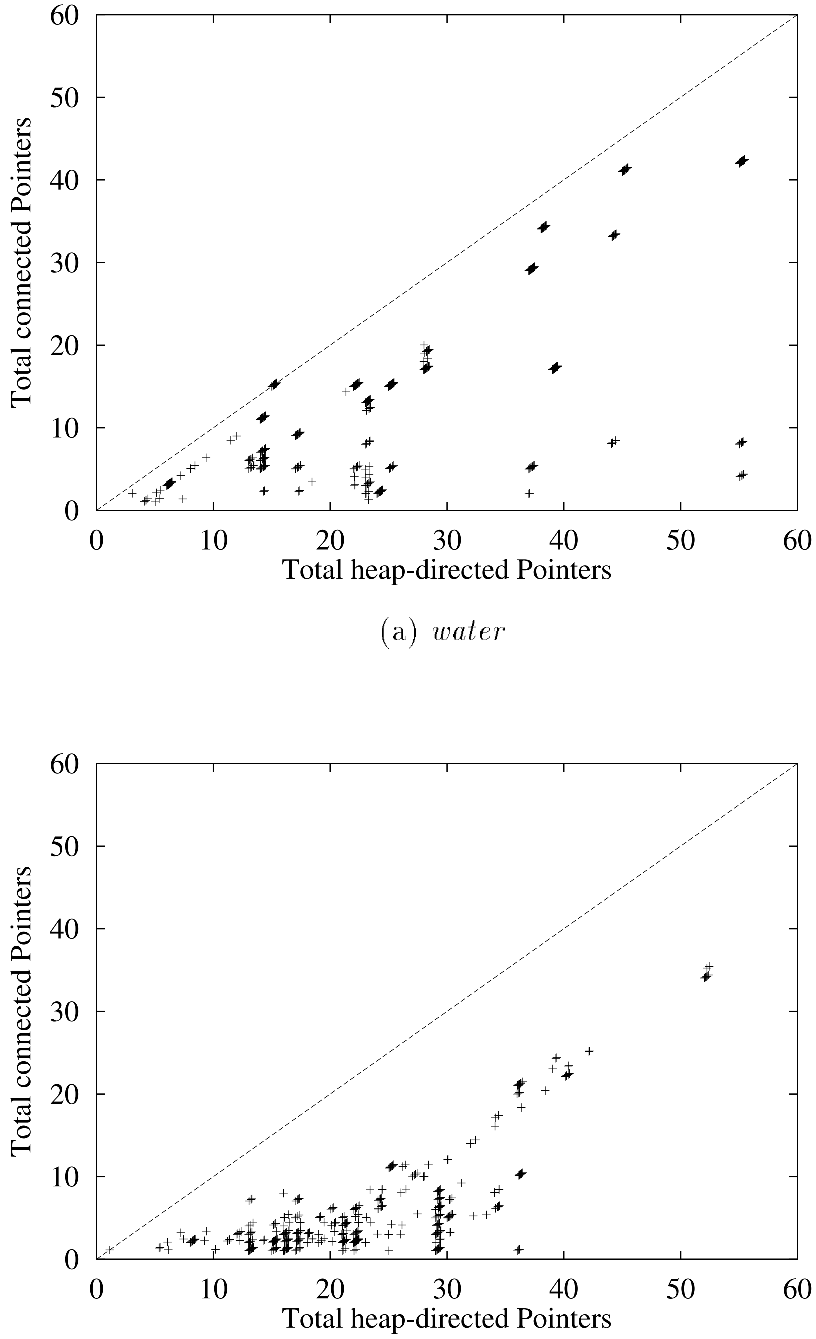


(a) *loader*



(b) *volrend*

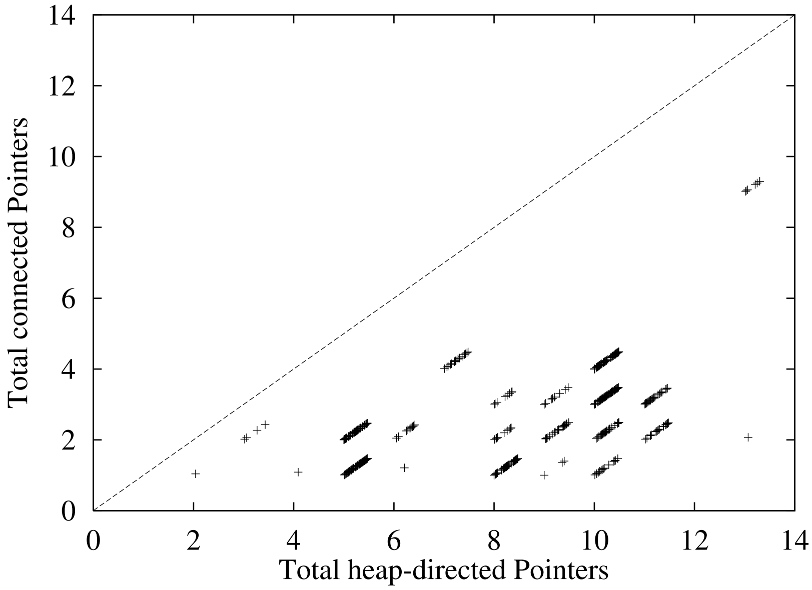
Figure 6.3: Connection Relationships at Indirect References



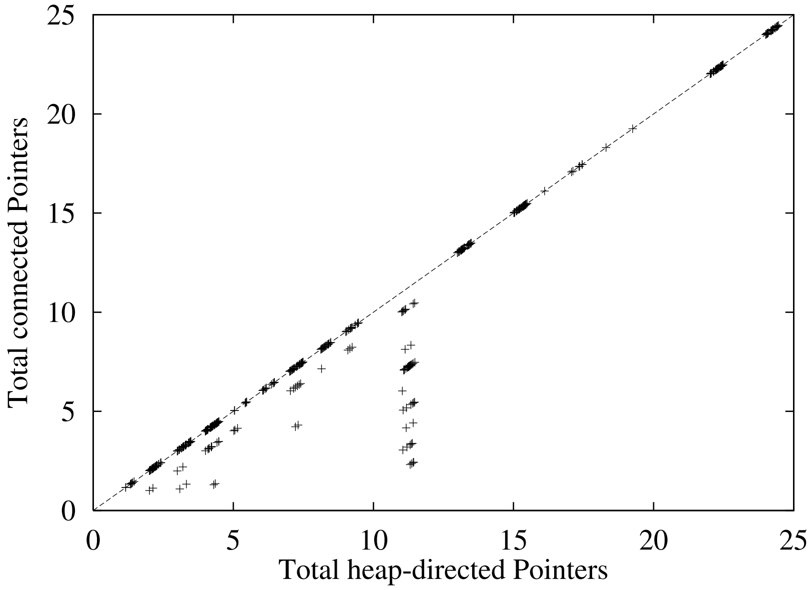
(a) *water*

(b) *cholesky*

Figure 6.4: Connection Relationships at Indirect References



(a) *mp3d*



(b) *sparse*

Figure 6.5: Connection Relationships at Indirect References

Program	fns	call sites	ig nodes	Recur nodes	Appr nodes	nodes/ call
genetic	17	32	45	0	0	1.41
sim	14	26	44	2	8	1.70
blocks2	20	28	28	1	2	1.00
ear	64	144	235	2	2	1.63
assembler	52	263	642	0	0	2.44
loader	30	82	125	2	2	1.52
cholesky	47	72	93	2	2	1.29
mp3d	23	28	32	0	0	1.14
water	15	21	26	0	0	1.24
volrend	53	108	169	2	2	1.56
chomp	22	47	98	7	7	2.09
sparse	28	76	121	0	0	1.59
pug	41	69	101	0	0	1.46

Table 6.4: Invocation Graph Charateristics of Connection Analysis Benchmarks

functions actually called in the program, the total number of call-sites in the program, and the total number of nodes in its invocation graph. The function `main` is not counted as a function or a call-site, but the invocation graph node representing `main` is counted. The last three columns give the number of recursive and approximate nodes, and the number of nodes per call-site, in the invocation graph of the given program.

In Table 6.5, we provide some *dynamic* interprocedural measurements for the benchmarks. More specifically, we give some statistics about the number of procedure calls analyzed during the analysis. The column labeled Tot gives the total number of procedure calls analyzed during the analysis. Our interprocedural algorithm analyzes a procedure once for each invocation context. Hence one might expect that the number of procedure calls analyzed would be equal to the number of nodes in the invocation graph. However, this is not true, since a procedure call can be analyzed several times for a single invocation context, if the call is involved in a loop or recursion fixed-point approximation.

The column labeled Memo in Table 6.5 shows the number of procedure calls that get memoized. A call is considered memoized if the input connection matrix (at the entry of the callee) for this call, is found equivalent to the stored input matrix at the invocation graph node corresponding to the call. Recall, that our interprocedural

Program	Calls Analyzed			Avgf	Avgc	Avgi
	Tot	Memo	Actual			
genetic	55	8	47	2.76	1.46	1.04
sim	71	10	61	4.36	2.35	1.39
blocks2	371	221	150	7.50	5.36	5.36
ear	268	30	238	3.72	1.86	1.01
assembler	767	101	666	12.80	2.53	1.04
loader	312	132	180	6.00	2.20	1.44
cholesky	132	35	97	2.06	1.35	1.04
mp3d	47	0	47	2.04	1.68	1.47
water	98	73	25	1.67	1.19	0.96
volrend	192	21	171	11.40	1.58	1.01
chomp	219	92	127	5.77	2.70	1.30
sparse	168	47	121	4.32	1.60	1.00
pug	160	47	113	2.75	1.64	1.12

Table 6.5: Interprocedural Measurements for Connection Analysis

analysis algorithm (shown in Figures 4.9 and 4.10), stores at each invocation graph node, the pair of input/output connection matrices valid respectively at the entry and exit of the callee, during the last visit to the node. So, in this case we can simply obtain the output matrix, from the invocation graph node, without re-analyzing the called procedure. So the number of calls actually analyzed is obtained by subtracting the number of memoized calls from the total number of calls analyzed (Tot - Memo). This number is given by the column labeled Actual in Table 6.5.

The last three columns in Table 6.5, labeled Avgf, Avgc and Avgi respectively give the average number of calls actually analyzed (given in the column labeled Actual) per function, per call-site and per invocation graph node. These averages are calculated by dividing the number in the Actual column, with the appropriate number from the first three columns in Table 6.4. In other words, Avgf, Avgc and Avgi respectively give the average number of times: (i) a function gets analyzed, (ii) a call-site is encountered during the analysis, and (iii) a call-chain in the program (possibly ending in recursion) is traversed during the analysis.

We make the following observations from the results reported in Tables 6.4 and 6.5:

- The maximum number of invocation graph nodes for our benchmark set is 642 for *assembler*, which also shows the highest nodes/call-site ratio of 2.44. This ratio is close to 1.5 for majority of the benchmarks, indicating that each call-site in general, appears on at most two call chains. These figures demonstrate the feasibility of our invocation graph based context-sensitive interprocedural analysis for this benchmark set.

However, this cannot be said in general. We have encountered some programs for which both the number of invocation graph nodes and the nodes/call-site ratio turn out to be pretty large, rendering our interprocedural strategy relatively expensive. These programs generally have a large number of call-sites for a few functions, which in turn have relatively big invocation (sub)graphs. These call-sites typically occur inside multiple `case` statements of big `switch` statements. The SPEC92 integer benchmark *sc*, which is a spreadsheet calculator, is one such example.

- A large number of procedure calls get memoized (Table 6.5). For *blocks2* as large as 221 of 371 calls and for *water* 73 of 98 calls get memoized. These are very encouraging results, specially considering the fact that presently we memoize a procedure call, only if it is analyzed more than once along the same call-chain (i.e. for the same invocation graph node). A higher degree of memoization can be achieved by trying to memoize all calls to a procedure (except the first one) irrespective of the call-chain they appear on. To this end, we need to compare the current input matrix with the stored matrix at all invocation graph nodes representing calls to the given procedure.

- Avgc for most of the benchmarks is close to 2.0 while Avgi is close to 1.0. This indicates that for our benchmark set, on an average a call-site is encountered twice, while a call-chain is traversed only once during the analysis. This is consistent with the observation that for this set, a call-site on an average appears on at most two call-chains. For *water* Avgi is less than 1.0, because in our statistics `main` is not counted as a call being analyzed, but it is counted as an invocation graph node.

  Avgf varies from benchmark to benchmark depending upon the relative number of functions and call-sites present. However, Avgf is relatively high for *assembler* (12.80), *volrend* (11.40) and *blocks2* (7.50). This happens because *assembler* and *volrend* have many calls-sites for small leaf functions (mostly error routines), while *blocks2* has several calls inside loops where some of these calls are also recursive.

  Further, since the calls to leaf functions get memoized, Avgc and Avgi for *assembler* and *volrend* are comparable with other benchmarks. However, this is not the case for *blocks2*, for which both Avgc and Avgi turn out to be 5.36. The fixed-point computations for loops containing procedure calls, which in turn are recursive and involve further fixed-point computations, contribute to the higher averages for this benchmark.

Thus our interprocedural algorithm works effectively for programs whose interprocedural structure is not very complex. To be able to handle a broader range of programs, we are planning to optimize our algorithm in the following ways: (i) excluding the functions from the invocation graph, which neither update nor access pointer variables, (ii) building the invocation graph in a lazy manner, as the demand for different invocation contexts arises during the analysis, and (iii) performing more extensive memoization as described above.

6.2 Shape Analysis Results

In this section, we present empirical results for shape analysis. Table 6.6 gives the benchmark programs used for this purpose, and their important characteristics. The characteristics reported are same as for connection matrix benchmarks in Table 6.1. However, the columns labeled vars give the number of variables abstracted by direction/interference matrices. The first ten benchmarks in Table 6.6, have been specifically selected to highlight the power as well as the limitations of the direction and interference matrix abstractions. The rest of the benchmarks are taken from the connection matrix suite, to verify the effectiveness of shape analysis for larger C applications.

It can be observed from the data on connection matrix benchmarks, that the number of variables abstracted by direction/interference matrices is higher than that abstracted by connection matrices. This happens because direction/interference matrix abstractions use one extra symbolic variable for each symbolic variable used by connection matrix abstraction (sections 4.3.5 and 5.5).

Another important observation from Table 6.6, is that for the benchmark *misr*, 27 indirect references can refer to both stack and heap locations. This happens because *misr* implements a linked list with its first element on the stack and the rest inside the heap. Consequently, the pointer used to traverse the list inside a loop, is reported by points-to analysis, to be possibly pointing to both the first element of the list on stack and to the abstract location *heap*.

The main goal of shape analysis is to identify the shape of the data structures built and used by a program. By identifying completely unaliased tree-like recursive

Program	Source Lines	SIMPLE stmts	Min vars	Max vars	Avg vars	Ind Refs	To Stack	To Heap	Stack/ Heap
bintree	351	342	4	23	10	50	10	40	0
hash	257	110	4	6	11	14	7	7	0
xref	153	139	20	40	24	31	0	31	0
misr	277	235	2	10	8	47	39	35	27
stanford	885	880	4	14	7	28	0	28	0
power	681	641	16	23	18	180	29	151	0
chomp	430	476	20	27	22	127	45	82	0
reverse	123	49	9	18	12	16	0	16	0
paraffins	381	180	6	31	21	37	2	35	0
assembler	3361	3071	22	36	24	718	666	52	0
loader	1539	1055	13	28	17	170	106	64	0
volrend	4207	4909	36	65	38	190	63	128	1
sim	1422	1760	76	111	83	374	34	340	0
blocks2	876	1070	56	82	61	373	98	275	0
water	2703	2418	16	79	36	581	32	549	0
nbody	2204	703	24	36	27	134	24	116	6
sparse	2859	1495	24	60	32	468	3	465	0
pug	2400	2089	32	153	48	822	147	688	13

Table 6.6: Charateristics of Shape Analysis Benchmarks

structures or aliased but acyclic dag-like structures in a program, we can enable several powerful program optimizations like concurrent execution of recursive procedure calls [Lar89, Hen90], software pipelining [HHN92a], and loop unrolling [HG92]. In this light, we estimate the effectiveness of shape analysis, by providing the following measurements in Table 6.7, for each benchmark:

- Refs: The number of heap-related indirect references in the program.
- T, D, C: These three columns respectively give the number of heap-related indirect references where the dereferenced pointer, say `p`, points to a tree-like, dag-like or cyclic data structure: i.e. A[p.shape] = Tree, Dag or Cycle, where A is the attribute matrix at the given program point.

The multi-columns labeled `*a/(*a).b` and `a[i]` (where `a` is a pointer) in Table 6.7, separately give the above measurements for indirect references of the respective form, while the multi-column labeled Overall gives the overall statistics for the given program.

Below we analyze the results for each benchmark, by comparing the actual shape of the data structures it builds, with that reported by the analysis. For benchmarks where the two don't match, we investigate why the analysis gives a conservative answer.

bintree: As the name suggests this program builds a binary tree. Each tree node consist of a `char` pointer, and pointers to left and right children. The program first builds the tree using a loop, where each iteration inserts a node with a new string pointed to by the `char` pointer. A new node is always inserted as the child of an existing node with no children.

Thus the data structure always remains tree-like and does not become dag-like even temporarily. Once the tree is built, several traversals are done on it. They only update the value of the string field in its nodes, and do not modify other pointer fields.

Our analysis gives precise results. It reports all dereferenced pointers to be pointing to tree-like structures. The indirect references of the form `*a/(*a).b` in this program refer to nodes of the binary tree, while the four indirect references of the form `a[i]` are used in a string compare function, where the strings are stored in dynamically allocated objects.

hash: This program builds a hash table. It uses an array of pointers on the stack, namely `htable`, and each of the pointers `htable[i]` points to a linked list of items.

Program	*a / (*a).b				a[i]				Overall			
	Refs	T	D	C	Refs	T	D	C	Refs	T	D	C
bintree	36	36	0	0	4	4	0	0	40	40	0	0
hash	7	7	0	0	0	0	0	0	7	7	0	0
xref	29	29	0	0	2	2	0	0	31	31	0	0
misr	35	35	0	0	0	0	0	0	35	35	0	0
stanford	28	28	0	0	0	0	0	0	28	28	0	0
power	147	147	0	0	4	4	0	0	151	151	0	0
chomp	56	56	0	0	26	26	0	0	82	82	0	0
reverse	16	11	5	0	0	0	0	0	16	11	5	0
paraffins	26	8	18	0	9	3	6	0	35	11	24	0
assembler	45	45	0	0	7	7	0	0	52	52	0	0
loader	55	55	0	0	9	9	0	0	64	64	0	0
volrend	96	96	0	0	32	32	0	0	128	128	0	0
sim	96	29	67	0	244	221	23	0	340	250	90	0
blocks2	119	16	37	66	156	64	43	49	275	80	80	115
water	250	181	0	69	299	124	0	175	549	305	0	244
nbody	74	22	0	52	42	14	0	28	116	36	0	80
sparse	384	14	0	370	0	0	0	0	384	14	0	370
pug	514	16	0	498	174	1	0	173	688	17	0	671

Table 6.7: Empirical Measurements for Shape Analysis Results

An item is added by first hashing it to a number `j`, and then appending it at the end of the list pointed to by the pointer `htable[j]`.

Our analysis abstracts all pointers `htable[i]` by the single name `htable`. The shape attribute of `htable` is therefore the merge of the shape attributes of all pointers it represents. Since each of these pointers points to a tree-like structure, our analysis reports the shape attribute of `htable` as Tree. Note that if the lists pointed to by two pointers `htable[i]` and `htable[j]` share a node, then the analysis would report the shape attribute of `htable` as Dag or Cycle.

Consider the example in Figure 6.6. Before the statement, pointer `htable` has paths to both `p` and `q` as it represents both pointers `htable[i]` and `htable[j]`. Further, the shape attribute of all three pointers (`htable`, `p` and `q`) is Tree. After the statement `p->next = q`, analysis would infer that `htable` has an additional path to `q` via `p` and make its shape attribute as Dag.

Further, in the example in Figure 6.7, pointers `htable[i]` has a path to `p`. Thus, from the analysis point of view `htable` has a path to `p`. Now, after the statement `p->next = htable[j]`, analysis finds that `p` has a path to `htable`, while `htable` already has a path to `p`. So it infers the creation of a cycle, and makes the shape attribute of both `htable` and `p` Cycle.

From the above discussion, we can conclude that if a pointer `p` represents an array of pointers, and our analysis reports its shape attribute as Tree, the data structures accessible from all pointers `p[i]`, are tree-like and more importantly are completely disjoint from each other. This information is crucial, as a loop iterating over such an array, would access disjoint heap locations in each iteration, and hence can be potentially parallelized.

Finally, note that *hash* does not have any heap related indirect references of the form `a[i]`. This is because access to an array of pointers on stack, `p[i]` is simply a pointer reference and not a pointer dereference.

xref: This program builds a binary tree of items (character strings) for cross referencing purposes. Thus each tree node also has a pointer pointing to a linked list of items. The overall shape of the data structure is tree-like and our analysis accordingly reports all dereferenced pointers having their shape attribute as Tree.

The benchmark *misr* uses a linked list, *stanford* implements a tree sort algorithm, while *chomp* implements a game tree and also uses a linked list. We get expected results from the analysis for all three benchmarks.

power: This porgram implements the *Power System Optimization* problem [LML+93]. It represents the power network as a tree with the power plant as the root and

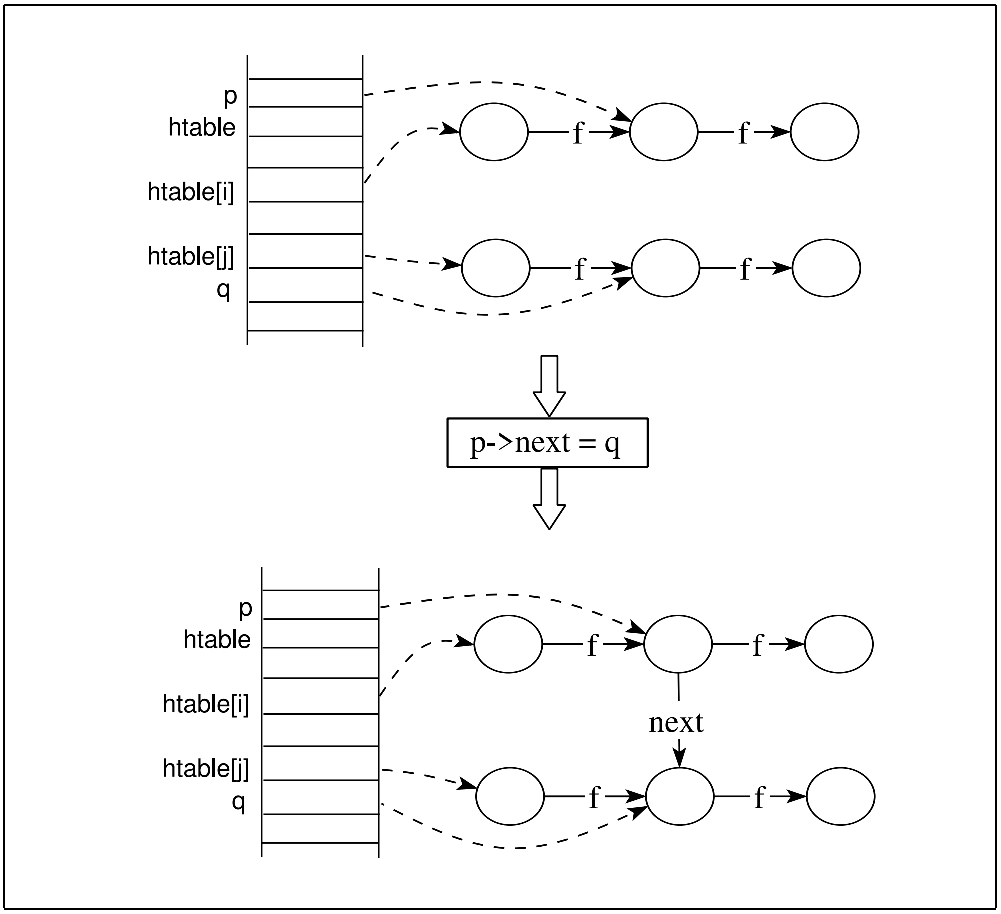


Figure 6.6: Shape Attribute Becomes Dag due to Array of Pointers

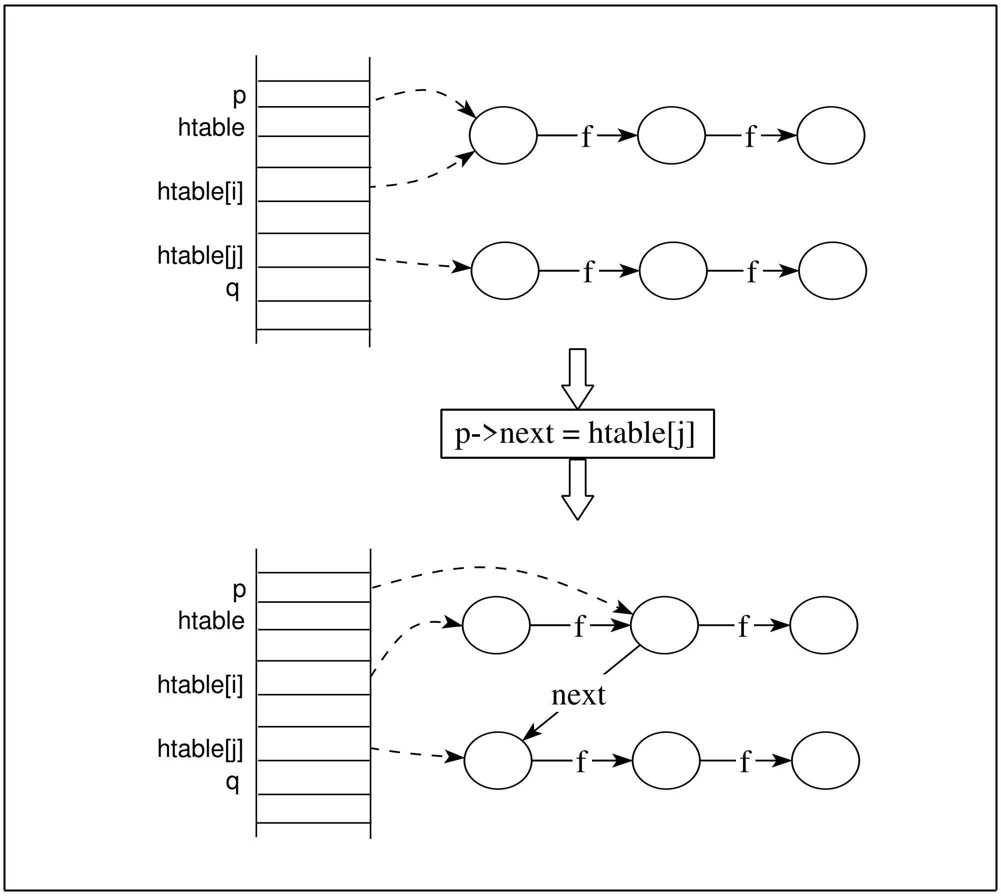


Figure 6.7: Shape Attribute Becomes Cycle due to Array of Pointers

customers as leaves. The root node has an array of pointers called feeders, pointing to various lateral nodes. Each lateral node has a pointer to a branch node and the next lateral node. Each branch node used a pointer to the next branch, and an array of pointers called leaves, pointing to customer (leaf) nodes.

We analyze the sequential version of the this program, originally implemented for Olden [CRRH93] by Martin Carlisle. Our analysis reports the shape attribute of the data structure built by this program, as Tree. After constructing the power network tree, it propagates pricing information from the root node to the customer nodes, and demand information from customer nodes to the root node.

The main loop in this program iterates over the feeder array in the root node, and each iteration computes this information for the feeder corresponding to the current index. Now, feeder array is an array of pointers and our analysis reports its shape attribute as Tree. This implies that all array indices point to disjoint data structures, and the loop can be parallelized.

reverse: This is a small program that builds a binary tree and then recursively swaps the left and right children of each node using the following procedure:

```
reverse(bintree *t)
  { if (t == NULL)
      return;
    l = t->left;
    r = t->right;
    reverse(l);
    reverse(r);
    t->left = r;
    t->right = l;
  }
```

Before the call to `reverse`, our analysis reports the shape of the data structure to be Tree, and after the call it reports it to be Dag. Note that actually the binary tree only temporarily becomes a dag after the statement `t->left = r` when pointer `t` has paths to pointer `r` via both `left` and `right` links. It becomes a tree again after the statement `t->right = l`, when the `right` link is reset. Shape analysis is able to identify the first situation and makes the shape attribute of `t` as Dag. However, it does not record the information as to why `t` becomes a Dag. So it cannot identify that the statement `t->right = l` restores the tree attribute of pointer `t`.

This example exposes a major limitation of our analysis. Analyses using more powerful abstractions like path matrices [HN90] can handle such cases, but they also incur higher cost.

paraffins: This program generates all the paraffins of size upto n, where paraffins are molecules of chemical formula C_nH_{2n+2}. The principal data structure built by this program is shown in Figure 6.8. The variables `BCP`, `CCP` and `Radicals` are arrays of pointers located on the stack. The array `Radicals` is similar to the structure shown in the example in Figure 6.6. So it is reported to be Dag by the analysis. Further, since this structure is accessible from arrays `BCP` and `CCP`, their shape is also reported to be Dag. In Table 6.7, majority of indirect references for *paraffins* fall in the Dag category. The ones falling in the Tree category, represent references to newly allocated nodes, before they are hooked in the main data structure.

The benchmarks *assembler* and *loader* implement linked lists, while *volrend* uses a stack-based array of pointers pointing to to bit vectors allocated in the heap, all of which are disjoint from each other. The analysis results for these benchmarks are consistent with the actual shape of data structures used by them.

sim: This program dynamically allocates an array of pointers to structures, which in turn themselves are dynamically allocated. The stack-based pointer `LIST`, that points to the array of pointers, is reported to be dag-like because of the following code fragment:

```
S1: LIST[I] = LIST[J];
S2: LIST[J] = newStruct();
```

After `S1`, both the pointers `LIST[I]` and `LIST[J]` point to a common heap-allocated structure. The stack-based pointer `LIST` has two paths to this structure: both via `LIST[I]` and `LIST[J]`, so its shape is reported to be Dag. However, after `S2`, `LIST[J]` points to a newly allocated structure, and `LIST` no longer points to a dag-like structure. Direction matrix abstraction cannot infer kill information for a particular array index, and the shape attribute of `LIST` remains as Dag.

This program also builds a linked list, which is reported to be dag-like because of the following code fragment, where `z` points to a linked list and `row` is a dynamic array of pointers:

```
S1: z->NEXT = row[I];
S2: row[I] = z;
```

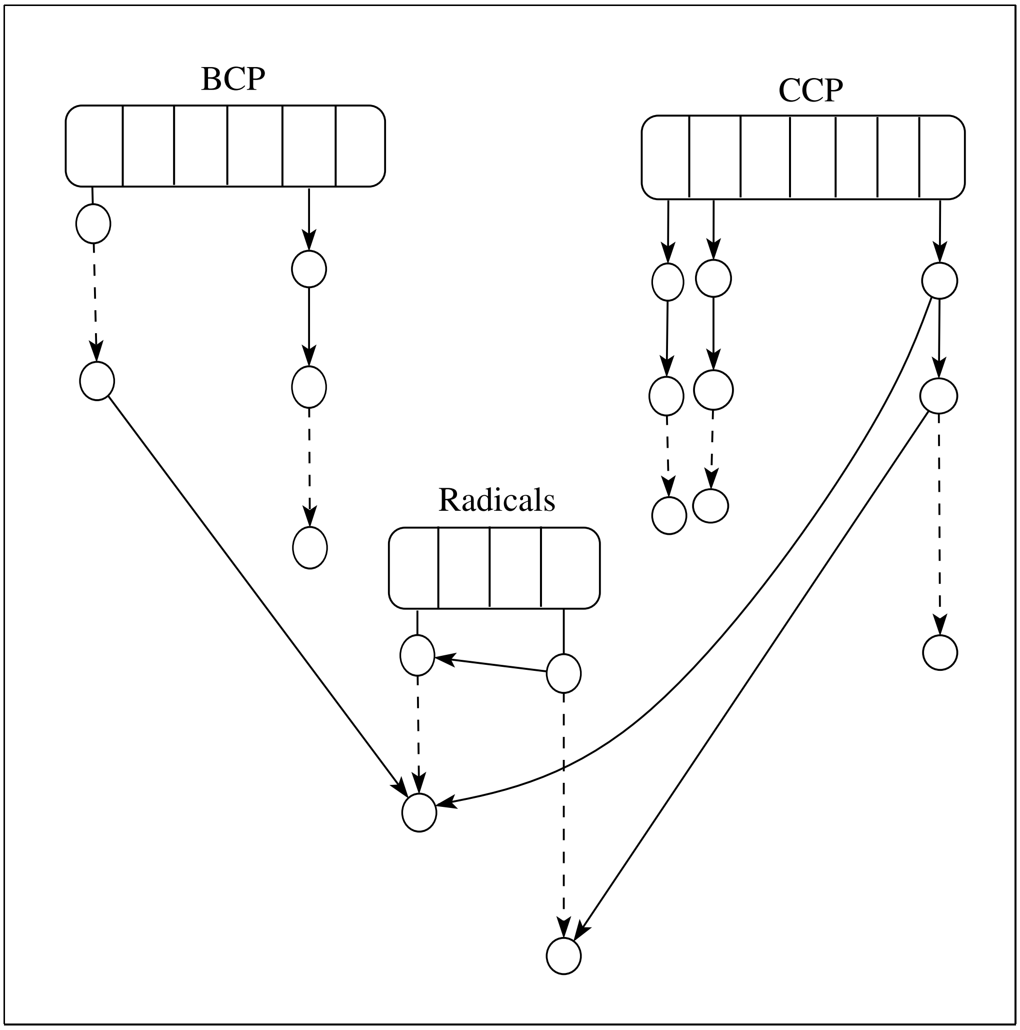


Figure 6.8: Data Structure Built by *paraffins* Benchmark

Program	fns	call sites	ig nodes	Recur nodes	Appr nodes	nodes/ call
bintree	17	31	32	2	4	1.03
hash	5	8	8	0	0	1.00
xref	8	14	15	2	4	1.07
misr	5	7	7	0	0	1.00
stanford	8	12	13	2	4	1.08
power	18	31	53	6	6	1.71
chomp	20	47	98	7	7	2.09
reverse	5	10	11	2	4	1.10
paraffins	7	6	7	0	0	1.16
nbody	34	67	118	2	2	1.76

Table 6.8: Invocation Graph Characteristics for Shape Analysis Benchmarks

Analysis fails to identify that the same index of the array `row` gets updated in `S2` and gives a conservative answer.

The benchmarks *blocks2* and *water* use linked lists and heap-allocated arrays of pointers to linked lists. These arrays become dag-like or cyclic because of situations similar to the ones depicted in Figures 6.6 and 6.7, except that arrays of pointers for these benchmarks are heap-allocated. In Table 6.7, we have indirect references for *blocks2* refering all three types of data structures. While for *water*, they are reported to refer to only tree-like and cyclic data structures.

nbody: This program implements the hierarchical N-body problem to calculate gravitational forces acting on N bodies in space and computing their new co-ordinates at each time step. It stores the N bodies as a linked list, and builds an octree data structure to represent the relative position of the bodies in three dimensional space.

The root of this tree represents a space cell containing all bodies in the system. The tree is built by adding bodies into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a single body. In the process, new cells are generated and they are inserted between an existing cell and a leaf node (a body) in the tree. This insertion causes the analysis to infer the shape of the data structure as Dag. For example, see the following code fragment:

```
S1: old_cell->item = q;
S2: new_cell = newCell();
S3: new_cell->item = q;
S4: old_cell->item = new_cell;
```

After `S1`, `old_cell` has a path to `q`. After `S3`, `new_cell` also has a path to `q`. Presently the shape of the data structures accessible from both `old_cell` and `new_cell` is Tree. After `S4`, `old_cell` has a path to `new_cell`, and it does not have a direct path to `q` (not considering the one via `new_cell`). However, shape analysis does not have enough information to infer the latter. Hence it assumes that old_cell has two paths to `q`: a direct path and a path through `new_cell` and reports its shape to be Dag. If more insertions are done in this apparently dag-like data structure, analysis gets overly conservative and finally reports it to be cyclic.

For the above reasons, we have majority of indirect references in *nbody* falling in the Cycle category. Note that the octree data structure built by this program is inherently a dag-like structure, as its leaves are linked. However, our in-depth analysis of this program revealed that we would get the same results even if this were not the case.

Finally, the benchmarks *sparse* and *pug* use inherently cyclic data structures with back pointers. So majority of indirect references for them fall in the Cycle category. The ones in the Tree category again represent newly allocated nodes, before they are hooked in the main data structure of the program.

One can note that majority of the indirect references of type `a[i]` fall in the Tree category. This happens because for most of the benchmarks, these arrays are stand-alone data structures in the heap, which can be considered as trees with a single node. However some benchmarks like *paraffins*, *blocks2* and *water* use heap allocated arrays of pointers to recursive structures like linked lists. Indirect array references for such benchmarks fall into all three shape categories.

From the above analysis of shape analysis results for the various benchmarks, we draw the following conclusions:

- If a program builds a tree-like data structure in such a manner, that a new node is always appended at the beginning/end of the existing structure, then shape analysis is able to infer the shape of this data structure as Tree. For example, in the above benchmark set, *bintree* and *stanford* build binary trees by appending the new node to a leaf node, while *hash*, *misr*, and *xref* build linked lists by appending a new item at the beginning/end of the list.

Program	Calls Analyzed			Avgf	Avgc	Avgi
	Tot	Memo	Actual			
bintree	59	12	47	2.76	1.51	1.47
hash	11	1	10	2.00	1.25	1.25
xref	60	13	47	5.88	3.36	3.13
misr	7	0	0	1.40	1.00	1.00
stanford	36	6	30	3.75	2.50	2.31
power	112	49	63	3.50	2.03	1.19
chomp	390	196	194	9.70	4.13	1.98
reverse	52	16	36	7.20	3.60	3.27
paraffins	9	0	9	1.29	1.50	1.29
nbody	252	42	210	6.17	3.13	1.78
assembler	1057	221	836	16.08	3.18	1.30
loader	328	126	202	6.73	2.46	1.62
volrend	247	26	221	4.17	2.05	1.31
sim	161	8	153	10.93	6.19	3.66
blocks2	690	432	258	12.90	9.21	8.90
water	366	332	34	2.27	1.62	1.31
sparse	195	64	131	4.68	1.72	1.08
pug	213	53	160	3.90	2.39	1.36

Table 6.9: Interprocedural Measurements for Shape Analysis

- If a new node is inserted between two existing nodes in a data structure, analysis fails to retain the shape attribute of the data structure as Tree and infers it to be first a Dag, and on successive insertions a Cycle. The construction of the octree structure for the *nbody* benchmark is an example of this case.

- If a data structure temporarily becomes dag-like or cyclic and then becomes tree-like again, shape analysis cannot identify this case, and continues to report it as dag-like or cyclic. The program *reverse* demonstrates this limitation of the analysis. A more powerful abstraction is required to overcome this limitation.

- Abstracting entire arrays (of pointers) as one pointer, has both advantages and disadvantages. On the positive side, if the shape attribute of such a pointer is reported to be Tree, one is guaranteed that all indices of the array point to tree-like structures, completely disjoint from each other. This applies to benchmarks *hash* and *power*.

  On the negative side, such abstraction can introduce undesirable imprecision, as demonstrated by the cases shown in Figures 6.6 and 6.7, which apply to benchmarks *paraffins*, *blocks2* and *water*. Further, shape analysis cannot obtain any kill information when an array index is updated. This lack of kill information results in loss of precise shape information, as shown for the benchmark *sim*.

  One approach to avoid this imprecision can be, to use subscript analysis for arrays of pointers, and identify when two array references access the same pointer or distinct pointers. We are presently investigating this approach.

Finally, the interference matrix information itself can be used to find if two pointers can access a common heap object. This can provide an improvement over connection matrix information, which only informs if two pointers can lead to a common (connected) heap data structure. However, our experiments showed that this is not the case, and interference matrix results turn out to be almost identical to connection matrix results. On further investigation, we discovered that during the analysis, interference matrix information is more sophisticated than connection information, but the final merged information deposited on the SIMPLE tree turns out to be identical for the two abstractions. Now, interference matrix abstraction was designed to use its information *during* the analysis, in order to improve the overall precision of shape information. Thus, it serves its design target.

In Table 6.8, we present the invocation graph characteristics for shape analysis benchmarks, and in Table 6.9 some interprocedural measurements for shape analysis. The two tables present similar data for shape analysis, as Tables 6.4 and 6.5 present for connection analysis. Further in Table 6.8, we do not include the shape analysis benchmarks taken from the connection matrix suite, as the data for them is same as that presented in Table 6.4.

It can be observed from Tables 6.4 and 6.8 that majority of the shape analysis benchmarks, have recursive and approximate invocation graph nodes. Since most of these programs use recursive data structures, they also employ recursion as the control structure to traverse and modify them.

The interprocedural measurements show similar trends as the ones for connection analysis, with memoization providing significant advantages. The averages Avgf, Avgc, and Avgi are relatively higher than those for connection analysis. This happens because more iterations are required for loop and recursion fixed-point approximations for shape analysis, as four separate abstractions are calculated at the same time.

6.3 Summary

In this chapter we provided empirical evidence of the effectiveness of our heap analysis techniques on real C application programs. We also demonstrated that each of our analyses gives accurate results for its target domain of applications, and conservative results for others. This proved the validity of our hierarchical approach to heap analysis. In future, we plan to collect empirical data on the impact of accurate heap analysis towards more accurate dependence analysis and increased opportunities for optimizations like loop parallelization and instruction scheduling.

Chapter 7

Conclusions and Future Work

In this thesis, we have presented a hierarchy of two practical heap data structure analyses: connection analysis and shape analysis. The first analysis identifies pointer accesses to completely disjoint heap data structures. The shape analysis estimates the shape of the data structure accessible from a heap-directed pointer, with special focus on identifying completely unaliased tree-like structures and aliased but acyclic dag-like structures, built by the program under analysis.

We outlined the basic analysis rules for these analyses, which can be used to develop the corresponding analyses for any language that supports dynamic data structures. Further, based on the basic analysis rules, we developed complete analysis frameworks to analyze a language as complex as C. We also extended the context-sensitive interprocedural analysis framework built by points-to analysis. Our extension involved introduction of additional symbolic variables, to correctly handle the heap relationships of *inaccessible* local pointers of a function, across procedure calls.

We also implemented the connection and shape analyses at the structured tree-based SIMPLE intermediate representation of the McCAT C compiler. Using this implementation, we analyzed a collection of real C applications, and provided empirical evidence of the effectiveness of each analysis for its intended target domain of applications. Connection analysis provided very accurate results for programs that allocate a number of (mostly non-recursive) disjoint data structures in the heap. Shape analysis accurately identified tree-like structures, when they were built by inserting new nodes as children of existing leaf nodes. These empirical results also validated our approach of decoupling the analysis of stack-directed and heap-directed pointers, and developing a hierarchy of analyses for heap data structure analysis. Finally, the interprocedural measurements demonstrated the benefits achievable from memoization, for a context-sensitive interprocedural analysis.

We have illustrated that heap data structure analysis can be both efficient and effective, provided that appropriate abstractions are designed in keeping with the data structures used by the target domain of application programs. In future, we plan to extend our work in several directions. We discuss these extensions below.

We plan to develop efficient techniques for performing heap interference (dependence) analysis using the data structure information from connection and shape analyses. A general approach for heap interference analysis has been proposed by Hummel et al. [HHN94]. We plan to develop some practical variations of their approach, suitable to the data structure information collected by our analyses. We also plan to investigate the use of shape information for optimizing transformations like pipelined and parallel execution of recursive calls traversing tree-like structures, and software pipelining and unrolling of loops iterating over lists and arrays of (disjoint) recursive data structures.

We plan to optimize our interprocedural analysis algorithm to be able to handle programs which have large invocation graphs. We mentioned several approaches for this optimization in chapter 6. We also plan to measure how much precision is gained by performing a context-sensitive connection analysis as compared to a context-insensitive analysis. Shape analysis, however needs to be context-sensitive, because one extraneous path information can result in loss of critical shape information.

Finally, we plan to design more powerful abstractions to analyze programs that (i) build data structures by *insertion* i.e. by inserting new nodes between existing nodes, and (ii) use complex cyclic data structures. While examining application programs that use dynamic data structures, we discovered that often they build data structures that are dag-like or cyclic in shape, but still have regular properties. Typical examples include leaf-linked trees, doubly linked lists, and trees with parent and/or sibling pointers. One way to accurately abstract such data structures is provided by language-based mechanisms like ADDS [HHN92a] and alias axioms [HHN94]. We have designed a new abstraction called partial path matrix, that can automatically capture the critical properties of these data structures, and also accurately handle node *insertions*.

Partial path matrix abstraction can be considered as an enhanced version of the direction matrix abstraction. Given any two heap-directed pointers `p` and `q`, the partial path matrix entry PP[p,q] contains the first links(s) of the possible path(s) from the heap object pointed to by `p` to the heap object pointed to by `q`. For example, if `p` points to the root of a tree, and `q` points to a node accessible from the root by following a `left` link and a `right` link, the entry PP[p,q] will be simply `left`. Keeping the first link information will provide us kill information for the basic heap statement `p->f = q`, and enable accurate estimation of the shape of data structures built by insertion.

More importantly, partial path matrix abstracts the shape of a data structure in a more comprehensive manner. Like shape analysis, it maintains an auxiliary attribute matrix called AP. However, AP abstracts the shape of a data structure along *specific links*. The basic idea is that the overall shape of a data structure accessible from a pointer may be dag-like or cyclic, but it can be still tree-like or dag-like if only specific links in the data structure are considered.

For example, a binary tree with parent pointers is a cyclic structure. However, if in some program segment, it is traversed only using the `left` and `right` links (i.e. its tree edges), we can consider its shape to be tree-like for this part of the program, and apply optimizing transformations applicable to tree traversals. Suppose a pointer `p` points to the root of such a tree. The shape attribute of `p` is abstracted by the matrix AP along each combination of links as follows:

AP[p.shape,left,left] = Tree	AP[p.shape,right,right] = Tree
AP[p.shape,parent,parent] = Tree	AP[p.shape,left,right] = Tree
AP[p.shape,left,parent] = Cycle	AP[p.shape,right,parent] = Cycle

It can be noticed that using this abstraction, shape attributes of other dag-like and cyclic data structures like leaf-linked trees, trees with sibling pointers and doubly linked lists, can be accurately and comprehensively captured. Presently, we are developing the basic analysis rules for the partial path matrix abstraction. We soon plan to implement it in the framework of the McCAT C compiler, and measure its effectiveness on real C benchmark programs.

Bibliography

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer, 1988.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[CRRH93] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. Early experiences with Olden. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of Lecture Notes in Computer Science, pages 1–20, Portland, Oregon, August 1993. Springer-Verlag. Published in 1994.

[CWZ90] D. R. Chase, M. Wegman, and F. K. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the ACM 17th Symposium on Principles of Programming Languages*, pages 157–168, 1990.

[Deu92] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, pages 2–13, April 1992.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[Don94] C. M. Donawa. A structured approach to the design and implentation of a backend for the McCAT C compiler. Master's thesis, School of Computer Science, McGill University, April 1994.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[EH94] A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of IEEE 1994 International Conference on Computer Languages*, May 1994.

[Ema93] M. Emami. A practical interprocedural alias analysis for an optimizing/parallelizing compiler. Master's thesis, School of Computer Science, McGill University, September 1993.

[Ero94] A. M. Erosa. A goto-elimination method and its implementation for the McCAT C compiler. Master's thesis, McGill University, May 1994.

[Ghi92] Rakesh Ghiya. Interprocedural analysis in the presence of function pointers. ACAPS Technical Memo 62, School of Computer Science, McGill University, Montréal, Québec, March 1992. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[Gua88] V. A. Guarna Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, volume 2, pages 212–220, 1988.

[HA93] W. Ludwell Harrison III and Z. Ammarguellat. A program's eye view of Miprac. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 512–537. Springer-Verlag, 1993.

[Har89] W. Ludwell Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[HDE+93] L. J. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 406–420. Springer-Verlag, 1993.

[HEGV93] L. J. Hendren, M. Emami, R. Ghiya, and C. Verbrugge. A practical context-sensitive interprocedural analysis framework for C compilers. ACAPS Technical Memo 72, School of Computer Science, McGill University, Montréal, Québec, July 1993.

[Hen90] L. J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, April 1990. TR 90-1114.

[HG92] L. J. Hendren and G. R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proceedings of the 4th IEEE International Conference on Computer Languages*, April 1992.

[HGS92] L. J. Hendren, G. R. Gao, and V. C. Sreedhar. ALPHA: A family of structured intermediate representations for a parallelizing C compiler. ACAPS Technical Memo 49, School of Computer Science, McGill University, Montréal, Québec, November 1992.

[HHN92a] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, June 1992.

[HHN92b] J. Hummel, L. J. Hendren, and A. Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3):243–260, September 1992.

[HHN94] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 218–229, June 1994.

[HN90] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distributed Computing*, 1(1):35–47, January 1990.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[Hud86] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986.

[ISY88] K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM TOPLAS*, 10(4):555–578, October 1988.

[JM81] N. D. Jones and S. S. Muchnick. *Program Flow Analysis, Theory and Applications*, chapter 4, Flow Analysis and Optimization of LISP-like Structures, pages 102–131. Prentice-Hall, 1981.

[JM82] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[KKK90] D. Klappholz, A. D. Kallis, and X. Kang. Refined C: An update. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 331–357. The MIT Press, 1990.

[KS93] N. Klarlund and M. Schwartzbach. Graph types. In *Proceedings of the ACM 20th Symposium on Principles of Programming Languages*, pages 196–205, January 1993.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4), December 1992.

[Lar89] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California, Berkeley, 1989.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, 1988.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[LML+93] S. Lummetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing. In *Proceedings of Supercomputing 93*, pages 243–249, November 1993.

[LR92] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[MLR+93] T. Marlowe, W. Landi, B. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[PCK94] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Sixth International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 37–56. Springer-Verlag, 1994.

[RM88] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 285–293, 1988.

[Sri92] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.

[Sta90] R. M. Stallman. Using and porting the GNU CC. Technical report, Free Software Foundation, Cambridge, Massachusetts, 1990.

[SWG91] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report No. CSL-TR-91-469, Computer Systems Laboratory, Stanford University, Stanford, California, June 1991.

[WH92] E. Wang and P. N. Hilfinger. Analysis of recursive types in LISP-like languages. In *Proceedings of the '92 ACM Conference on LISP and Functional Programming*, pages 216–225, June 1992.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, Massachusetts, 1989. In Research Monographs in Parallel and Distributed Computing; revised version of the author's Ph.D. dissertation published as Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.

Appendix A

Implementation Details

The connection and shape analyses have been implemented on the SIMPLE intermediate representation, using the structured analysis framework of the McCAT C compiler [Sri92, Ema93]. We handle C language in its full glory, except for *setjump* and *longjump*, union types, exception handling and type-casting of pointers to integers and vice versa. For union types, presently we assume the existence of all fields. However, we also need to take into account that these fields are statically aliased. This is a minor extension to our implementation.

Both connection and shape analyses are performed after points-to analysis. They require points-to information and also depend on the interprocedural analysis framework built by points-to analysis. However, connection and shape analyses themselves are independent of each other. The user can invoke the particular analysis he/she wants by setting the appropriate optimization flag at compile time.

For either analysis, a first pass is done through the program (after points-to analysis) to determine which pointers can point to heap locations i.e. the abstract location *heap*. If a pointer can point to heap at some point in the program, a row index is reserved for it in the appropriate matrix. A separate matrix is constructed for each function in the program. The matrix for a given function consists of two parts: the *global* part which consists of rows allocated for variables global in scope, and (ii) the *local* part which consists of rows allocated for variables local to the given function. The size of the global part is same for matrices of all functions, while the size of the local part varies from function to function.

When a pointer is added to a matrix (i.e. a row index is reserved for it in the matrix), the symbolic variables corresponding to it are also simultaneously generated and added to the matrix. Recall that symbolic variables need to be generated corresponding to all heap-directed pointers which are global in scope, are formal parameters or represent symbolic variables used by points-to analysis (sections 4.3.5 and 5.5).

Thus, the number of rows in the matrix of any function, is equal to the number of pointers in the program that can be heap-directed at some program point and are visible in the function, plus the number of symbolic variables corresponding to these pointers. Once the number of rows is known, we implement the matrix by allocating a bit vector for each row. Each bit vector should have one bit corresponding to each row (i.e. each pointer abstracted) in the matrix. The number of bytes required for a bit vector is calculated by dividing the number of rows in the matrix by eight, where each byte is assumed to have eight bits, and taking the ceiling in case a fraction is obtained.

The attribute matrix used to store root and shape attributes, is appended to direction matrix in the implementation. It requires three bits for each pointers. The first bit is used to store the root attribute: if it is zero, root attribute is True (default value), else it is False. The remaining two bits are used to store the shape attribute. Since four shape attributes can be represented using two bits, while we need to abstract only three, we do not use one combination of bits. The choice is motivated by making the merge operation for attributes and pointer relationships uniform. So we follow the following convention:

Bits	Shape
00	Tree (Default)
01	Dag
10	Not Used
11	Cycle

It can be observed that logical OR operation on any two bit sets, gives the bit set for the appropriate attribute. So the three bits used to store attributes of a pointer are appended to the bit vector corresponding to it, in the direction matrix. Thus for bit vectors for direction matrix, we need to allocate space for three additional bits.

For matrices that abstract symmetric relationships, like connection and interference matrices, space is allocated only for the lower half of the matrix i.e. for the bit vector corresponding to the *i*th row, space is allocated to store only i bits. Consequently, an access like M[i,j] to a symmetric matrix M, is converted to an access to the element M[j,i] if j > i. This scheme results in substantial space savings.

The maximum of the average number of variables abstracted by a connection matrix is 89 for the benchmark *cholesky* (Table 6.1). For this size, if we implement the full matrix, each bit vector needs space for storing 89 bits. Thus, 12 bytes need to be allocated for each bit vector, and total ($89 * 12 = 1068$) bytes for the entire matrix. Since we allocate only the lower half of the matrix, the space requirement should be ($1068/2 = 534$) bytes. In practice it turns out to be larger, because the basic unit of allocation used by our analysis is a word (4 bytes). Thus, even if a bit vector needs to store only one bit, a word is allocated for it. Taking this factor into account, the number of bytes allocated for the above matrix can be calculated as follows: ($(32 * 4) + (32 * 8) + (25 * 12)$) = 684).

Among direction matrix benchmarks, the maximum of the average number of variables abstracted is 83 for *sim*. Since, direction and interference matrix abstractions are computed simultaneously, we take into account the space requirements for both the matrices. Considering that direction relationships are not symmetric, while interference relationships are, the total space requirement for the two matrices with 83 variables is: ($(83 * 12) + (32 * 4) + (32 * 8) + (19 * 12) = 1680$ bytes). Note that ($83 + 3 = 86$) bits will be used for each direction matrix bit vector from the total space for 96 bits allocated. Thus, it can be noticed that space requirement for direction/interference matrix abstractions is much higher than that for connection matrix abstraction, when the number of variables abstracted is comparable.

For heap analyses, bit vectors form the suitable data structures, as the number of relationships between heap-directed pointers can be quite large. For example, if the program uses a single data structure, every pointer will be connected with every other pointer. Similarly if a data structure is cyclic, all pointers pointing to it would have paths to each other. Further, bit vectors enable fast merge operation on matrices, which is simply a logical OR operation.

Our analysis framework uses a global data structure called `DATA`. It stores the currently valid matrix, and pointers to other structures used by the analysis, like `break-list`, `continue-list` and `return-list` (described in chapter 4). The fields of this data structure get updated as analysis proceeds from statement to statement. When the analysis visits a statement, it obtains the currently valid matrix (or matrices) from `DATA`, and stores it (them) in the SIMPLE node corresponding to the statement. If a matrix is already deposited at the statement node due to a previous visit, the two matrices are merged.

Next, the statement is analyzed and the matrix stored in `DATA` is updated if required. The updated matrix is then saved at the next statement visited by the analysis. Thus, our scheme cleanly separates the data structures used for calculating the abstraction, from the ones used to store it for further use. Further, the matrix deposited at a statement node contains information valid *before* the execution of the statement.

The space requirements estimated above for the various matrices, indicate that storing them at each statement node can prove to be expensive. Further, if a statement sequence involves no pointer updates, the matrices for all statements in the sequence would be identical. Thus, substantial space optimizations are possible.

Presently, we employ a simple scheme that avoids the duplication of matrices at the basic block level. Under this scheme, we store the matrices at basic SIMPLE statements which follow a statement that accesses a pointer variable (and hence can affect pointer relationships). They are also stored at statements falling before and after a call-site, and at the entry of a function or a control construct. The rest of the statements simply have a pointer to the matrix stored in the nearest preceding statement. The matrices valid at the exit of a function are stored in the corresponding function declaration node in the SIMPLE tree. The above scheme reduces storage requirements, while allowing us to output the analysis results at each statement, for debugging purposes. More effective space optimization can be achieved by using advanced intermediate representations like ALPHA [HGS92].

Another approach can be to store the matrices only at statements that dereference heap-directed pointers, because it is at these statements where the analysis information really gets used. Further, at these statements we need to store only the information for the pointers being dereferenced, instead of the entire matrix (matrices). If the information about the dereferenced pointers turns out to be sparse, it can be stored as a linked list instead of as a bit vector. This approach can be quite easily implemented in our analysis framework. We do not use it in our present implementation, as we need the analysis information at each statement in order to study the factors that influence various heap analyses.

Besides the data structure `DATA`, another important data structure used by the heap analyses is the invocation graph. Each path in the invocation graph represents a call-chain, while each node represents a call in the given call-chain. Each node stores the pair of input/output matrices last seen (to enable memoization), and a pointer to the map information data structure. Map information is stored as a dynamically allocated array of integers called `map_info`, where `map_info[i]` gives the index of the variable in callee to which the variable with index `i` in caller is mapped. If a variable in caller, say with index `j`, is not mapped to any variable in the callee, we set `map_info[j]` as -1.

Finally, we need to optimize our analysis in a number of other ways: most importantly by minimizing the calls to the memory routines `malloc` and `free`. With our current implementation, connection analysis takes less than 15 seconds for the benchmarks given in Table 6.1, while shape analysis can take upto 354 seconds for heap-intensive benchmarks like *pug*. We will make more detailed timing data available once we fine-tune our implementation.