Parallelizing Programs with Recursive Data Structures

Laurie J. Hendren

Ph.D Thesis

TR 90-1114
April 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

PARALLELIZING PROGRAMS WITH RECURSIVE DATA STRUCTURES

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Laurie J. Hendren
January 1990

Abstract

Interference estimation is a useful tool in developing parallel programs and is a key aspect of automatically parallelizing sequential programs. Interference analysis and disambiguation mechanisms for programs with simple data types and arrays have become a standard part of parallelizing and vectorizing compilers. However, efficient and implementable techniques for interference analysis in the presence of dynamic data structures have yet to be developed.

This thesis addresses the problem of estimating interference and parallelizing programs in the setting of an imperative language that supports dynamic data structures. By focusing on analysis methods for recursively defined pointer data structures such as trees and DAGs, we have developed efficient and implementable interference analysis tools and parallelization techniques.

The interference analysis methods are based on estimating the the relationships between accessible nodes in a data structure. We define a data abstraction for estimating relationships that leads to an efficient interference analysis. Analysis functions are provided for SIL, a simple imperative language that includes conditionals, loops and recursive procedures. The analysis is proven sound with respect to the standard semantics for SIL. Based on the interference analysis tools, a collection of parallelization techniques are developed and the coarse-grain techniques are used to develop a simple system for parallelizing programs for a shared memory machine. The analysis techniques and parallelization system have been implemented, and examples illustrating the methods are provided.

Biographical Sketch

Laurie Jane Hendren was born in Peterborough, Ontario, Canada on a frosty winter day, December 13th, 1958. After a happy childhood spent mostly in Peterborough and at Chandos Lake, Laurie and her cousin "Susie" Galway launched themselves into their university careers at Queen's University in Kingston, Ontario.

Although initially intending to study chemistry, she soon found that she preferred punching cards and counting columns, to hunting for ions in the "chem lab". After two years of computer science, she forgot her experience in the "labs", and did a brief stint in medical school. Finally returning to computer science once and for all, she received her B.Sc. (Honours) and M.Sc. degrees in Computing and Information Science from Queen's University at Kingston, Canada. While at Queen's she was a member of the development group for the Nested Interactive Array Language, Q'Nial.

Laurie arrived in Ithaca in the fall of 1985, only to find that her office was in yet another chemistry lab. Luckily, this lab was full of computer scientists who thought that fume hoods were for storing empty coke bottles. Soon after arriving she had her spot staked out at Oliver's, the Friday A.I. tradition firmly entrenched, and the Q-exam jitters. After passing most of the qualifying exams she escaped to Cambridge for six months where she learned the real meaning of cold and damp, and learned the joys of programming in ML. Upon returning, to an office that had never been a "chem lab", she began her thesis work. Her search for a thesis avoidance activity quickly ended with the founding of the department's women's ice hockey team, the "Flying Diskettes". After a 4-2-0 season on the ice, and a 2-1-0 season on the conference circuit, she finds herself ready to be graduated. Oops, she forgot to mention that she got married in August 1988!

Acknowledgements

I would first like to thank all of my relatives, and in particular my parents who have always provided a supportive environment, and have encouraged me in all my endeavours.

During my career as a computer scientist, I have been fortunate in meeting a diverse collection of supportive colleagues. I would like to express my appreciation to all of my friends at Queen's University who made the study of computer science interesting and exciting. In particular, I would like to thank Mike Jenkins who encouraged me to broaden my horizons.

At Cornell, I found my niche working with my advisor Alex Nicolau, who has always listened to my ideas with an open mind and who has continued to support me from the far reaches of sunny California. Keshav Pingali has been a helpful member of my committee as well as providing many interesting and informative lectures in his courses. I would like to thank the members of the Women's Studies field who provided me with an interesting minor and an opportunity to attend classes led by women. Kathryn March deserves special thanks for serving as the minor committee member. I would like to acknowledge Prakash Panangaden and Radhakrishnan Jagadeesan for their invaluable help with the semantic formalizations and the "taming of the soundness proof". I would also like to thank Anne Neirynck and Anne Rogers for their careful reading of preliminary papers and thesis drafts.

I would like to thank all those who support the tools of the trade: the members of the CER who provided ample computing power, John Reppy and Standard ML of New Jersey support group who provided ever-improving versions of SML, and the thesis macro writers who came before me.

My stay at Cornell has been made more pleasurable by the many friends that I have met. Prakash is a special friend who I won't have to leave. My office mates, Nax Mendler, Olga Peschansky, Aleta Ricciardi (Ms. A.R.), and Anne Rogers (Miss. A.R.) have provided a stimulating environment. Special thanks to Nax for his help during the Q-exam years, to Anne for her interest and help with my thesis, and to the N.A. Gods for coming through in the clutch.

There are so many others that it is hard to mention them all, but a special mention to Dan Dan P. Huttenlocher and Bruce-O Donald. Finally, the "hockey gang" has provided a great diversion from work, and the "Flying Diskettes" provided an entertaining way of bringing the women in the department together.

To my maternal grandmother Edith Galway (née Lane),
and to the memory of my paternal grandmother Elsie Hendren (née McIlvena).

Contents

List of Figures

Chapter 1

Introduction

The emergence of parallel architectures holds the promise of faster execution of programs. Unfortunately, dealing with parallelism adds a new dimension to the design of algorithms and programs, thereby further increasing the complexity of programming. Traditional programming languages do not provide any mechanism for handling this added complexity. One approach to handling parallelism is to design a language that includes constructs for expressing parallel computations directly. With this approach, the onus is on the programmer to express which computations may be done in parallel. A second approach is to use a more conventional programming language, but to provide a parallelizing compiler to extract parallelism. Thus, a sequential program is converted into an equivalent parallel program in which some computations may be done in parallel.

When the underlying programming language is imperative, alias analysis and interference detection are central to both approaches. Two computations *interfere* if one computation writes to a location that the other computation reads or writes. Two sequential computations may be safely executed in parallel when they do not interfere. In the case of a parallel programming language, interference analysis can be used to assist the programmer in program development. When developing a parallel program, the programmer may inadvertently specify computations that interfere. Such interference may lead to non-deterministic bugs which are extremely difficult to detect. By incorporating interference analysis into the compiler for a parallel programming language, interfering computations can be detected at compile-time and diagnostic information can be reported to the programmer. In the case of a parallelizing compiler, interference detection is critical in exposing opportunities for parallelism.

In the setting of functional languages, data structures cannot be updated, and interference among computations is not an issue. However, when computing with large aggregate data structures such as trees or arrays, the functional approach may lead to excessive copying. In such cases, the ability to directly update a large data structure would be beneficial.

Many imperative programs which use recursive data structures exhibit a regular updating pattern. We are developing interference analysis tools and parallelization methods which take advantage of the regular nature of the data structure and the programs using the structure. By providing interference analysis tools for such data structures, we allow the programmer to specify imperative programs which have both the determinate behaviour of functional programs and the efficiency of updating in place.

Interference detection in the presence of dynamic data structures and pointers is a difficult problem that has been only partially solved. Intuitively, the difficulty lies in the lack of compile-time names for all allocated objects. Static scalar structures are the most straightforward to handle because they can be associated with an identifier name at compile-time [Ban79b, Bar78]. Aggregate structures such as arrays are more difficult to deal with. Although entire arrays can be associated with an identifier name, elements within an array have computed names. Thus, it is relatively easy to determine that two arrays X and Y are disjoint collections of objects or that the elements X[i] and Y[j] are different objects. However, it is substantially more difficult to determine that X[i] and X[j] are different objects. Since i and j are computed values, subscript analysis must be performed. Various techniques have been developed for analysing simple, regular array references [Ban79a, BC86, Nic84, Wol82]. As a result, significant progress has been made in parallelizing scientific programs in which the array references occur in a simple, regular manner [AK87, PW86]. However, with dynamic data structures, objects are allocated and linked together at run-time. Thus, not only is there no compile-time name for each individual object, there is also no simple way to name collections of objects. Just as array alias analysis can be done in many important regular cases, we show that alias analysis of dynamic structures can be done with recursive data structures such as trees. Furthermore, we propose that such techniques can be used for interference detection and parallelism extraction in programs which use dynamic data structures in a regular (recursive) manner. Our methods apply to imperative programming languages which have procedures, functions and recursion.

1.1 Related Work

Various approaches have been suggested for analysis of programs in the presence of dynamic data structures and pointers. Jones and Muchnick suggested one of the first approaches with a flow-analysis for Lisp-like structures [JM81]. This analysis was defined on a simple language that did not include procedures, and was designed to statically distinguish among cells that could be immediately deallocated, cells that could be reference counted, and cells that should be garbage collected. With this approach they introduced the notion of using *k-limited* graphs as a finite approximation to linked structures built by a program. A *k-limited* graph can have paths of length at most *k*, and thus must approximate many nodes in the actual data structure with one node in the abstract data structure. Although an interesting abstraction for statically estimating which cells should be garbage collected, the information is not precise enough to be used in the context of interference analysis for parallelization.

Jones and Muchnick [JM82] have also proposed a general purpose framework for data flow analysis on programs with recursive data structures. This method uses tokens to designate the points in a program where recursive data structures are created or modified. A retrieval function is then defined to finitely represent the relationships among tokens and data-values. By varying the choice of token sets and approximation lattices, a wide range of analyses can be expressed in this framework. Although flexible, the method is mostly of theoretical interest and is potentially expensive in both time and space.

Another approach to static analysis for dynamic structures has been proposed by Neirynck [Nei88, NPD87]. This method uses abstract interpretation techniques to provide information about aliasing and side effects in a higher-order expression language. Within this framework, dynamic data structures are handled by estimating each linked structure in an abstract store. Each call to a recursive function that creates a linked data structure is approximated by one entry in the abstract store. Although this method handles both recursive and higher-order functions, it also fails to give fine-grain interference information that is useful for parallelization.

Horwitz, Pfeiffer, and Reps [HPR89] have provided an extension of the method originally presented by Jones and Muchnick[JM81]. This method uses a refinement of *k-limited* graphs called *k-limited* stores. The major innovation is that the cells in *k-limited* stores are labeled by the program points that set their contents and this additional information is used to calculate dependence information. It should be noted that this method was not extended to handle procedures or functions, and it suffers from the same problem as the previous methods. That is, using one abstract location in the *k-limited store* to approximate many locations in real store leads to an overly conservative approximation.

The previous methods have been mostly of theoretical interest, and have not been targeted towards parallel programming languages or parallelizing compilers. The following methods are more directly applicable to the problems addressed in this thesis.

Lucassen and Gifford have proposed a language-based approach [Luc87, LG88]. They have defined a language (FX-87) that incorporates an effect system as well as a type system. The *effect* of a computation is a summary of the observable side-effects of the computation. For example, {read(X), write(Y), alloc(Z)} represents the effect of a computation that reads from region X, writes to region Y, and allocates objects in region Z. A *region* describes the area of store in which side-effects might occur. Two computations interfere when they have interfering effects. Although the effect system differentiates between totally disjoint linked structures, it does not provide a way of distinguishing between different parts of a large data structure. For example, even though the left and right sub-trees of a binary tree do not share any storage, the effect system forces both sub-trees to be associated with the same region. This lack of fine-grain information based on the recursive structure of the object results in an overly conservative interference analysis.

Another language-based approach, *Refined C*, has been suggested by Klappholz et al. [DK85, KKK89]. Refined languages extend conventional languages with special data partitioning constructs. Thus, rather than rely solely on compile-time analysis for interference detection, run-time code is associated with the partitioning constructs, and interference results in run-time errors. The extension of *Refined C* to linked data-structures is recent, and it has yet to be determined if the partitioning constructs will be useful for programming, and if the run-time checks can be done efficiently.

In the area of parallelizing compilers, interest is increasing in providing tools for parallelizing Lisp and C. In both cases accurate analysis of dynamic data structures is critical.

A technique for parallel restructuring of C programs has been described by Guarna [Gua88]. Although this technique includes methods for pointer structures, it does not handle procedures and functions. In addition, the technique *assumes* that the structures are trees. If the structure specified by a program is not a tree, then the interference analysis will not be safe.

Harrison has developed a parallelizing compiler for Lisp [Har86, HP88]. This approach uses a novel representation for S-expressions to facilitate the parallel creation and access of lists. However, because of the added complexity of analysing interference with pointer updates, and restrictions due to the representation of S-expressions, this work was limited to a subset of Lisp which prohibited pointer updates. More recently, Harrison has developed a more general interprocedural analysis method for Scheme programs [Har89]. To solve the update problem, Harrison suggests alternate definitions for *cons*, *car*, *cdr*, *set-car*, and *set-cdr* which simulate pointer updates with closures that do not contain any pointer updates [Har89]. Although examples of parallelizing applicative programs are given, the utility of the closure approach for updates has not yet been demonstrated.

Another approach, also targeted to restructuring Lisp programs for concurrent execution, has been suggested by Larus and Hilfinger [LH88a, LH88b]. Their method is designed to handle objects composed of structures. A *structure* is defined as a memory-resident object composed of a collection of named fields where each field may contain either a pointer to a structure or a non-pointer value. The analysis uses *alias graphs* to estimate the relation between variables, structures and pointers. In order to handle general purpose structures, the alias graphs are complex, and thus operations on alias graphs are potentially expensive. When these methods are implemented and integrated into their Curare program restructuring system, further information about their accuracy and practicality will be available.

Other analysis techniques that deal with dynamic data structures, but not specifically with interference analysis and parallelization, include: a semantic model of reference counting for optimizing applicative programs [Hud86], and lifetime analysis for dynamically allocated objects [RM88].

1.2 Our Approach

We focused on developing efficient and implementable methods for recursive data structures that exhibit regular properties (trees and DAGs). By restricting our method to these regular and widely used data structures, we were able to design a data abstraction that leads to fine-grain interference analysis. By abstracting on properties of the data structure, rather than accounting for every cell (node) in the data structure, we avoided the trap of using one abstract cell to estimate many real storage locations. Thus, we exploited the regularities of the data structure in order to obtain an efficient solution which yields more useful and accurate results than would be possible otherwise.

Our overall strategy was to:

1. design an imperative language that included recursively defined dynamic data structures, and recursive procedures and functions;
2. develop a data abstraction and associated interference analysis tools for programs that use dynamic data structures in a regular manner;
3. implement the analysis tools in a prototype system;
4. give a formal description of the analysis and prove soundness of the method;
5. develop parallelization techniques based on the interference analysis;
6. implement an experimental back-end for a shared memory machine; and
7. test the method on a set of representative programs.

In developing the interference analysis methods, we placed particular emphasis on reducing the run-time complexity by exploiting the regularities of the data structure and on choosing an abstract representation of the data structure which provides useful information for parallelization.

Note that our techniques are defined for an imperative language that supports recursively defined pointer structures, dynamic allocation, pointer updates, and recursive procedures and functions. Since the natural programming paradigm for recursive data structures requires recursive procedures and functions, it is critical to include a mechanism for handling recursion in the interference analysis. By implementing our methods we demonstrated that the techniques are practical enough to integrate into parallelizing compilers. Furthermore, our implementation was used to illustrate the effectiveness of the methods on non-trivial examples. As well as providing an implementation, we also described our methods within a formal framework that was used to prove soundness of the method. Thus, our overall goal was to demonstrate an effective, safe, and implementable method that is general enough to be useful in parallelizing compilers.

1.3 Organization of the Thesis

Chapters 2 through 5 provide a complete description of the interference analysis and parallelization methods. Chapter 2 specifies the imperative language for which our analysis is defined and gives a description of the sorts of recursive data structures we consider. Chapter 3 gives a detailed development of the interference analysis tools along with many illustrative examples. In chapter 4 we give a formal model of the interference analysis and abstract domain. In the final section of chapter 4 we use the formal model to reason about the fixed-point approximation and to prove soundness of the method. Chapter 4 does not introduce any new analysis methods and may be skipped by those not interested in the details. Chapter 5 presents three parallelization methods built on the interference analysis tools presented in chapter 3.

In chapter 6 we use the analyses developed in the previous chapters to develop a scheme for exploiting coarse-grain parallelism on the BBN Butterfly. We illustrate our methods by parallelizing a sequential SIL program for adaptive bitonic sort. We also present concrete speedup results for our example and give experimental data illustrating various forms of overhead. Finally, in chapter 7 we put forward some conclusions and give suggestions for areas of future work.

Chapter 2

Setting

In this chapter we outline the setting in which the interference analysis tools and parallelization techniques are developed. In the first section a simple imperative language (SIL) is defined, and in the second section we describe the data structures for which our analysis is defined.

2.1 SIL

To focus our attention on the features relevant to analysis of recursive data structures, we have defined a simple imperative language that includes basic control flow statements, recursively defined pointer structures, dynamic allocation, and recursive procedures and functions.

A SIL program consists of a type definition for specifying a recursive data structure, a parameterless procedure *main* and a set of auxiliary procedures and functions. The auxiliary procedures and functions may be recursive. The language is statically scoped and has call-by-value semantics. An outline of the abstract syntax for SIL is given in figure 2.1, and the standard semantics is specified in appendix A.

Two data types are supported, a scalar *integer* type, and a recursive *handle* type. The *handle* type is specified by a **nodedef** definition given at the beginning of each SIL program. Figure 2.2 gives examples of **nodedef** definitions for linked lists, binary trees, and 2-3-trees, along with the recursive *handle* type that each definition denotes.

Operationally one can think of handles as pointers to nodes in a heap. Thus, with call-by-value semantics, a procedure call passes integer and heap pointer values to the called procedure. We provide a built-in function *new* that allocates new nodes in the heap. The return value from an invocation of *new* must be assigned to a variable with type *handle.*

The SIL statements of particular interest for interference analysis are those that access or modify the data structures through the use of handle variables. Given

```
<Program> ::= program <ProgId>
                  [ constant <ConsantList> end;]
                  nodedef <FieldList> end;
                  <ProcedureFunctionList>
                  <MainProcedure>

<Procedure> ::= procedure <ProcId> ( <ParamList> )
                  <LocalList>
                  <Block>

<Function> ::= function <FuncId> (<ParamList>) <ReturnTypeList>
                  <LocalList>
                  <Block> => return ( <ReturnIdList> )

<Block> ::= begin <StmtList> end

<Arg> ::= <IntegerExpr> | <HandleName>

<Stmt> ::= <ScalarAssignment>
         | <HandleAssignment>
         | <HandleUpdate>
         | if <Expr> then <Stmt> [else <Stmt>]
         | while <Expr> do <Stmt>
         | repeat <Stmt> until <Expr>
         | <Block>
         | <ProcedureName> ( <ArgList> )
         | <IdList> := <FunctionName> ( <ArgList> )
```

Figure 2.1: Abstract Syntax of SIL.

```
nodedef                              type handle =
  value: int;                          nil |
  link: handle                         (value: int, link: handle)
end;

nodedef                              type handle =
  key: int;                            nil |
  left, right: handle                  (key: int; left, right: handle)
end;

nodedef                              type handle =
  lowofsecond, lowofthird: int;        nil |
  first, second, third: handle         (lowofsecond, lowofthird: int;
end;                                      first, second, third: handle)
```

Figure 2.2: Examples of recursive handle types.

that a and b are handle variables, h is a handle field, s is a scalar field, and x is an integer variable, the *basic handle statements* are of the form:

$a :=$ **nil**;
$a := new()$;
$a := b$;
$a := b.h$;
$a.h := b$;
$x := a.s$;
and $a.s := x$.

By focusing on this small set of basic handle statements, we reduce the number of analysis rules required without losing any expressiveness in the language. More complex handle statements such as $a.left.right := b.right$ are easily translated into a sequence of basic handle statements ($t1 := a.left$; $t2 := b.right$; $t1.right := t2$).

There are two further simplifications worth noting. The first is that expressions in SIL are *pure*, they cannot have side-effects. The second simplification is that handle arguments to procedures and functions must be handle identifiers. Thus, statements of the form $f(a.left)$ must be translated into a statement sequence of the form $t1 := a.left$; $f(t1)$.

SIL should be considered a subset of a more complete programming language, or as a high-level intermediate representation for imperative programs. For example, there is a direct mapping from SIL to a subset of C and conversely it is quite straightforward to write a translator that converts programs written in a subset of C to SIL. The translation from SIL to C is outlined in appendix B and an example of the translation is found in appendix D. Currently, we are using SIL as a test-bed for experiments with various analysis and parallelization tools.

2.2 Trees and DAGs

The basic building blocks of our data structures are nodes. Each *node* consists of one or more fields and each *field* is designated as a *scalar* field or a *handle* field. A handle field contains either **nil** or a handle (pointer) to a node of the same type. In general, objects built by linking nodes together are *directed graphs*. We classify two special types of directed graphs: (1) a *TREE* is a directed graph in which each node has at most one parent, and (2) a *DAG* is a directed graph in which some node has more than one parent and the graph does not contain a directed cycle.

The potential for parallelism in programs that use trees arises from the following observation. If a program builds linked structures that are of type *TREE*, then unrelated sub-trees, T_i and T_j, of tree T are guaranteed to share no common storage. Thus, a computation on T_i or any sub-tree of T_i will not interfere with a computation on T_j or any sub-tree of T_j. In addition, for both *TREE* and *DAG* data structures, we can make the following observation: if node a is above node b, then node a can never be accessed starting at *b*.

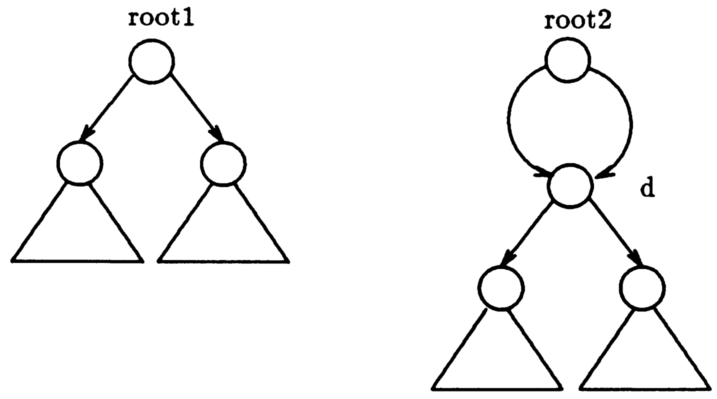


Figure 2.3: A *TREE* and a *DAG*.

As an introduction to the interference analysis presented in the next chapter, consider the *TREE* and *DAG* data structures in figure 2.3. A coarse-grain analysis may indicate that all the nodes in the structure *root1* are distinct from the nodes in *root2*. However, a more accurate analysis could determine that that *root1* is a *TREE* data structure and therefore any computation on *root1.left* or a sub-tree of *root1.left* is guaranteed not to interfere with a computation on *root1.right* or any sub-tree of *root1.right*. The analysis should also detect that *root2* is a *DAG*. In this case, *root2.left* and *root2.right* point to the *same* sub-tree, and computations on *root2.left* could interfere with computations on *root2.right*. However, if *d* is the only *DAG* node in the data-structure, computations on *d.left* will not interfere with computations on *d.right*.

Chapter 3

Interference Analysis Tools

In this chapter a collection of structure-based, dataflow analysis tools are developed. The goals of the analysis are: (1) to check that the data structures built be the program are the special cases of *TREES* or *DAGS*, and (2) to determine when two handles refer to the roots of unrelated sub-pieces of the data structure. Given an input program, the analysis computes a set of possible relationships among handles live at each point in the program. A *point* refers to a position between two statements in the program. A handle h is *live* at a point x if there is some execution path starting at x that uses h. The structure of the analysis is illustrated in figure 3.1. Given p, an estimate of the relationships among all handles live at point x, we wish to compute p', an estimate of the relationships among all handles live at point y.

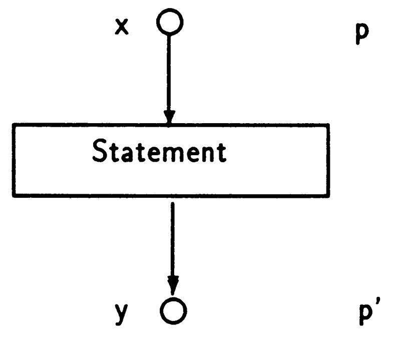


Figure 3.1: Structure of the analysis.

The estimate of relationships among handles captures the relative position of handles within a tree or forest. Relative information can be used to detect if a statement creates a data structure that is possibly not a *TREE* or a *DAG*. For example, if node a is a descendant of node b, then the statement *a.left* := b will create a cycle and the structure can no longer be considered a *TREE* or a *DAG*. Similarly, relative information can be used to detect when a statement of the form *a.left* := b creates a *DAG* structure in which b has more than one parent. Relative information may also be used to determine if two handles refer to disjoint sub-trees in a structure of type *TREE*. If node a is not a descendant of node b and node b is not a descendant of node a, then a and b refer to disjoint sub-trees and a computation on a cannot interfere with a computation on b.

The structure of this chapter is as follows. In section 3.1, we define the data structure abstraction used to estimate relationships among handles. Using this abstraction, we present the analysis techniques for simple statements and *TREE* data structures in section 3.2. Section 3.3 outlines an improved approximation that incorporates a simple analysis for estimating whether a handle is a **nil** pointer, or a pointer to a node. Extensions to the analysis for *DAG* data structures is given in section 3.4. In section 3.5 we use the simple statement analysis rules as building blocks for the development of analysis for compound statements, and in section 3.6 we present the analysis rules for procedure and function calls.

3.1 Data Structure Abstraction

In choosing an approach to interference analysis, the choice of data structure abstraction is critical to both the accuracy and efficiency of the analysis methods. The data structure abstraction must capture information that is accurate enough to detect unrelated sub-pieces of a data structure, while also providing a representation that allows for reasonably efficient analyses. If the data structure abstraction is too coarse, or abstracts the wrong characteristic, then the results of the analysis techniques will be overly conservative and not useful for parallelization. On the other hand, if the abstraction is too fine, or abstracts more characteristics than are required, the resulting analyses will be inefficient and not practical for use with parallelizing compilers.

3.1.1 Path Matrices and Path Expressions

In choosing our abstraction, *path matrices*, we concentrated on designing an abstraction that exploits the regularities found in tree-like data structures. Rather than estimating the relationships between *all* nodes in a data structure, path matrices are designed to capture the important relationships among the nodes to which the program has access (the live handles at a point in the program). Thus, the *paths* between live nodes is the characteristic we abstract. We represent this abstraction with a *path matrix* that encodes a *path estimate* for each pair of live handles. For each pair of live handles, h_i and h_j, the *relationships* between h_i and h_j are approximated by a list of *path expressions* stored in path matrix p, at location $p[h_i, h_j]$.

Definition 1. A *path expression* has one of three possible forms:

1. S, denoting the set containing the empty path (the nodes are the same),
2. *link_expression*, denoting a set non-empty paths, or
3. S + *link_expression*, denoting the set containing both the empty path and the set of non-empty paths given by the *link_expression.*

The *links* of a *link expression* are determined by the type of the data structure node. The following definition gives the possible links for binary trees. Links for other types of nodes are similarly defined.

Definition 2. A *link* for binary trees is one of:

L^i: exactly i left edges,

L^+: one or more left edges,

R^i: exactly i right edges,

R^+: one or more right edges,

D^i: exactly i down (left or right) edges, or

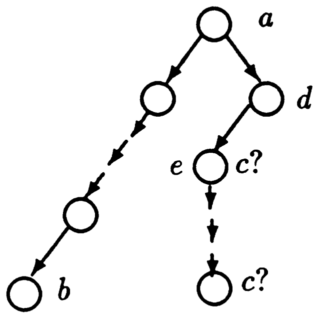
D^+: one or more down edges.

Note that path expressions are a restricted form of regular expressions. Although we can write some regular expressions as path expressions, ($RLRR+RLLR$ is equal to the path expression $R^1L^1D^1R^1$), we cannot express all regular expressions as path expressions. For example, we have no path expression for the regular expression $LR + LRR + LRRR$. However, we can approximate this regular expression by L^1R^+. The restrictions on the path expressions are chosen to represent abstracted information about paths between handles, while providing efficient operations such as equality testing and merging.

Each path expression is classified as *definite* - a path between two nodes is guaranteed to exist, or *possible* - a path between the nodes may or may not exist.

Figure 3.2 gives an example of a path matrix that represents the possible relationships among the live handles a, b, c, d, and e. For example, p[a,e] = $\{R^1L^1\}$ specifies that there is a definite, directed path of R^1L^1 from the node pointed to by a to the node pointed to by e. Note that the entry p[e,a] = $\{\}$ specifies that there is no path from e to a. Two handles h_i and h_j are unrelated (refer to disjoint sub-trees) if p[h_i,h_j] = p[h_j,h_i] = $\{\}$. In our example, b and c are unrelated and b and e are unrelated. The path matrix entry, p[c,e] = $\{S?\}$, is an example of a possible path. Handle c may or may not refer to the same node as handle e.

Two sorts of approximation are encoded in path matrices: path *length* approximation, and path *direction* approximation. The entry p[a,b] = $\{L^1L^+L^1\}$ is an example of a path with an exact direction (left), but an approximate length (3 or more). The entry p[a,c] = $\{R^1D^+\}$ is an example of a path that has an approximate direction (D may be left or right) and an approximate length (2 or more). Note that since a path matrix may contain path expressions with approximate length and/or approximate direction, a path matrix specifies a set of possible data structures.



	a	b	c	d	e
a	S	$L^1L^+L^1$	R^1D^+	R^1	R^1L^1
b		S			
c			S		$S?$
d			D^+	S	L^1
e			$S+D^+$		S

Figure 3.2: An example path matrix.

3.1.2 Operations on Path Expressions

In developing the analysis tools, three basic operations on path expressions are required: (1) equality, (2) merging, and (3) concatenation.

Definition 3. A path expression p has a *normal form* that consists of a concatenation of subsequences of the form L^i, L^+, L^iL^+, R^i, R^+, R^iR^+, D^i, D^+, D^iD^+, such that no two subsequences of the same kind (L,R,or D) are adjacent, and no subsequence containing D^+ is adjacent to a subsequence containing L^+ or R^+.

The normal form of a path expression p is computed by two traversals of p. The first traversal performs ***first-order normalization***, while the second traversal performs *second-order normalization.* First-order normalization reduces a link subsequence having the same kind (L, R, or D for binary trees) to a subsequence of the form $\mathcal{X}^i$, $\mathcal{X}^+$, or $\mathcal{X}^i\mathcal{X}^+$. First-order normalization is based on the observation that $\mathcal{X}^i\mathcal{X}^j$ denotes the same set of paths at $\mathcal{X}^{(i+j)}$, and that $\mathcal{X}^+\mathcal{X}^+$ denotes the same set as $\mathcal{X}^1\mathcal{X}^+$. Given a subsequence S containing links of the same kind, the reduced subsequence S_{red} is computed as follows. S is divided into $S_{link} = \mathcal{X}_1^{l_1}\ldots\mathcal{X}_m^{l_m}$ and $S_{plus} = \mathcal{X}_1^+\ldots\mathcal{X}_n^+$, where S_{link} collects all links of the form $\mathcal{X}^i$ and S_{plus} collects all links of the form $\mathcal{X}^+$. Given that n_{plus} is the number of links in $\mathcal{X}_{plus}$ and n_{link} is the sum of the l_i's in S_{link}, the reduced subsequence S_{red} is given by:

$$S_{red} = \begin{cases} \mathcal{X}^{n_{link}} & (n_{link} > 0, n_{plus} = 0) \\ \mathcal{X}^{(n_{plus}-1)}\mathcal{X}^{+} & (n_{link} = 0, n_{plus} > 0) \\ \mathcal{X}^{(n_{plus}+n_{link}-1)}\mathcal{X}^{+} & (n_{link} > 0, n_{plus} > 0). \end{cases}$$

Given a first-order normalized path expression, second-order normalization reduces adjacent subsequences when one subsequence contains L^+ or R^+, and the other subsequence contains D^+. These reductions are based on the fact that subsequences of the form $\mathcal{X}^+D^+$ ($\mathcal{X}$ = L or R) denote the same set of paths as $\mathcal{X}^1D^+$, and similarly $D^+\mathcal{X}^+$ denotes the same set as $D^+\mathcal{X}^1$. The reduction rules for adjacent subsequences are as follows:

$$\begin{aligned} \mathcal{X}^+D^+ &\Rightarrow \mathcal{X}^1D^+ \\ \mathcal{X}^i\mathcal{X}^+D^+ &\Rightarrow \mathcal{X}^{(i+1)}D^+ \\ \mathcal{X}^+D^iD^+ &\Rightarrow \mathcal{X}^1D^iD^+ \\ \mathcal{X}^i\mathcal{X}^+D^jD^+ &\Rightarrow \mathcal{X}^{(i+1)}D^jD^+ \\ D^+\mathcal{X}^+ &\Rightarrow D^+\mathcal{X}^1 \\ D^+\mathcal{X}^i\mathcal{X}^+ &\Rightarrow D^+\mathcal{X}^{(i+1)} \end{aligned}$$

As an example of normalization, the normal form reduction of $D^+L^1L^+L^3R^+R^+$ proceeds by a first-order normalization giving $D^+L^4L^+R^1R^+$ and then a second-order normalization giving $D^+L^5R^1R^+$. Similarly, $D^+L^+L^+L^3R^1R^+$ reduces first to $D^+L^4L^+R^1R^+$ and then to $D^+L^5R^1R^+$.

Note, that since normalization can be accomplished by two traversals of a path expression p, the complexity of the normal form computation is $O(n)$, where n is the number of links in p.

Definition 4. Two path expressions p_1 and p_2 are *equal* if they are of the same form (S, e, or $S + e$), and the normal forms of their link expressions are identical (they have exactly the same sequence of links).

The complexity of equality testing of two normalized link expressions e_1 and e_2 is $O(min(n, m))$, where n is the number of links in e_1 and m is the number of links in e_2.

Definition 5. Link expression e_1 is *more general* than link expression e_2 if all paths in the set denoted by e_2 are also in the set denoted by e_1 and there exists some path in the set denoted by e_1 that is not in the set denoted by e_2.

Definition 6. The *minimal length* of a link expression is the sum of the minimal lengths of its links, where the minimal length of D^+, R^+, or L^+ is 1 and the minimal length of D^i, R^i, or L^i is i.

Definition 7. The *squash* of two normalized link expressions e_1 and e_2 is a normalized link expression x such that: (1) any path in the sets denoted by e_1 or e_2 is also in the set denoted by x, and (2) the minimal length of x is no greater than $min(n, m)$, where n is the minimal length of e_1 and m is the minimal length of e_2.

There are many choices for valid squash functions. A coarse-grain, inexpensive squash is of the form: $squash(e_1, e_2) = D^+$. Certainly this squash function obeys the properties given in definition 7. However, such coarse-grain approximation leads to a very conservative analysis, and we usually require a more informative approximation. The strategy outlined below has complexity $O(max(n, m))$, where n is the minimal length of e_1 and m is the minimal length of e_2. Given two link expressions e_1 and e_2, $x = squash(e_1, e_2)$ can be computed by building x stage-wise from the front and the back. At each stage, the match of the first subsequences of e_1 and e_2 and the match of the last subsequences of e_1 and e_2 is estimated and the subsequence with the best match is squashed using simple rules defined on small subsequences (for example, $squash(L^iL^+, L^jL^+) \Rightarrow L^{min(i,j)}L^+$ and $squash(D^iD^+, L^jL^+) \Rightarrow squash(D^{min(i,j)}D^+)$). Each stage is a constant time operation, and there are at most $O(max(n, m))$ stages. By varying the matching estimate and the simple squash functions, a variety of efficient squash functions have been defined. Some examples of valid squashes are: $squash(R^+, R^1) = R^+$, $squash(L^2R^1, L^1R^2) = L^1D^1R^1$, and $squash(L^1L^+, L^1L^+L^1) = L^1L^+$.

We can use the definition of *squash* to define both *merge*, a conservative approximation of *set union*, and *concatenation* of path expressions. The *equal*, *merge*, and *concatenation* operations are defined in figure 3.3.

As mentioned previously, there are two kinds of path expressions: *possible* and *definite*. If we represent a definite path expression p with the pair $\langle p, true\rangle$, and a possible path expression $p?$ with the pair $\langle p, false\rangle$, then we can define the following extensions to the path expression operations.

Definition 8. *Extended* operations for *equality* ($=$), *and-merging* ($\bowtie^\wedge$), *or-merging* ($\bowtie^\vee$), and *concatenation* ($\cdot$) of two path expressions $\langle p_1, b_1\rangle$, and $\langle p_2, b_2\rangle$ are defined as follows:

Path expressions $\langle p_1, b_1\rangle$ and $\langle p_2, b_2\rangle$ are *equal* $\Leftrightarrow$ $p_1 = p_2$ and $b_1 = b_2$.

The *and-merge* of path expressions $\langle p_1, b_1\rangle$ and $\langle p_2, b_2\rangle$ is defined as:

$$\langle p_1, b_1\rangle \bowtie^\wedge \langle p_2, b_2\rangle = \langle p_1 \bowtie p_2, b_1 \wedge b_2\rangle.$$

The *or-merge* of path expressions $\langle p_1, b_1\rangle$ and $\langle p_2, b_2\rangle$ is defined as:

$$\langle p_1, b_1\rangle \bowtie^\vee \langle p_2, b_2\rangle = \langle p_1 \bowtie p_2, b_1 \vee b_2\rangle.$$

The *concatenation* of path expressions $\langle p_1, b_1\rangle$ and $\langle p_2, b_2\rangle$ is defined as:

$$\langle p_1, b_1\rangle \cdot \langle p_2, b_2\rangle = \langle p_1 \cdot p_2, b_1 \wedge b_2\rangle.$$

Note that the path expression resulting from the concatenation of p_1 and p_2 is definite only when both p_1 and p_2 are definite.

Path Expression Equality ($p_1 = p_2$)

$p_2 \setminus p_1$	S	e_1	$S + e_1$
S	true	false	false
e_2	false	$normal(e_1) = normal(e_2)$	false
$S + e_2$	false	false	$normal(e_1) = normal(e_2)$

Path Expression Merging ($p_1 \bowtie p_2$)

$p_2 \setminus p_1$	S	e_1	$S + e_1$
S	S	$S + e_1$	$S + e_1$
e_2	$S + e_2$	$squash(e_1, e_2)$	$S + squash(e_1, e_2)$
$S + e_2$	$S + e_2$	$S + squash(e_1, e_2)$	$S + squash(e_1, e_2)$

Path Expression Concatenation ($p_1 \cdot p_2$)

$p_2 \setminus p_1$	S	e_1	$S + e_1$
S	S	e_1	$S + e_1$
e_2	e_2	$e_1 e_2$	$e_2 \bowtie (e_1 e_2)$
$S + e_2$	$S + e_2$	$e_1 \bowtie (e_1 e_2)$	$S + (e_1 \bowtie e_2 \bowtie (e_1 e_2))$

Figure 3.3: Path expression equality, merging, and concatenation.

3.2 Analysing Simple Statements (*TREES*)

Given the description of the data structure abstraction and the basic operations of path expressions, the overall structure of the analysis can now be more precisely stated. For each kind of statement, an analysis function is defined that takes as input an instance of a statement s, and a path matrix p that is a safe estimate of the relationships among all non-nil handles before execution of statement s, and produces as output a new path matrix p' that is a safe estimate of the relationships after execution of the statement s. Note that each live handle may either be **nil**, or a pointer to a node. Since there is never any path from a **nil** handle to any other handle, the path matrix analysis only computes the path expressions for the non-nil case.

Simple statements that result in changes in path matrices can be classified as: (1) statements that only add new relationships to a path matrix (*handle assignment statements*), and (2) statements that change the relationships in a path matrix (*handle update statements*).

3.2.1 Handle Assignment Statements

There are four forms of handle assignment statements: $A :=$ **nil**, $A := new()$, $A := B$, and $A := B.f$. The first three forms are quite straightforward and their rules are given in figure 3.4. In the case of $A :=$ **nil** and $A := new()$, the rules simply state that the output path matrix p' is the same as the input path matrix p, except for the column entries $p'[h_i, A]$, and the row entries $p'[A, h_j]$. These entries of p' are defined such that $p'[A, A] = S$, and all other entries $p'[h_i, A]$ and $p'[A, h_j]$ are empty (there are no paths between A and any other handle).

The rule for $A := B$ is only slightly more complex. As in the previous rules, the only differences between p and p' occur in the entries $p'[h_i, A]$ and $p'[A, h_j]$. Since A and B refer to the same node, the entries $p'[A, B]$ and $p'[B, A]$ are both S. The remaining entries for A are just the corresponding entries for B: $p'[h_i, A] \mapsto p[h_i, B]$ and $p'[A, h_j] \mapsto p[B, h_j]$.

An example of the application of the simple handle assignment rules is given in figure 3.5. Given the input path matrix of figure 3.5(a), the path matrices given in 3.5(b,c, and d) illustrate the results of analysing the sequence of statements: $c :=$ **nil**, $d := new()$, and $e := a$.

Statements of the form $A := B.f$ require a more complex analysis. As an example, consider the rule given in figure 3.6 for $A := B.left$. As in the previous handle assignment rules, p' differs from p in the column that defines the paths between the handles h_i and A, and in the row that defines the paths between A and h_j. The basic cases specify that $p'[A, A] = \{S\}$, $p'[B, A] = \{L^1\}$, and $p'[A, B] = \{\}$. For the remaining entries, we construct the relationships $p'[h_i, A]$ and $p'[A, h_j]$ from the corresponding entries $p[h_i, B]$ and $p[B, h_j]$.

Figure 3.7 illustrates the mapping of path expressions from $p[h_i, B]$ and $p[B, h_j]$

```
$A :=$ nil
    $\forall\ h_i,\ h_j \in H'$ .
        $p'[h_i,\ h_j] \mapsto$
            if $h_i = A$ and $h_j = A$ then
                $\{S\}$
            else if $h_i = A$ or $h_j = A$ then
                $\{\}$
            else
                $p[h_i,\ h_j]$
```

```
$A := new()$
    $\forall\ h_i,\ h_j \in H'$ .
        $p'[h_i,\ h_j] \mapsto$
            if $h_i = A$ and $h_j = A$ then
                $\{S\}$
            else if $h_i = A$ or $h_j = A$ then
                $\{\}$
            else
                $p[h_i,\ h_j]$
```

```
$A := B$
    $\forall\ h_i,\ h_j \in H'$ .
        $p'[h_i,\ h_j] \mapsto$
            if $h_i = A$ then
                if $h_j = B$ or $h_j = A$ then $\{S\}$ else $p[B,h_j]$
            else if $h_j = A$ then
                if $h_i = B$ then $\{S\}$ else $p[h_i,B]$
            else
                $p[h_i,\ h_j]$
```

Figure 3.4: Rules for $A :=$ **nil**, $A := new()$, and $A := B$.

(a) Initial Path Matrix

	a	b
a	S	L^+
b		S

(b) After statement : $c :=$ **nil**

	a	b	c
a	S	L^+	
b		S	
c			S

(c) After statement : $d := new()$

	a	b	c	d
a	S	L^+		
b		S		
c			S	
d				S

(d) After statement : $e := a$

	a	b	c	d	e
a	S	L^+			S
b		S			
c			S		
d				S	
e	S	L^+			S

Figure 3.5: Example application of the simple handle assignment rules.

```
$A := B.left$
    $\forall\ h_i,\ h_j \in H'$ .
        $p'[h_i,\ h_j] \mapsto$
            if $h_i = A$ then
              if $h_j = A$ then
                $\{S\}$
              else if $h_j = B$ then
                $\{\}$
              else
                map below_rule $p[B,h_j]$
            else if $h_j = A$ then
              if $h_i = B$ then
                $\{L^1\}$
              else
                map above_rule $p[h_i,B]$
            else
              $p[h_i,\ h_j]$
```

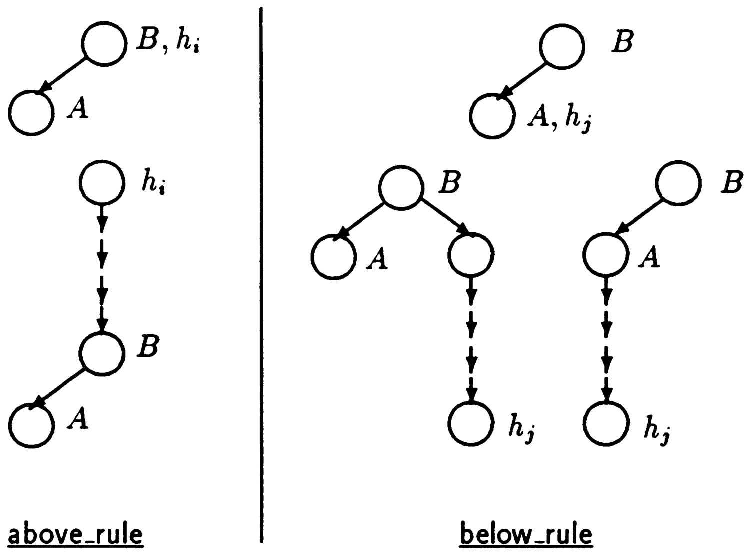
Figure 3.6: Rule for $A := B.left$.

to the appropriate path expressions for $p'[h_i, A]$ and $p'[A, h_j]$. In these functions the pair $\langle$ e, d $\rangle$ represents an input path expression where d is true if the path expression is definite and false if the path expression is possible.

The function above_rule$\langle$e, d$\rangle$ is used to map path expressions from $p[h_i, B]$ to path expressions for $p'[h_i, A]$. If h_i points to the same node as B, then there is a path L^1 between h_i and A. Similarly, if there is a path denoted by the link expression x between h_i and B, then there is a path denoted by xL^1 between h_i and A.

The function below_rule$\langle$e, d$\rangle$ is used to map path expressions from $p[B, h_j]$ to path expressions for $p'[A, h_j]$. If the path between B and h_j begins with L, there is a shorter path between A and h_j. If the path begins with R, then there will be no path between A and h_j. If the path between B and h_j begins with a D link, then it is uncertain whether or not there is a path between A and h_j. Thus, a *possible* shorter path is included. Note that when the first link of the path expression e is D^+ or L^+, there are two possible shorter paths that must be merged.

An example application of the handle assignment rules is given in Figure 3.8. Figure 3.8(a) illustrates an initial path matrix representing the relationships between the handles a, b, and c. Figure 3.8(b) shows the path matrix that would result from the statement $d := a.right$ and 3.8(c) illustrates the resulting path matrix after the additional statement $e := d.left$. Note that although the path matrices in 3.8(a) and 3.8(b) have only *definite* paths, the path matrix in 3.8(c) contains some *possible* paths (denoted by ?). Since both the length and direction



```
above_rule⟨e, d⟩
( $\langle S, d\rangle$        $\Rightarrow$ $\{\langle L^1, d\rangle\}$
  $\langle x, d\rangle$        $\Rightarrow$ $\{\langle xL^1, d\rangle\}$
  $\langle S + x, d\rangle$    $\Rightarrow$ $\{(above\_rule\langle S, d\rangle) \bowtie^{\wedge} (above\_rule\langle x, d\rangle)\}$
)
below_rule⟨e, d⟩
( $\langle S, d\rangle$        $\Rightarrow$ $\{\}$
  $\langle x, d\rangle$        $\Rightarrow$ $shorten\langle x, d\rangle$
  $\langle S + x, d\rangle$    $\Rightarrow$ $shorten\langle x, false\rangle$
)
shorten⟨e, d⟩
( $\langle L^1, d\rangle$      $\Rightarrow$ $\{\langle S, d\rangle\}$
  $\langle L^n, d\rangle$      $\Rightarrow$ $\{\langle L^{n-1}, d\rangle\}$
  $\langle L^+, d\rangle$      $\Rightarrow$ $\{\langle S + L^+, d\rangle\}$
  $\langle L^1x, d\rangle$     $\Rightarrow$ $\{\langle x, d\rangle\}$
  $\langle L^nx, d\rangle$     $\Rightarrow$ $\{\langle L^{n-1}x, d\rangle\}$
  $\langle L^+x, d\rangle$     $\Rightarrow$ $\{\langle x \bowtie L^+x, d\rangle\}$
  $\langle D^1, d\rangle$      $\Rightarrow$ $\{\langle S, false\rangle\}$
  $\langle D^n, d\rangle$      $\Rightarrow$ $\{\langle D^{n-1}, false\rangle\}$
  $\langle D^+, d\rangle$      $\Rightarrow$ $\{\langle S + D^+, false\rangle\}$
  $\langle D^1x, d\rangle$     $\Rightarrow$ $\{\langle x, false\rangle\}$
  $\langle D^nx, d\rangle$     $\Rightarrow$ $\{\langle D^{n-1}x, false\rangle\}$
  $\langle D^+x, d\rangle$     $\Rightarrow$ $\{\langle x \bowtie D^+x, false\rangle\}$
  -                            $\Rightarrow$ $[]$
)
```

Figure 3.7: Functions used in the analysis of *A := B.left*.

of the path between handles d and c is estimated ($p[d,c] = D^+$), the path between handles e and c is can be S, or $D+$, or no path at all. Thus, $p'[e,c]$ must include all the possibilities with $p'[e,c] = (S + D^+)?$.

3.2.2 Handle Update Statements

There are two forms of handle update statements: $A.f$:= **nil**, and $A.f := B$. Unlike the handle assignments statements, handle update statements can *change* the relationships between handles other than A and B.

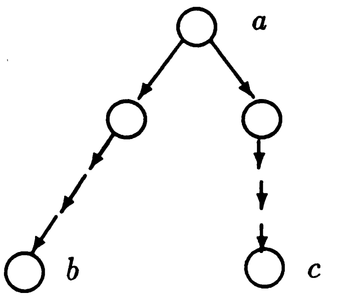
We first consider rules of the form $A.f$:= **nil** by examining the analysis rule for $A.left$:= **nil**. The effect of the statement $A.left$:= **nil** is to break the link between A and the node pointed to by $A.left$. This breaking operation not only changes that relationship between A and its left child, but also the relationships between all pairs of handles (h_i, h_j) such that h_i is above or equal to A, and h_j is below or equal to A's left child.

The analysis rule for $A.left$:= **nil** is given in figure 3.9. In this rule we use $path(A, path\_description, B, p)$ as a predicate to test for certain paths in the path matrix entry $p[A, B]$. For example, $path(h_i, *?, A, p)$ is true if there is *any* path expression (possible or definite) in $p[h_i, A]$, and $path(h_i, *, A, p)$ is true if there is any *definite* path expression in $p[h_i.A]$. Thus the first conditional test in figure 3.9 is true if there is definitely a path between h_i and A, and there is definitely a path starting with a left link between A and h_j. Similarly, the second conditional is true if there may be a path between handle h_i and A, and there may be a path beginning with a left link between A and h_j.

Given the description of the function *path*, we can now examine how the analysis rule estimates the effect of $A.left$:= **nil**. The entries $p'[h_i, h_j]$ are the same as those of $p[h_i, h_j]$ except for the entries where h_i may be above or equal to A and h_j may below or equal to the left child of A. If h_i is *definitely* above or equal to A, and h_j is *definitely* below or equal to the left child of A, then breaking the left link of A will break the path between h_i and h_j, and $p'[h_i, h_j]$ will be $\{\}$. If the relationships are not definite, then we must make an approximation of the effect of breaking the link. A simple approximation is given by the three clauses involving *remove_head*, *remove_tail*, and *make_possible*. If h_i is the same as A, then all paths between h_i and h_j that begin with L are broken, paths that begin with D are converted to possible paths, and paths that begin with R remain intact. Similarly, if h_j is the same as A's left child, then all paths between h_i and h_j that end with L are broken. For all other paths, the paths may or may not be broken, and we must include all paths in $p[h_i, h_j]$, as possible paths in $p'[h_i, h_j]$.

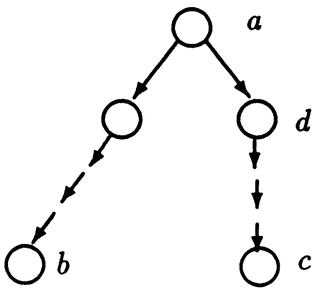
The last simple update rule is of the form $A.f := B$. As indicated in the rule for $A.left := B$ given in figure 3.10, the effect of $A.left := B$ is to produce a data structure with a cycle, a *DAG*, or a *TREE*. In figure 3.10 we outline the case for *TREE*s. In section 3.4 we give extensions to the analysis to handle *DAGS*. In the case of a cycle, the analysis terminates with a failure indicating that a cycle has been

(a) Initial path matrix



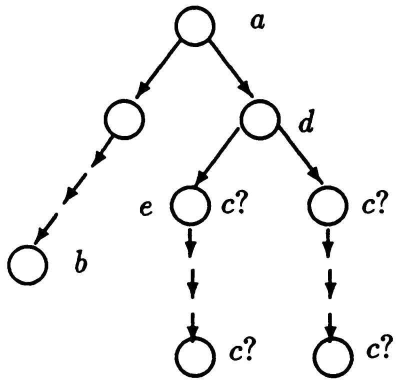
	a	b	c
a	S	L^2L^+	R^1D^+
b		S	
c			S

(b) After statement : $d := a.right$



	a	b	c	d
a	S	L^2L^+	R^1D^+	R^1
b		S		
c			S	
d			D^+	S

(c) After statement : $e := d.left$



	a	b	c	d	e
a	S	L^2L^+	R^1D^+	R^1	R^1L^1
b		S			
c			S		$S?$
d			D^+	S	L^1
e			$(S+D^+)?$		S

Figure 3.8: An example of handle assignments.

```
A.left := nil
    ∀ $h_i$, $h_j$ ∈ H' .
        $p'[h_i, h_j] \mapsto$
          if $path(h_i,*,A,p)$ and $path(A,L*,h_j,p)$ then
            {}
          else if $path(h_i,*?,A,p)$ and $path(A,L*?,h_j,p)$ then
            if $path(h_i,S,A,p)$ then
              $remove\_head(L,p[h_i,h_j])$
            else if $path(A,L,h_j,p)$ then
              $remove\_tail(L,p[h_i,h_j])$
            else
              $make\_possible(p[h_i,h_j])$
          else
            $p[h_i, h_j]$
```

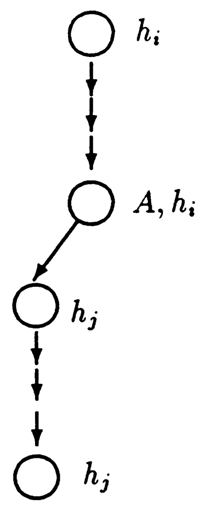


Figure 3.9: Analysis for *A.left* := **nil**.

detected. The analysis for *A.left* := *B* begins by producing a path matrix p' that is an estimate of the effect of *A.left* := **nil**. Path matrix p'' is produced by adding to p' the paths created by linking *A.left* to *B*. For all pairs of handles (h_i,h_j), such that h_i may be above or equal to *A*, and h_j may be below or equal to *B*, $p''[h_i, h_j]$ is the union of $p'[h_i, h_j]$ and all path expressions created by connecting the entries in $p'[h_i, A]$ and $p'[B, h_j]$ with an *L* link. A subtle aspect of this rule concerns whether or not *B* is a **nil** pointer, or a pointer to a node. If *B* is **nil**, then no connections are required and if *B* is definitely a pointer to a node, the connections are made by direct application of the concatenation ($\cdot$) operation. However, if the nilness of *B* is unknown, we must make a conservative approximation, and only consider the path expressions between *B* and h_j as *possible* paths. A simple extension to the analysis to estimate the nilness of handles is presented in the next section.

3.3 Estimating *nil* handles.

In the previous section we discussed the analysis rules for each of the basic handle statements. For statements of the form *A.f* := *B*, we found that a more accurate analysis can be specified if we know whether *B* is **nil** or a handle to a node. In this section we will outline a simple extension to the path matrix analysis for estimating **nil** handles.

```
$A.left := B$
    $p' \mapsto f(A.left{:=}nil, p)$;
    if $path(B, *?, A, p')$ then
      $possible\_cycle()$
    else if $handle\_above(B, p')$ then
      $possible\_dag()$
    else
    $\forall\ h_i,\ h_j \in H'$ .
        $p'' [h_i, h_j] \mapsto$
          if $path(h_i, *?, A, p')$
                and $path(B, *?, h_j, p')$ then
            if $is\_node(B)$ then
              $p'[h_i, h_j] \cup$
                ( $connect\_all$
                    ( $p'[h_i, A]$,
                      $\langle L^1, true\rangle$,
                      $p'[B, h_j]$
                    )
                )
            else
              $p'[h_i, h_j] \cup$
                ( $connect\_all$
                    ( $p'[h_i, A]$,
                      $\langle L^1, true\rangle$,
                      $make\_possible(p'[B, h_j])$
                    )
                )
          else
            $p'[h_i, h_j]$
```

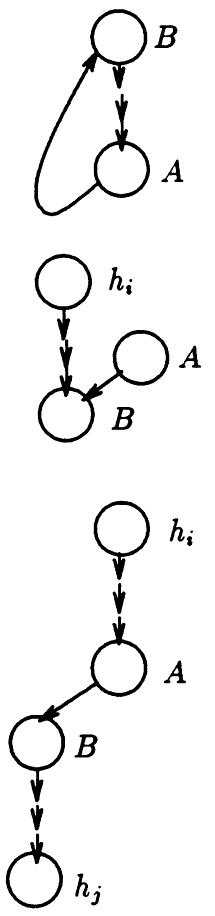


Figure 3.10: Analysis for $A.left := B$.

3.3.1 Data abstraction for *nil* handle estimates.

In order to perform the *nil* handle estimate, we augment our path matrix abstraction with an estimate for the nilness of each handle. The abstraction is based on a three-valued domain where $\circ$ represents **nil**, $\bullet$ represents a handle to a node, and $\odot$ represents the unknown case (the handle could be either **nil** or a handle to a node). The partial ordering on these items is given in figure 3.11.

$$\begin{tikzcd}
 & \odot \text{ (Either)} & \\
\bullet \text{ (Node)} \arrow[ur] & & \circ \text{ (Nil)} \arrow[ul]
\end{tikzcd}$$

Figure 3.11: Data abstraction for nil estimates.

For each handle we estimate the nilness of the handle itself, as well as estimating the nilness for each of its handle fields. Thus, for binary trees estimates are represented as pairs of the form *(handle estimate, field estimates)*. For example, the entry $p.IsNode[h_i] = (\bullet, \circ\odot)$ indicates that handle h_i is a pointer to a node, $h_i.left$ is **nil**, and $h_i.right$ is either **nil** or a pointer to a node. In the following rules we refer to the components of the estimate for handle *h_i* in path matrix p as: $p.IsNode[h_i].self$, $p.IsNode[h_i].left$, and $p.IsNode[h_i].right$.

Definition 9. Given the ordering of $\bullet$, $\circ$, and $\odot$ as specified in figure 3.11, the *nil_merge* of nil estimates $(x.self, x.leftx.right)$ and $(y.self, y.lefty.right)$ is $(x.self \sqcap y.self,\ (x.left \sqcap y.left)\ (y.right \sqcap y.right))$.

3.3.2 Analysis rules for *nil* handle estimates.

As in the previous path matrix analysis, we first define the analysis rules for the handle assignment statements, followed by the rules for the handle update statements. Figure 3.12 gives the rules for statements of the form $A :=$ **nil**, $A := new()$, and $A := B$. As expected for the simple statements, these rules are straightforward. For each statement, the nilness estimate for p' is the same as that for p, except for the entry $p'.IsNode[A]$. After execution of $A :=$ **nil**, A is definitely a **nil** pointer. Executing $A := new()$ results in a path matrix where, A is definitely a pointer to a node, while the left and right fields of A are definitely **nil**. After executing $A := B$ the pointer status of A in p' is just the pointer status of B in p.

Figure 3.13 shows the extended analysis for the example previously given in figure 3.5. The initial path matrix approximates the relationships between handles a and b. The nilness estimate for a $(\bullet, \bullet\bullet)$ indicates that a, $a.left$, and $a.right$ are all handles to nodes, while the estimate for b $(\bullet, \odot\odot)$ indicates that b is a handle to a node, but $b.left$ and $b.right$ may be handles to nodes or may be **nil** pointers. The three statements $c :=$ **nil**, $d := new()$, and $e := a$ illustrate the three analysis rules given in 3.12.

```
A := nil
    ∀ $h_i \in H'$ .
        $p'.IsNode[h_i] \mapsto$
          if $h_i = A$ then $(\circ, \circ\circ)$ else $p.IsNode[h_i]$;
B := new()
    ∀ $h_i \in H'$ .
        $p'.IsNode[h_i] \mapsto$
          if $h_i = A$ then $(\bullet, \circ\circ)$ else $p.IsNode[h_i]$;
A := B
    ∀ $h_i \in H'$ .
        $p'.IsNode[h_i] \mapsto$
          if $h_i = A$ then $p.IsNode[B]$ else $p.IsNode[h_i]$;
```

Figure 3.12: Nil estimate for $A :=$ **nil**, $A := new()$, and $A := B$.

The final handle assignment statement is of the form $A := B.f$. We outline the approach by presenting the rule $A := B.left$ in figure 3.14. For this rule $p'.IsNode[h_i]$ is the same as $p.IsNode[h_i]$ except for $h_i = A$, and $h_i = B$. For $p'.IsNode[A]$, the first component is given by $p.IsNode[B].left$, and the remaining field components are specified as unknown $(\odot)$. In the case of $p'.IsNode[b]$, we know that B must be a handle to a node for this to be a valid access. Thus, we refine the entry $p'.IsNode[B].self$ to indicate that B is a handle to a node $(\bullet)$.

The last two analysis rules specify how to estimate the effect of executing the handle update statements $A.f :=$ **nil** and $A.f := B$. Executing an update statement may change the nilness estimate for handle other than A. For example, if X is a handle that points to the same node as A, then executing the statement $X.left :=$ **nil** changes the nilness of both $A.left$ and $X.left$. In figure 3.15 we define the rules for estimating nilness for statements of the form $A.left :=$ **nil**. The estimate indicates that A must be a node $(\bullet)$, $A.left$ must be **nil** $(\circ)$, and $A.right$ remains unchanged. For all handles h_i that are definitely the same as A (*path(h_i,S,A,p) = true*), $h_i.left$ is **nil** $(\circ)$, and for all handles h_i that might be the same as A, $h_i.left$ is the merge of $\circ$ and the previous value $p.IsNode[h_i].left$. The rule for $A.left := B$ is essentially the same as for $A.left :=$ **nil**, except that the nilness of the left field of A and all handles that are the same as A is defined by the value $p.IsNode[B].self$.

(a) Initial Path Matrix

	$a(\bullet,\bullet\bullet)$	$b(\bullet,\odot\odot)$
a	S	L^+
b		S

(b) After statement : $c :=$ **nil**

	$a(\bullet,\bullet\bullet)$	$b(\bullet,\odot\odot)$	$c(\circ,\circ\circ)$
a	S	L^+	
b		S	
c			S

(c) After statement : $d := new()$

	$a(\bullet,\bullet\bullet)$	$b(\bullet,\odot\odot)$	$c(\circ,\circ\circ)$	$d(\bullet,\circ\circ)$
a	S	L^+		
b		S		
c			S	
d				S

(d) After statement : $e := a$

	$a(\bullet,\bullet\bullet)$	$b(\bullet,\odot\odot)$	$c(\circ,\circ\circ)$	$d(\bullet,\circ\circ)$	$e(\bullet,\bullet\bullet)$
a	S	L^+			S
b		S			
c			S		
d				S	
e	S	L^+			S

Figure 3.13: Example application of nil estimate.

```
$\underline{A := B.left}$
    $\forall\ h_i \in H'$ .
      $p'.IsNode[h_i] \mapsto$
          $\underline{\text{if}}\ h_i = A\ \underline{\text{then}}$
            $(p.IsNode[B].left,\ \odot\ \odot)$
          $\underline{\text{else}}\ \underline{\text{if}}\ h_i = B\ \underline{\text{then}}$
            $(\bullet,\ (p.IsNode[B].left)\ (p.IsNode[B].right))$
          $\underline{\text{else}}$
            $p.IsNode[h_i]$;
```

Figure 3.14: Nil estimate for $A := B.left$.

3.4 Analysing Simple Statements (*DAGS*)

The previous section provided an overview of an interference analysis method for *TREES* and an extension of the analysis to estimate nil handles. In calculating the relative positions of live handles at each point in the program, the analysis relies on the fact that the data structure is a *TREE*; thus if a possible *DAG* is detected, the analysis must fail. In this section an extension of the *TREE* analysis to handle *DAGS* is presented.

Simple imperative programs that process *TREES* may also create *DAGS* temporarily. For example, consider the program in figure 3.16. This program operates on nodes with three handle fields, *left*, *middle*, and *right*. The effect of the program is to build a tree rooted at h and then rotate the children of h. Although the rotation creates a data structure of type *TREE*, there are intermediate points (B and C) where the data structure is a *DAG*.

In order to analyse *DAGS* it is necessary to maintain more information in the path matrix. In addition to the relationships among live handles, information about nodes that have more than one parent is also required. A *dag node* is a node for which there may be more than one parent (more than one incoming edge). For each dag node, the required information is: (1) an estimate of its position relative to the live handles and other dag nodes, and (2) an estimate of its reference count.

The basic analysis functions for *DAGS* are straightforward extensions to those for *TREES*. Since the number of dag nodes created is not known statically, dag nodes are approximated by associating a *dag handle* with each handle update statement in the program. For example, in the program in figure 3.16, a dag handle would be associated with the statements $h.left := m$, $h.middle := r$, and $h.right := l$. If the analysis of the statement indicates that a dag node is possible, then the dag handle associated with the statement is placed in the path matrix. If analysis of a later statement indicates that the dag handle now refers to a node that has at most one parent (reference count < 2), the dag handle can be removed from the path matrix, and the analysis can continue as in the *TREE* case. In the case of the example

```
A.left := nil
  ∀ $h_i \in H'$ .
      $p'.IsNode[h_i] \mapsto$
        if $h_i = A$ then
          (•, ∘ (p.IsNode[A].right))
        else if path($h_i$,S,A,p) then
          (p.IsNode[$h_i$].self, ∘ (p.IsNode[$h_i$].right))
        else if path($h_i$,S?,A,p) then
          (p.IsNode[$h_i$].self,
            ( ∘ ⊓(p.IsNode[$h_i$].left)
            (p.IsNode[$h_i$].right)
          )
        else
          p.IsNode[$h_i$];
A.left := B
  ∀ $h_i \in H'$ .
      $p'.IsNode[h_i] \mapsto$
        if $h_i = A$ then
          (•, (p.IsNode[B].self) (p.IsNode[A].right))
        else if path($h_i$,S,A,p) then
          (p.IsNode[$h_i$].self, (p.IsNode[B].self) (p.IsNode[$h_i$].right))
        else if path($h_i$,S?,A,p) then
          (p.IsNode[$h_i$].self,
            ((p.IsNode[B].self) ⊓ (p.IsNode[$h_i$].left))
            (p.IsNode[$h_i$].right))
          )
        else
          p.IsNode[$h_i$];
```

Figure 3.15: Nil estimate for *A.left* := **nil** and *A.left* := *B*.

```
program DagDemo
nodetype left, middle, right: handle; value: int end;
procedure main()
  h,l,m,r: handle
begin
  { ... build a tree rooted at h, let l, m, r be handles to its children }
  h := BuildTree();
  l := h.left;
  m := h.middle;
  r := h.right;
  { rotate children left }
  { ⇐ PROGRAM POINT A – path matrix P_A }
  h.left := m;
  { ⇐ PROGRAM POINT B – path matrix P_B }
  h.middle := r;
  { ⇐ PROGRAM POINT C – path matrix P_C }
  h.right := l;
  { ⇐ PROGRAM POINT D – path matrix P_D }
  { use h,m,l,r .... }
end;
```

P_A

	h	l	m	r
h	S	L^1	M^1	R^1
l		S		
m			S	
r				S

P_B

	h	l	m	r	m:0@1{2}
h	S		L^1, M^1	R^1	L^1, M^1
l		S			
m			S		S
r				S	
m:0@1			S		S

P_C

	h	l	m	r	r:1@1{2}
h	S		L^1	M^1, R^1	M^1, R^1
l		S			
m			S		
r				S	S
r:1@1				S	S

P_D

	h	l	m	r
h	S	R^1	L^1	M^1
l		S		
m			S	
r				S

Figure 3.16: An example program that creates DAG nodes.

program, the analysis of the statement $h.left := m$ causes the dag handle *m:0@1* (with reference count 2) to be added to P_B; similarly $h.middle := r$ causes the dag handle *r:1@1* to be added to P_C. Each of these dag handles is removed when its reference count is decreased to 1.

3.4.1 Data abstraction for reference counts

A crucial component of the *DAG* analysis is the abstraction used for estimating reference counts. An abstracted reference count is either a positive integer, or a special designation (++) indicating that the reference count is stuck. In figure 3.17 we give the operations to increment, decrement and merge reference counts.

```
IncrRef(n) =
  if n = ++ then ++ else n+1
DecrRef(n) =
  if n = ++ then ++ else n-1
grow_merge(n,m) =
  if n = ++ or m = ++ then
    ++
  else
    if n = m then n else ++
max_merge(n,m) =
  if n = ++ or m = ++ then ++ else max(n,m)
```

Figure 3.17: Operations on abstract reference counts.

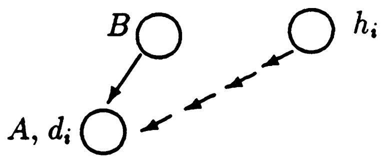
3.4.2 Analysis Rules for *DAG* estimates.

Handle assignment statements of the form: $A :=$ **nil**, $A := new()$, and $A := B$ are not changed to handle *DAG* estimates. However, the analysis rule for handle assignments of the form $A := B.f$ must be slightly modified. As illustrated for $A := B.left$ (figure 3.18), we must now consider the possibility that $B.left$ points to a *DAG* node (a node with more than one parent). In this case, it is *not* sufficient to calculate $p'[h_i, A]$ from $p[h_i, B]$ (there may be paths from h_i to A that do not go through B). Thus, $p'[h_i, A]$ must inherit the paths from the *DAG* nodes equal to $B.left$. If dag node d_i is definitely equal to $B.left$, then the paths from $p[h_i, d_i]$ are mapped directly to paths in $p'[h_i, A]$, otherwise if d_i is possibly equal to $B.left$, then the paths from $p[h_i, d_i]$ are mapped to possible paths in $p'[h_i, A]$.

The two forms of handle update statements also require some modification to extend to the *DAG* analysis. First consider the modifications for $A.left :=$ **nil**.

```
$A := B.left$
    $\forall\ h_i,\ h_j \in H'$ .
        $p'[h_i,\ h_j] \mapsto$
            if $h_i = A$ then
                if $h_j = A$ then
                    $\{S\}$
                else if $h_j = B$ then
                    $\{\}$
                else
                    map below_rule $p[B,h_j]$
            else if $h_j = A$ then
                if $h_i = B$ then
                    $\{L^1\}$
                else
                    map above_rule $p[h_i,B]$
            else
                $p[h_i,\ h_j]$

    $\forall\ d_i \in dags(H)$ .
        if $path(B,L,d_i,p)$ then
            $inherit\_entries(p,d_i,A,p')$
        else if $path(B,L?,d_i,p)$ then
            $inherit\_possible\_entries(p,d_i,A,p')$
```



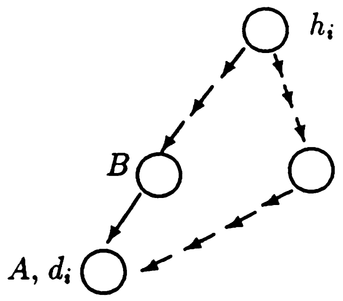
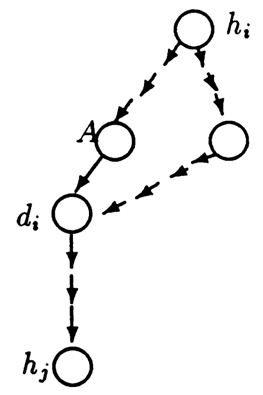


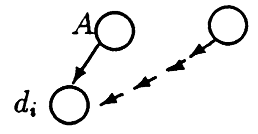
Figure 3.18: Extended rule for $A := B.left$.

A.left := **nil**

```
∀ $h_i$, $h_j$ ∈ H' .
    $p'[h_i, h_j] \mapsto$
        if $path(h_i,*,A,p)$ and
          $path(A,L*,h_j,p)$ and
          not $(dag\_between(A,L?,h_j,p))$ then
          {}
        else if $path(h_i,*?,A,p)$ and
                $path(A,L*?,h_j,p)$ then
          if $path(h_i,S,A,p)$ then
            $remove\_head(L,p[h_i,h_j])$
          else if $path(A,L,h_j,p)$ then
            $remove\_tail(L,p[h_i,h_j])$
          else
            $make\_possible(p[h_i,h_j])$
        else
          $p[h_i, h_j]$;

∀ $d_i$ ∈ $dags(H)$ .
    if $path(A,L,d_i,p)$ or
      ( $path(A,L*,d_i,p)$ and
        not $(handle\_between(A,L?,d_i,p))$ ) then
      $p'.RefCnt[d_i] \mapsto DecrRef(p.RefCnt[d_i])$
    else
      $p'.RefCnt[d_i] \mapsto p.RefCnt[d_i]$
```





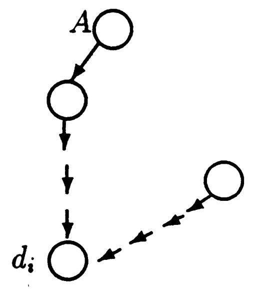


Figure 3.19: Extended rule for *A.left* := **nil**.

There are two differences between the rule presented for *TREES* in figure 3.9 and the rule for *DAGS* in figure 3.19, The first modification is required for the rule that removes all links between handles h_i and h_j. In the *TREE* case it was sufficient to check that h_i was definitely above or equal to A, and that $A.left$ was definitely above or equal to h_j. However, as illustrated by the topmost diagram in figure 3.19, we must also check that there is no *DAG* node along the path from $A.left$ to h_j.

The second modification calculates the new reference counts for p'. If a dag node d_i is definitely the same as $A.left$, then executing the statement $A.left :=$ **nil** reduces the number of parents of d_i by 1. Generalizing on this observation, note that a *DAG* node d_i that is definitely below or equal to $A.left$ may also be decremented if there are no other live handles or *DAG* nodes along the path from $A.left$ to d_i. These two situations are illustrated by the bottom diagrams in figure 3.19.

Statements of the form $A.f := B$ may create new *DAG* nodes. As illustrated by the rule for $A.left := B$ (figure 3.20), we can easily extend the *TREE* rule from figure 3.10 to handle the *DAG* case. As in the *TREE* rule, path matrix p' is an estimate of the effect of $A.left :=$ **nil**, and p'' is produced by adding to p' the effects of linking $A.left$ to B. Note that each handle update statement has an associated *DAG* handle, $dagname(B)$. If the execution the statement $A.f := B$ creates a possible *DAG*, the appropriate *DAG* handle entries are used to encode the relationships between all other live handles and B. If the *DAG* node has not yet been used (reference count < 2), then the relationships $p''[h_i, dagname(B)]$ are the same as the corresponding entries for $p''[h_i, B]$. If the *DAG* node is already in use, then the old path expressions in $p'[h_i, dagname(B)]$ must be merged with the new relationships for $p''[h_i, B]$.

3.5 Analysing Compound Statements

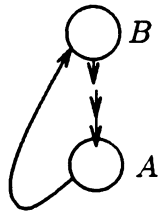
Analysis functions have been defined for each kind of basic handle statement. In this section we use these basic functions as building blocks in developing analysis functions for blocks (statement sequences), conditional statements, and while loops. In discussing while loops, an overview of the fixed-point calculation and merging ($\bowtie$) of path matrices is given. A more complete discussion of merging and fixed-point computations is given in chapter 4.

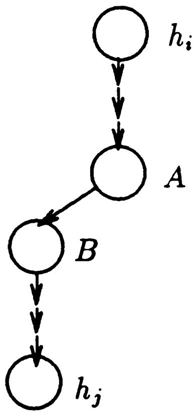
Given an input path matrix p_0 and a sequence of statements $s_1; s_2; ...; s_n$ from a block, the final path matrix p_n is produced as follows: for each statement, s_i, for $i = 1..n$, the statement analysis function is applied to p_{i-1} and s_i resulting in p_i.

Given an input path matrix p and a conditional statement of the form **if** $\langle expr \rangle$ **then** $\langle stmt_1 \rangle$ **else** $\langle stmt_2 \rangle$, a pair of path matrices $\langle p_{then}, p_{else} \rangle$ is produced as follows: the statement analysis function is applied to p and $stmt_1$ to produce p_{then}, and the statement analysis function is applied to p and $stmt_2$ to produce p_{else}. The analysis may continue with the pair of path matrices $\langle p_{then}, p_{else} \rangle$, or with one merged path matrix $p_{then} \bowtie p_{else}$.

$\underline{A.left := B}$

```
    $p' \mapsto f(A.left{:=}nil,p)$;
    if $path(B,*?,A,p')$ then
      $possible\_cycle()$
    else
    $\forall\ h_i,\ h_j \in H'$ .
        $p''\ [h_i,\ h_j] \mapsto$
          if $path(h_i,*?,A,p')$
                and $path(B,*?,h_j,p')$ then
            if $is\_node(B)$ then
              $p'[h_i,h_j] \cup$
                ( $connect\_all$
                      ( $p'[h_i,\ A]$,
                        $\langle L^1, true\rangle$,
                        $p'[B,\ h_j]$
                      )
                )
            else
              $p'[h_i,h_j] \cup$
                ( $connect\_all$
                      ( $p'[h_i,\ A]$,
                        $\langle L^1, true\rangle$,
                        $make\_possible(p'[B,\ h_j])$
                      )
                )
          else
            $p'[h_i,\ h_j]$;
    if $handle\_above(B,p')$ then
      if $unused\_dag(dag\_name(B),p')$ then
        $p''.RefCnt[dag\_name(B)] \mapsto 2$;
        $init\_entries(p'',Br,dag\_name(B))$
      else
        $p''.RefCnt[dag\_name(B)] \mapsto$ ++;
        $merge\_entries(p'',B,dag\_name(B))$;
      $\forall\ d_i \in dags(H)$ .
          if $path(B,S?,d_i,p')$ then
            $p''.RefCnt[d_i] \mapsto IncrRef(p'.RefCnt[d_i])$
```





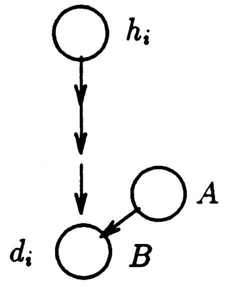


Figure 3.20: Extended rule for $A.left := B$.

Given an input path matrix and a while loop of the form **while** $\langle expr \rangle$ **do** $\langle stmt \rangle$, the analysis produces a pair of path matrices, $\langle p_0, p_+ \rangle$. The path matrix p_0 represents zero iterations of the while loop and is equal the input path matrix p. The output path matrix p_+ approximates one or more iterations of the while loop, and is computed using a fixed-point iterative approximation. The statement analysis function is applied to p and *stmt* to produce the first estimate, p_1. For each iteration i, the analysis function is applied to p_i and *stmt* to produce p_i'. The next iteration begins with p_{i+1}, where $p_{i+1} = p_i \bowtie p_i'$. The iterative approximation terminates when $p_i = p_{i+1}$. Figure 3.21 illustrates the iterative approximation for a simple while loop.

```
$l := h$;
while $l.left \neq$ nil do
  $l := l.left$
```

	h	l
h	S	S
l	S	S

p_0

	h	l
h	S	L^1
l		S

p_1

	h	l
h	S	L^+
l		S

p_2

	h	l
h	S	L^+
l		S

$p_3 = p_+$

Figure 3.21: Iterative approximation for a simple while loop.

3.5.1 Merging Path Matrices

The *merge* of two path matrices p_1 and p_2 is calculated as follows. Each entry in m_1 and m_2 is simplified by squashing all path expressions into one representative path expression using the or-merging ($\bowtie^{\vee}$) operation. The merge path matrix, $p_{merged} = p_1 \bowtie p_2$, is then calculated by the item-wise and-merging given by: $p_{merged}[h_i, h_j] = p_1[h_i, h_j] \bowtie^{\wedge} p_2[h_i, h_j]$. An example of the path matrix merging operation is given in figure 3.22.

Given the merge operation, it is possible to show that the iterative approximation for the while loop terminates. Path matrix p_1 equals path matrix p_2 when $p_1(h_i, h_j) = p_2(h_i, h_j)$ for all entries. Let $p_1, p_2, ..., p_n$ be the sequence of path matrix estimates calculated by the iterative approximation. Given the previous definitions of merge and squash, we know that the representative path expression for any element $p_k(h_i, h_j)$ can only be the same or more general than the representative path expression of the same element in $p_{k-1}(h_i, h_j)$. Also, the minimal length of the representative path expression in $p_k(h_i, h_j)$ must be no larger than the minimal length of the representative path expression in $p_{k-1}(h_i, h_j)$. Thus, the series of approxi-

Original path matrices

	a	b	c
a	S	L^1R^+, L^1L^+	L^1
b		S	S
c		S	S

	a	b	c
a	S	$L^1D^+L^1$, $L^1D^+R^1$	R^1
b		S	
c			S

Simplified matrices, each item squashed

	a	b	c
a	S	L^1D^+	L^1
b		S	S
c		S	S

	a	b	c
a	S	$L^1D^+D^1$	R^1
b		S	
c			S

Result of merging the two simplified matrices

	a	b	c
a	S	L^1D^+	D^1
b		S	$S?$
c		$S?$	S

Figure 3.22: An example of the merge operation.

mations for each element $p_1(h_i, h_j), p_2(h_i, h_j), ..., p_n(h_i, h_j)$ is finite, and the iterative approximation for while loops terminates.

3.6 Analysing Procedure Calls

We now complete our description of the interference analysis by presenting an overview of the analysis function for procedure and functions calls. This section provides an operational view of the methods and is intended to give some insight into the our approach to handling procedures and recursion. A more complete, formal presentation is given in chapter 4. Also note that a selection of interference analysis examples for programs using linked lists given in appendix C.

First, consider the case of a call to a non-recursive procedure. Given an input path matrix p and a procedure call of the form $f(h_1, ..., h_n)$, the resulting path matrix p' is produced as follows. The body of procedure f is analysed with an input path matrix q, where q is a path matrix that combines path information of the handles live at the point of the call to f and the handles which correspond to formal parameters of f. Note that by including the handles live at the point of the call in q, we are effectively encoding the context of the call.

Consider as an example the call *SwapChildren(root)* in figure 3.23. In this example, the call to *SwapChildren* changes the relationships between handles *root*, *l_child* and *r_child* by swapping the left and right sub-trees of *root*. Figure 3.23 illustrates four stages of the path matrix computation. P_A represents the path matrix at the point just before the call *SwapChildren(root)*. P_B, the path matrix at the beginning of the body of *SwapChildren*, combines the relationships of calling context handles and the parameter handles. In our example, the handles *l_child* and *r_child* must be included in the calling context since they are below the handle *root*. In P_B, *l_child@2* and *r_child@2* capture the context information of the call, *h*1* is used as a symbolic name for the argument handle (*root*), and *h* is a local handle whose initial relationships are the same as those of *h*1*. P_C illustrates the path matrix resulting from the analysis of the body, while P_D is the final result of the call.

Note that it is not always necessary to include all of the information about the context. We need include only handles which may be affected by the evaluation of the body of the procedure. For example, if we know that the data structure is a *TREE*, then handles that are above the argument handles cannot be reached by a computation in the body of the procedure, and therefore they need not be included in the context. Thus, we can reduce the complexity of the computation by using properties of the data structure.

As illustrated in figure 3.24, the analysis for function calls is similar to the analysis for procedures. In the case of functions, a function call is of the form $r_1, \ldots, r_m := f(h_1, \ldots, h_n)$, where $r_1, \ldots, r_m$ are variables that receive the results of evaluating the function call. When r_i is a handle variable, we must compute the relationships between r_i and all other handles live after the call to f. These

```
program DemoProcCall
nodedef left, right: handle; value: int end;
procedure SwapChildren(h: handle)
  l, r: handle
begin
  { ⇐ PROGRAM POINT B – path matrix P_B }
  l := h.left;
  r := h.right;
  h.left := r;
  h.right := l
  { ⇐ PROGRAM POINT C – path matrix P_C }
end;
procedure main()
  root, l_child, r_child: handle
begin
  { .... create a tree at root .... }
  l_child := root.left;
  r_child := root.right;
  { ⇐ PROGRAM POINT A – path matrix P_A }
  SwapChildren(root);
  { ⇐ PROGRAM POINT D – path matrix P_D }
  { ... use l_child, r_child, root ... }
end
```

P_A

	root	*l_child*	*r_child*
root	*S*	L^1	R^1
l_child		*S*	
r_child			*S*

P_B

	l_child@2	*r_child@2*	*h∗1*	*h*
l_child@2	*S*			
r_child@2		*S*		
h∗1	L^1	R^1	*S*	*S*
h	L^1	R^1	*S*	*S*

P_C

	l_child@2	*r_child@2*	*h∗1*
l_child@2	*S*		
r_child@2		*S*	
h∗1	R^1	L^1	*S*

P_D

	root	*l_child*	*r_child*
root	*S*	R^1	L^1
l_child		*S*	
r_child			*S*

Figure 3.23: Example of a procedure call.

```
program SimpleFunction
nodedef left, right: handle;   value: int end;
function FindMin(h: handle) handle
  m : handle
begin
  { ⇐ PROGRAM POINT B – path matrix $P_B$ }
  m := h;
  while m.left ≠ nil do
    m := m.left
  { ⇐ PROGRAM POINT C – path matrix $P_C$ }
end => return(m);
procedure main()
  root, min: handle
begin
  { .... create a tree at root .... }
  { ⇐ PROGRAM POINT A – path matrix $P_A$ }
  min := FindMin(root);
  { ⇐ PROGRAM POINT D – path matrix $P_D$ }
  { .... use root and min .... }
end;
```

P_A

	root
root	*S*

P_B

	h∗3	*m*
h∗3	*S*	*S*
m	*S*	*S*

P_C

	h∗3	*m*
h∗3	*S*	$(S+L^+)$
m	*S?*	*S*

P_D

	root	*min*
root	*S*	$(S+L^+)$
min	*S?*	*S*

Figure 3.24: Example of a function call.

relationships are encoded in the path matrix valid at end of the body of f. As an example, consider the call $min := FindMin(root)$ in figure 3.24. At program point P_C, $h{*}3$ is a symbolic name corresponding to *root* in the caller's context, and m represents the resulting handle. Thus, path matrix P_D is constructed from P_D by associating $h{*}3$ with *root*, and m with *min*.

Analysis of calls to recursive procedures and functions requires a combination of the techniques used in non-recursive procedure calls and the techniques used for fixed-point approximations. Consider a procedure f which contains one or more recursive calls to f in its body. We can define analysis functions for the recursive calls as an iterative approximation of the pair of path matrices $\langle p\_in_i, p\_out_j \rangle$. The path matrix p_in_i represents the merge of all possible path matrices at the beginning of the body of f, while p_out_j represents the merge of all possible path matrices at the end of the body of f. The iterative approximation terminates when $p\_in_i = p\_in_{i+1}$ and $p\_out_j = p\_out_{j+1}$.

As an example of the iterative approximation for the recursive procedures, consider the approximation for *reverse* given in figure 3.25. In this case, the first input approximation p_in_1 is due to the call $reverse(root)$ and the initial output approximation p_out_0 is *UNDEFINED.* The first defined output approximation p_out_1 is computed by analysing all non-recursive paths in the body of $reverse$. In this case the only non-recursive path is the empty statement, and thus p_out_1 is trivial to compute from p_in_1. The second input approximation p_in_2 is due to the recursive call $reverse(l)$. Note that at this point, path matrix p_A captures all the relationships among h, l, and r. Since, l does not have any live handles below it, we need only consider the effect on l. Encoding the call $reverse(l)$ gives p_in_2. Since p_in_2 is equal to $p \in_1$, p_out_1 can be used as an estimate of the effect of the call to $reverse(l)$, and similarly for the call to $reverse(r)$. The resulting path matrix at the end of the body of reverse is p_out_2. Since both $p\_in_1 = p\_out_1$, and $p\_out_1 = p\_out_2$, the iterative approximation terminates and p_out_2 is used to build the result of the call

as recursive procedures. Con-

$$\mathcal{E} : Expression \rightarrow (Env \rightarrow (Mem \rightarrow Value))$$
$$\mathcal{M} : Statement \rightarrow (Env \rightarrow (Mem \rightarrow Mem))$$
$$\mathcal{D} : Definition \rightarrow (Env \rightarrow Env)$$
$$\mathcal{P} : Program \rightarrow (File \rightarrow File + \underline{error})$$

```
nodedef left, right: handle;
procedure reverse(h:handle)
  l, r: handle
begin
{ ⇐ p_in_i (a merge of all input path matrices) }
  if h ≠ nil then
    begin
      l := h.left;
      r := h.right;
      { ⇐ p_A (before recursive call) }
      reverse(l);
      reverse(r);
      h.left := r;
      h.right := l
    end
{ ⇐ p_out_j (a merge of all output path matrices) }
end;

procedure main()
  root: handle
begin
  { ... build a tree at root ... }
  root := BuildTree();
  reverse(root) ;
  { .... use root .... }
end;
```

p_A

	*h*1*	*h*	*l*	*r*
*h*1*	*S*	*S*	L^1	R^1
h	*S*	*S*	L^1	R^1
l			*S*	
r				*S*

$$[p\_in_1, p\_out_0] = \left[\begin{array}{c|c|c} & h{*}1 & h \\ \hline h{*}1 & S & S \\ h & S & S \end{array}, UNDEFINED \right]$$

$$[p\_in_1, p\_out_1] = \left[\begin{array}{c|c|c} & h{*}1 & h \\ \hline h{*}1 & S & S \\ h & S & S \end{array}, \begin{array}{c|c} & h{*}1 \\ \hline h{*}1 & S \end{array} \right]$$

$$[p\_in_2, p\_out_2] = \left[\begin{array}{c|c|c} & h{*}1 & h \\ \hline h{*}1 & S & S \\ h & S & S \end{array}, \begin{array}{c|c} & h{*}1 \\ \hline h{*}1 & S \end{array} \right]$$

Figure 3.25: Estimating the path matrices for reverse.

```
program RecursiveFunction
nodedef left, right: handle;   value: int end;
function FindMin(h: handle) handle
  t, m : handle
begin
{ ⇐ $p\_in_i$ (a merge of all input path matrices) }
  if h.left = nil then
    m := h
  else
    begin
      t := h.left;
      { ⇐ $p_A$ (before the recursive call) }
      m := FindMin(t)
    end
{ ⇐ $p\_out_j$ (a merge of all output path matrices) }
end => return(m);
procedure main()
  root, min: handle
begin
  { ... build a tree at root ... }
  min := FindMin(root);
  { .... use min, root .... }
end;
```

p_A

	$h{*}3$	t
$h{*}3$	S	L^1
t		S

$$[p\_in_1, p\_out_0] = \left[\begin{array}{c|c|c} & h{*}3 & h \\ \hline h{*}3 & S & S \\ h & S & S \end{array}, UNDEFINED \right]$$

$$[p\_in_1, p\_out_1] = \left[\begin{array}{c|c|c} & h{*}3 & h \\ \hline h{*}3 & S & S \\ h & S & S \end{array}, \begin{array}{c|c|c} & h{*}3 & m \\ \hline h{*}3 & S & S \\ m & S & S \end{array} \right]$$

$$\left[\begin{array}{c|c|c} & h{*}3 & h \end{array} \quad \begin{array}{c|c|c} & h{*}3 & m \end{array} \right]$$

$$\mathcal{E} : Expression \rightarrow (Env \rightarrow (Mem \rightarrow Value))$$
$$\mathcal{M} : Statement \rightarrow (Env \rightarrow (Mem \rightarrow Mem))$$
$$\mathcal{D} : Definition \rightarrow (Env \rightarrow Env)$$
$$\mathcal{P} : Program \rightarrow (File \rightarrow File + \underline{error})$$

where

pter 4

Formal Description of Interference Analysis

In this chapter, we provide a formal presentation of the interference analysis. In particular, we develop a semantic framework for the method, and present the abstract rules including a detailed discussion of the fixed-point approximation for while loops and recursive procedures. We argue that the method is sound with respect to a denotational semantics and that the method is efficient.

In section 4.1 we give a formal description of SIL, and in section 4.2 we give a formal description of the data structure abstraction along with the definition of important properties of path matrices. Section 4.3 contains an outline of the abstract semantics with particular emphasis on the rules for while loops and procedures, and in section 4.4 we establish soundness of the method.

4.1 A Formal Description of SIL

In chapter 2, we defined a simple imperative language that captures the important features of imperative languages that support recursively defined pointer structures.

We define a standard semantics for SIL using the semantic domains and functions outlined in figure 4.1. The standard semantics is quite straightforward and is similar to semantic presentations in [Gor79, Sto77, Ten81]. The three domains of interest are: (1) the *environment* which maps identifiers to locations or procedures, (2) the *store* which maps locations to values, and (3) the *heap* which maps node pointers to nodes. A *memory* is defined as a six-tuple containing a store, a heap, a stack pointer for the store, a heap pointer, an input file and an output file. There are four semantic functions: $\mathcal{E}$ for expressions, $\mathcal{M}$ for statements, $\mathcal{D}$ for definitions, and $\mathcal{P}$ for programs. The complete definitions for the semantic functions are given in appendix A.

The model of the environment and memory is illustrated in figure 4.2. We see that a *memory* has both a *store* and a *heap* as components. The space allocated for variables is modelled with the store, while the nodes of pointer data structures are

$$\mathcal{E} : Expression \rightarrow (Env \rightarrow (Mem \rightarrow Value))$$
$$\mathcal{M} : Statement \rightarrow (Env \rightarrow (Mem \rightarrow Mem))$$
$$\mathcal{D} : Definition \rightarrow (Env \rightarrow Env)$$
$$\mathcal{P} : Program \rightarrow (File \rightarrow File + \underline{error})$$

where

$$Env = Id \rightarrow Loc + Procedure$$
$$Store = Loc \rightarrow Value$$
$$Heap = NodePtr \rightarrow ((Int + \underline{undef}) \times (NodePtr + \underline{nil}) \times (NodePtr + \underline{nil}))$$
$$Value = Int + NodePtr + \underline{nil} + \underline{undef}$$
$$Procedure = Value^n \rightarrow (Mem \rightarrow Mem)$$
$$File = Int\ \mathbf{list}$$
$$NodePtr = Int$$
$$Loc = Int$$
$$Mem = (Store \times Heap \times Loc \times NodePtr \times File \times File) + \underline{error}$$

Figure 4.1: Semantic domains for SIL.

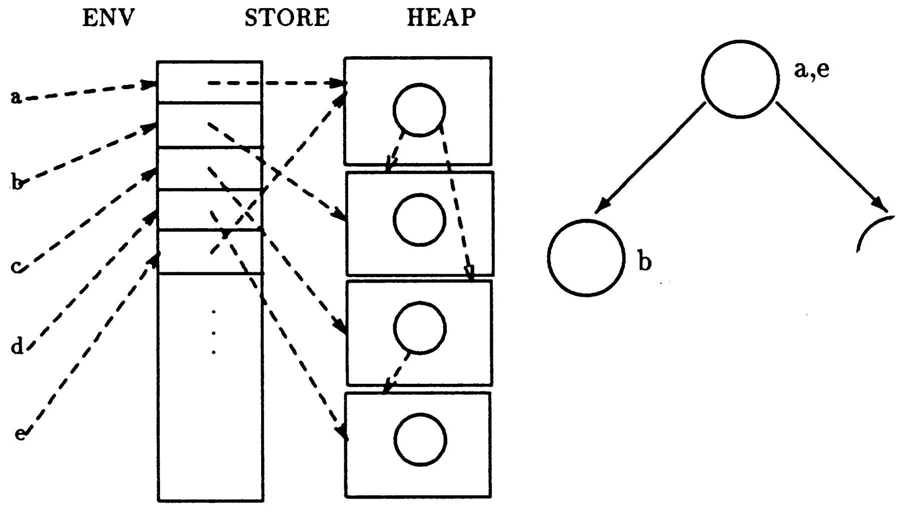


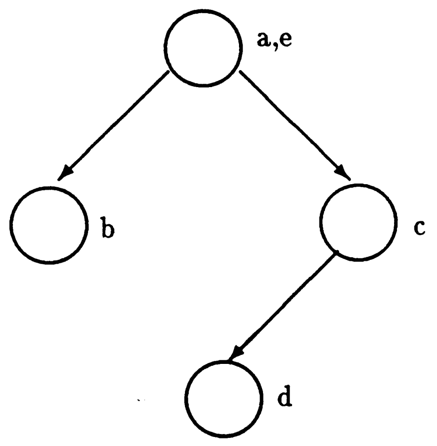
Figure 4.2: The model for the

Given a domain for pa
chain is $O(n)$. This re
expression will take
Given the defi
the merge of tw
Conditio
In
of i

ap. The tree in figure 4.2 gives a more abstract view of the con the environment and memory. In chapter 3 we discussed the interf ysis as estimating the paths between each pair of live handles h_i and h_j. semantic model this is equivalent to estimating the paths between $store(env(h_i))$ and $store(env(h_j))$ in the *heap*.

4.2 Path Matrices Revisited

The purpose of our analysis is to approximate the reachability relationships among accessible nodes in the heap. We represent this approximation with a path matrix that gives the relationships between all pairs of live handles. Given an environment defined on a set of variables, we call the set of variables with type **handle** the *handles* to the heap. The set of *live handles* are those variables which may be used in the remaining computation. For each pair of live handles A and B, we wish to approximate the path between the heap node accessed via handle A and the heap node accessed via handle B. We say that there is a path β from handle A to handle B if β *connects* the heap node $store(env(A))$ to the heap node $store(env(B))$. A *path* is denoted by either S (the heap nodes are the same), or by a non-empty sequence of field indicators (L or R for binary trees).

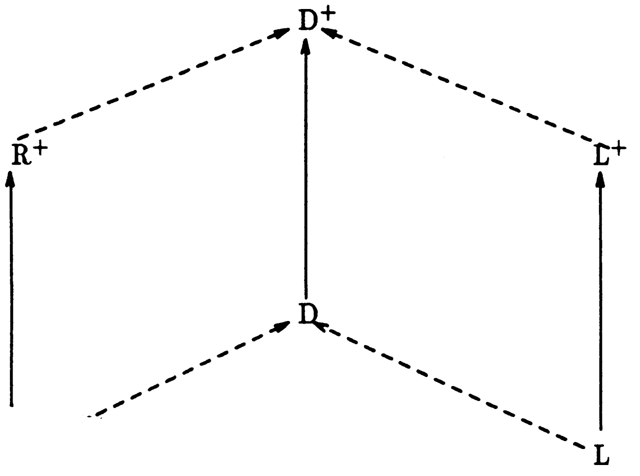


	a	b	c	d	e
a	S	L^1	R^1	R^1L^1	S
b		S			
c			S	L^1	
d				S	
e	S	L^1	R^1	R^1L^1	S

Figure 4.3: A tree and the corresponding path matrix

Figure 4.3 gives a path matrix that approximates the relationships between live handles in the heap pictured in figure 4.2. Note that this path matrix gives exactly the paths in the associated heap. In general, the path matrices calculated for programs with conditionals, while loops, and recursion must contain path estimates. Path estimates are captured by *path expressions* which denote sets of paths. The definitions for path expressions and operations on path expressions were given in chapter 3.

An important aspect of path expressions is that they can be given a natural partial order that represents their information content. A path expression is more informative (less general) if it encompasses fewer actual paths. The partial ordering is in the direction of increasing generality, not increasing information. Figure 4.4 gives the partial ordering for the path expressions of length 1, while 4.5 gives the partial ordering for path expressions of length 1 or 2. Note that all path expressions with the same normal form have been grouped together. The introduction of D's (dashed lines) represents direction approximation, while the introduction of $^+$ superscripts (solid lines) represents length approximation.



Given a domain for path expressions with length[1] n or less, the longest ascending chain is $O(n)$. This relationship ensures that an iterative approximation of a path expression will take at most $O(n)$ steps.

Given the definition of *merge* and *squash* presented in chapter 3, it is clear that the *merge* of two path expressions, $p_1 \bowtie p_2$, must obey the following condition:

Condition 1. $(p_1 \bowtie p_2) \subseteq (p_1 \cup p_2)$

In order to obtain an accurate and sound analysis one must calculate two kinds of information. Roughly speaking, one can think of this as positive and negative information. More precisely, there is information that states that a path of a certain form *might* exist (possible path expression) and that a path of a certain form *must* exist (definite path expression). One uses path expressions to represent both types of information, in order to distinguish the two we use a question mark to represent possible information.

We extend the ordering on path expressions as follows:

Definition 10. If p_1 and p_2 are path expressions without question marks and $p_1 \subseteq$ p_2 then the relation $\leq$ is given by: *(1)* $p_1 \leq p_2$, *(2)* $p_1 \leq p_2?$

The ordering can be

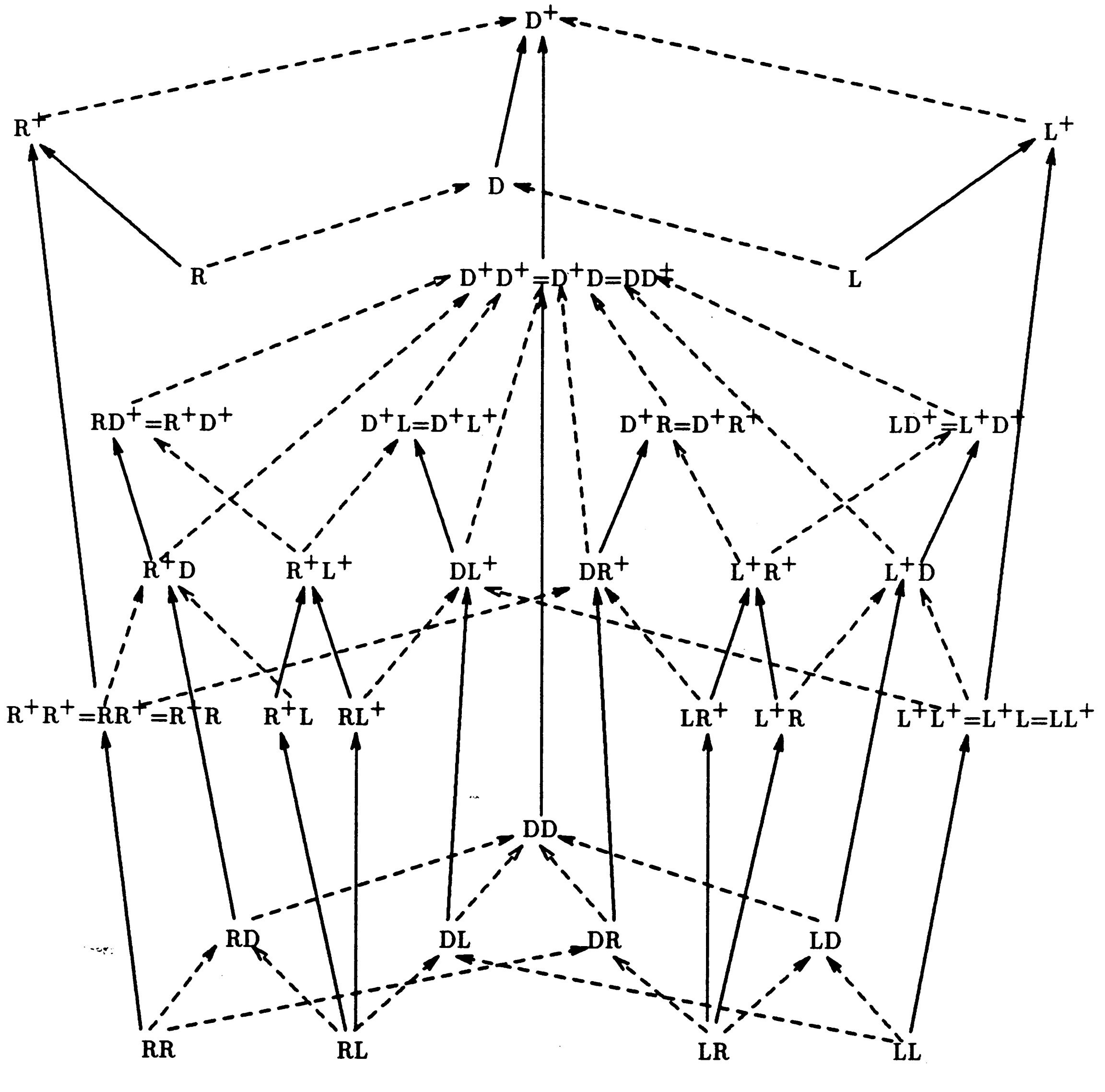


Figure 4.5: Partial ordering of path expressions of length 1 or 2

```
$\mathcal{M}[\![ A := nil ]\!]\ aenv\ p = p'$,  where
  $\forall\ h_i, h_j \in H'$ .
      $p'[h_i, h_j] \mapsto$
        if $h_i = A$ and $h_j = A$ then
          $\{S\}$
        else if $h_i = A$ or $h_j = A$ then
          $\{\}$
        else
          $p[h_i, h_j]$
  $\forall\ h_i \in H'$ .
      $p'.IsNode[h_i] \mapsto$
        if $h_i = A$ then $(\circ, \circ\circ)$ else $p.IsNode[h_i]$;

$\mathcal{M}[\![ A := new() ]\!]\ aenv\ p = p'$,  where
  $\forall\ h_i, h_j \in H'$ .
      $p'[h_i, h_j] \mapsto$
        if $h_i = A$ and $h_j = A$ then
          $\{S\}$
        else if $h_i = A$ or $h_j = A$ then
          $\{\}$
        else
          $p[h_i, h_j]$
  $\forall\ h_i \in H'$ .
      $p'.IsNode[h_i] \mapsto$
        if $h_i = A$ then $(\bullet, \circ\circ)$ else $p.IsNode[h_i]$;

$\mathcal{M}[\![ A := B ]\!]\ aenv\ p = p'$,  where
  $\forall\ h_i, h_j \in H'$ .
      $p'[h_i, h_j] \mapsto$
        if $h_i = A$ then
          if $h_j = B$ or $h_j = A$ then $\{S\}$ else $p[B,h_j]$
        else if $h_j = A$ then
          if $h_i = B$ then $\{S\}$ else $p[h_i,B]$
        else
          $p[h_i, h_j]$
  $\forall\ h_i \in H'$ .
      $p'.IsNode[h_i] \mapsto$
        if $h_i = A$ then $p.IsNode[B]$ else $p.IsNode[h_i]$;
```

Figure 4.9: Abstract semantic functions for handle assignment statements.

```
$\mathcal{A}[\![$ begin end $]\!]$ $aenv\ p =\ p$

$\mathcal{A}[\![$ begin $s_1$ ; $s_2$ ; ... ; $s_n$ end $]\!]$ $aenv\ p =$
  $\mathcal{A}[\![$ begin $s_2$ ; ... ; $s_n$ end $]\!]$ $aenv$ ( $\mathcal{A}[\![\ s_1\ ]\!]$ $aenv\ p$ )

$\mathcal{A}[\![$ if $exp$ then $s_1$ else $s_2$ $]\!]$ $aenv\ p =$
  ( $\mathcal{A}[\![\ s_1\ ]\!]$ $aenv\ p$ ) $\bowtie$ ( $\mathcal{A}[\![\ s_2\ ]\!]$ $aenv\ p$)

$\mathcal{A}[\![$ while $e$ do $s$ $]\!]$ $aenv\ p = A\ p$
  where
    $A = \lambda p$ .
      let
        $new\_p = (\mathcal{A}[\![\ s\ ]\!]\ aenv\ p) \bowtie p$
      in
        if $new\_p = p$ then $p$ else $A\ new\_p$

$\mathcal{A}[\![$ repeat $s$ until $e$ $]\!]$ $aenv\ p = A\ (\mathcal{A}[\![\ s\ ]]\ aenv\ p)$
  where
    $A = \lambda p$ .
      let
        $new\_p = p \bowtie (\mathcal{A}[\![\ s\ ]\!]\ aenv\ p)$
      in
        if $new\_p = p$ then $p$ else $A\ new\_p$
```

Figure 4.10: Abstract semantic functions for compound statements.

we can bound the number of iterations at $O(mn^2)$, where m is the longest path in the input path matrix, and n is the number of live handles. In practice, we observe that only a small subset of the live handles are affected by the loop, and the handles that are affected tend to be related. Thus, the number of iterations is usually small.

```
𝓜⟦ while e do s ⟧ env mem = fix(W) mem
  where
    W = λT . λm .
              if ℰ⟦ e ⟧ env m then T( 𝓜⟦ s ⟧ env m ) else m

𝒜⟦ while e do s ⟧ aenv p = A p
  where
    A = λp .
      let
        new_p = (𝒜⟦ s ⟧ aenv p) ⋈ p
      in
        if new_p = p then p else A new_p
```

Figure 4.11: Semantic functions for **while** loops.

In figure 4.12, we give the abstract semantic function for procedure calls and in figure 4.13 we give the semantic functions for the definition of non-recursive procedures. A call to procedure q simply looks up the definition of $q.D$ in the abstract environment and applies D to the handle arguments. The abstract definition of a procedure q causes the name $q.D$ to be bound to a function D in the abstract environment.

```
𝒜⟦ q( a₁ , a₂, ... , aₙ ) ⟧ aenv P =
  let h₁ ... hₘ = handles(a₁ ... aₙ) in
      aenv(q.D) h₁ ... hₘ aenv P
```

Figure 4.12: Abstract functions for procedure calls.

In figure 4.13, we give the standard function $\mathcal{D}$ and the abstract function $\mathcal{F}$ for non-recursive procedure definitions. The standard semantic function maps an input environment to an output environment in which q is bound to a function Q, where Q maps n input values and a memory, to a memory. If we examine Q more closely, we see that the body of q is evaluated with a local environment and a local store. Also note that the final result is a memory containing the initial *store* and stack pointer (sp), but a possibly updated *heap* and heap pointer (fl).

The abstract semantic function maps an input abstract environment to an output abstract environment in which $q.D$ has been bound to a function D, where D maps n handles and a path matrix, to a path matrix.

```
$\mathcal{D}[\![$ proc $q$ ( $x_1$:$t$ ; ... ; $x_n$:$t$ ) $l_1$:$t$ ; ... ; $l_m$:$t$ ; $s$ $]\!]$ $env$ = $env$ [$q \mapsto Q$]
  where $Q$ =
    $\lambda v_1 . \lambda v_2 . \ldots . \lambda v_n . \lambda\ (store,heap,sp,fl,in,out)$ .
      let
        $local\_env$ =
          $env[\ x_1 \mapsto sp+1, \ldots, x_n \mapsto sp+n,$
                $l_1 \mapsto sp+n+1, \ldots, l_n \mapsto sp+n+m\ ]$
        $local\_store$ =
            $store[local\_env(x_1) \mapsto v_1, \ldots, local\_env(x_n) \mapsto v_n,$
                    $local\_env(l_1) \mapsto \underline{undef}, \ldots, local\_env(l_m) \mapsto \underline{undef}\ ]$
        $new\_mem = \mathcal{M}[\![\ s\ ]\!]\ local\_env\ (local\_store,\ heap,\ sp+m+n,\ fl,\ in,\ out)$
      in
        if $new\_mem = \underline{error}$ then
          $\underline{error}$
        else
          let
            $(local\_store', heap', sp', fl', in', out') = new\_mem$
          in
            $(store, heap', sp, fl', in', out')$

$\mathcal{F}[\![$ proc $q$ ( $x_1$:handle ; ... ; $x_n$:handle )
    $l_1$:handle ; ... ; $l_m$:handle ; $s$ $]\!]$ $aenv$ = $aenv$ [$q.D \mapsto D$]
  where

      $D = \lambda h_1 . \ldots . \lambda h_n . \lambda e . \lambda P$ .
        let
          $(P_{var}, P_{inv}) = split\ P\ [h_1, \ldots, h_n]$
          $P_1 = rename([h_1 \mapsto \underline{x_1}, \ldots, h_n \mapsto \underline{x_n}], P_{var})$
          $P_2 = F\ e\ P_1$
          $P_3 = rename([\underline{x_1} \mapsto h_1, \ldots, \underline{x_n} \mapsto h_n], P_2)$
        in
          $join(P_{inv},P_3)\ [h_1, \ldots, h_n]$

      $F = \lambda e . \lambda P$ .
        let
          $P_1' = insert([x_1 \mapsto \underline{x_1}, \ldots, x_n \mapsto \underline{x_n}, l_1 \mapsto \underline{nil}, \ldots, l_m \mapsto \underline{nil}], P)$
          $P_2' = \mathcal{A}[\![\ s\ ]\!]\ aenv\ P_1'$
        in
          $remove([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2')$
```

Figure 4.13: Semantic functions for non-recursive procedures.

The first step in D splits the input path matrix P into two path matrices P_{inv} and P_{var}. P_{inv} represents the relationships between all handles that cannot reached through paths starting from any of the argument handles, while P_{var} represents the relationships between handles that can be reached via an argument handle. Note that handles that are inaccessible from the arguments cannot have their relationships with other inaccessible handles modified. Thus, in order to get the effect of executing the procedure call on path matrix P, it is only necessary to compute the effect on P_{var}.

The second step in D is to create P_1 from P_{var} by renaming the caller's argument handles to names local to the procedure call[2]. P_2 results from executing the body of the procedure on P_1 (as modelled by the function F), and P_3 is just renaming the local names back to the caller's names. Note that the function F just allocates local copies of the arguments and local variables, executes the body, and deallocates. Finally, the overall result is given by joining P_{inv} and P_3. The joining of P_{inv} and P_3 reconstructs the entire resulting path matrix by connecting all handles in the invariant part with those in the variant part. More specifically, if $a_1 \ldots a_n$ are the argument handles, h_i is a handle in in P_{inv}, and h_v is a handle in the variant part, all paths from h_i to h_v can be constructed by concatenating all paths from h_i to a_j in P_{inv}, with all paths from a_j to h_v in the variant part.

The rule for recursive procedure calls (figure 4.14) uses the same basic strategy as the non-recursive case, but has a more complex rule for F, and a slightly different rule for D. Note that for the recursive case we must define *insert* carefully. If a handle is inserted into a path matrix in which it already exists, the relationships for both cases must be merged. However, it is important to note that this merging is often not required because a reused handle name is often in the invariant part of the input path matrix.

This rather daunting rule for recursive procedures has two novel aspects. First of all, one needs to configure the base case rather carefully. This is done by computing the changes to the path matrix that are produced by all traversals through the procedure body *that do not encounter a further recursive call.* This is the role of the function *base*.

The second fact worth noting is how we approximate functions. As is well known from denotational semantics the meaning of a recursively defined function or procedure is given by the fixed point of a higher-order functional. Thus the approximants to the fixed point live in the function space. Normally in abstract interpretations, strictness for example, these functions are between small finite lattices and hence one can use "tabular" representations or perhaps optimized representations thereof. In our case this is not true since the abstract domain is infinite. We use *step functions* written as pairs to represent a *finite piece of information* about these approximants. More precisely, the environment associates with each recursive procedure r a recursively defined function $r.D$, an estimate written $r.est$ and a boolean flag $r.new\_est$

[2]We assume that all parameter names and local variables are uniquely named (implemented by tagging them with their procedure number).

```
$\mathcal{F}[\![$ **recproc** $r$ ( $x_1{:}t$ ; ... ; $x_n{:}t$ ) $l_1{:}t$ ; ... ; $l_m{:}t$ ; $s$ $]\!]$ $aenv$ =
    $aenv\ [r.D \mapsto D,\ r.new\_est \mapsto true,\ r.est \mapsto (\Omega,\Omega)]$ where
  $D = \lambda h_1 \ldots\ldots \lambda h_n\ .\ \lambda e\ .\ \lambda P\ .$
    let
      $(P_{var}, P_{inv}) = split\ P\ [h_1, \ldots, h_n]$
      $P_1 = rename([h_1 \mapsto \underline{x_1}, \ldots, h_n \mapsto \underline{x_n}], P_{var})$
      $P_2 =$ if $e(r.new\_est)$ then $F\ e[r.est \mapsto (P_1,\ base(P_1))]\ P_1$ else $F\ e\ P_1$
      $P_3 = rename([\underline{x_1} \mapsto h_1, \ldots, \underline{x_n} \mapsto h_n], P_2)$
    in
      $join(P_{inv}, P_3)\ [h_1, \ldots, h_n]$
  $F = \lambda e\ .\ \lambda P\ .$
    let $(P_{in}, P_{out}) = e(r.est)$ in
      if $e(r.new\_est)$ then
        { ---- CASE 1 (generalize output?) ---- }
        let
          $P_1 = insert([x_1 \mapsto \underline{x_1}, \ldots, x_n \mapsto \underline{x_n},$
                        $l_1 \mapsto \underline{nil}, \ldots, l_m \mapsto \underline{nil}], P)$
          $P_2 = \mathcal{A}[\![\ s\ ]\!]\ e[r.new\_est \mapsto false]\ \ P_1$
          $NP_{out} = remove([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2)$
        in
          if $NP_{out} \sqsubseteq P_{out}$ then
            $P_{out}$  { -- 1(a) -- }
          else
            $F\ e[r.est \mapsto (P, (NP_{out} \bowtie P_{out}))]\ P$ { -- 1(b) -- }
      else
        { ---- CASE 2 (generalize input?) ---- }
        if $P \sqsubseteq P_{in}$ then
          $P_{out}$  { -- 2(a) -- }
        else
          let
            $NP_{in} = (P \bowtie P_{in})$
          in { -- 2(b) -- }
            $F\ e[r.new\_est \mapsto true,\ r.est \mapsto (NP_{in},\ base(NP_{in}))]\ NP_{in}$
  $base = \lambda P\ .$
    let
      $P_1 = insert([x_1 \mapsto \underline{x_1}, \ldots, x_n \mapsto \underline{x_n}, l_1 \mapsto \underline{nil}, \ldots, l_m \mapsto \underline{nil}], P)$
      $P_2 = \mathcal{A}[\![\ s \ominus r\ ]\!]\ aenv\ P_1$
    in
      $remove([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2)$
```

Figure 4.14: Abstract semantic function for recursive procedures.

that will be explained below. The estimate is a pair of path matrices (P_{in},P_{out}). Such a pair says that if the input path matrix to the procedure call is less general than P_{in} then the output path matrix is less general than P_{out}, in other words, P_{out} can be used safely as the output.

One can now operationally understand the working of the rule above. When procedure r is defined it is initialized with a completely uninformative estimate *r.est*, written (Ω, Ω), and the flag *r.new_est* set to true. The first time r is called, *r.est* is initialized with an estimate computed by *base*. The function *base* is defined to compute the effect using the nonrecursive paths through the body (s), this is represented by $s \ominus r$. This may be implemented by constructing a pruned parse tree that contains only the sub-trees that do not contain a recursive call, or by propagating a special undefined path matrix through all paths encountering a recursive call.

The first part of the conditional (case 1) in F is called when a fresh computation begins (*r.new_est* = *true*). If the recursive invocations do not cause the new output path matrix NP_{out} to generalize beyond the stored estimate P_{out}, one need not invoke the abstract function further; but if it does, then the estimate must be iteratively refined by invoking F again with the output estimate suitably generalized. Superficially this may seem to be all that is needed. Unfortunately our estimates are only valid for *some* inputs, i.e. those that are specializations of the input path matrix at the original point of call. Thus we cannot use our estimates at any point where a recursive call is made. One must merge together all the input path matrices at all points of call or at least at all points along a given chain of calls. The final conditional (case 2) in the definition of F is designed to handle this. If the current input is not less than the estimated input then the input is generalized and the successive approximations to the output are recalculated. The boolean flag basically tells us whether we are generalizing the input or the output.

The proof that F terminates follows from the fact that both the input and the output component are being generalized, at least one of them is generalized at each invocation of F, and the fact that the ascending chains have finite height.

A less precise (but more efficient) fixed point estimation has also been implemented. The major difference lies in the treatment of the case when P is not contained in the stored estimate P_{in} (case 2(b)). In the previous fixed-point estimate (figure 4.14), a new invocation of F(*new_est* = *true*) is started each time the input needs to be generalized. A more efficient method is to terminate the current estimation of F and restart it with the generalized input[3]. With the more efficient fixed-point analysis, we can give a reasonable bound on the number of iterative approximations. As with the while loop case, the number of possible generalizations of the input path matrix is $O(mn^2)$, where m is the length of the path expressions and n is the number of handles. For each input generalization, there are also $O(mn^2)$ possible output generalizations, giving a total number of generalizations

[3]This is easily implemented with the ML exception mechanism

of $O(m^2n^4)$. It should be noted that this is an upper-bound, and since not all handles are affected, and those that are affected tend to be related, the number of generalizations is usually quite small.

4.4 Soundness Theorem

In order to establish the soundness of the estimate one needs a relationship between the abstract values (path matrices), and the state of the memory, as defined by the standard denotational semantics. Recall that a path matrix expresses connectivity relations between live handles, which are identifiers. Thus in order to state the desired safety relationship one needs to include the environment as well as the memory. The following definitions relate a memory and environment pair in the standard semantics to path matrices in the abstract semantics.

Definition 13. For a heap H, and heap pointers h_i and h_j, $H[h_i \rightsquigarrow h_j]$ denotes the set of paths connecting h_i to h_j in H. That is, $\forall r \in (L+R)^*$. if r connects h_i and h_j in H, then $r \in H[h_i \rightsquigarrow h_j]$.

Definition 14. Suppose that $X = \langle (d_1, \ldots, d_m), (c_1?, \ldots, c_n?) \rangle$ is a list of path expressions, H is a heap, and h_i and h_j are heap pointers. We say that X *safely estimates* the paths between h_i and h_j in H, written $X \trianglelefteq H[h_i \rightsquigarrow h_j]$, if:

1. $\forall r \in H[h_i \rightsquigarrow h_j]\,.\,(\exists i \in \{1 \ldots m\}.r \in d_i)$ or $(\exists i \in \{1 \ldots n\}.r \in c_i)$;
2. $\forall i \in \{1 \ldots m\}\,.\,\exists r \in d_i\,.\,r \in H[h_i \rightsquigarrow h_j]$.

The first condition states that the set denoted by X must contain all paths actually in the heap. The second condition states that each definite path expression d_i must denote a set that contains at least one path r such that r is a path in the heap.

Definition 15. Suppose that P is a path matrix, M is a memory with a heap H, and *env* is an environment. Suppose also that $D = (d_1, \ldots, d_m)$ lists all the heap pointers to *DAG* nodes in H, and that $L = (l_1, \ldots, l_n)$ lists all the non-nil, live handle variables in *env*. We say that P *safely estimates* (M, env), written $P \triangleleft (M, env)$ if:

(I) P is defined for all handles in L;

(II) There exists dag handles $X = (x_1, \ldots, x_m)$ in P that can be associated item-wise with $D = (d_1, \ldots, d_m)$ in H such that:

1. $\forall l_i, l_j \in L \,.\, P[l_i, l_j] \trianglelefteq H[store(env(l_i)) \rightsquigarrow store(env(l_j))]$
2. $\forall l_i \in L, d_j \in D \,.\, P[l_i, x_j] \trianglelefteq H[store(env(l_i)) \rightsquigarrow d_j]$
3. $\forall d_i \in D, l_j \in L \,.\, P[x_i, l_j] \trianglelefteq H[d_i \rightsquigarrow store(env(l_j))]$
4. $\forall d_i, d_j \in D \,.\, P[x_i, x_j] \trianglelefteq H[d_i \rightsquigarrow d_j]$
5. $\forall d_i \in D \,.\, RefCnt(d_i) \leq P.RefCnt[x_i]$
6. $\forall l_i \in L \,.\, Nilness(env(l_i)) \leq P.Nilness[l_i]$.

Condition I states that P must contain the relationships for all live handles in *env*. Condition II (1-4) states that there must be a list of dag handles x_i in P such that each x_i can be associated with a d_i in a manner such that P safely estimates all the relationships between all live handles l_j and dag nodes d_k in M. Note that not all the x_i's need to be distinct. We can use one dag handle in P to approximate the relationships for many dag nodes in M. Conditions II (5 and 6) state that the reference count estimates and nilness estimates in P must be conservative approximations of the nilness and reference count in (M, env).

The link between the definition of safety and the ordering on path estimates and path matrices is given by the following lemmas.

Lemma 1. If $X \trianglelefteq H[h_i \rightsquigarrow h_j]$ and $X \sqsubseteq X'$, then $X' \trianglelefteq H[h_i \rightsquigarrow h_j]$.

Proof: Suppose that r is a path in $H[h_i \rightsquigarrow h_j]$. Suppose also that $X = \langle (d_1, \ldots, d_k), (e_1?, \ldots, e_l?) \rangle$, and $X' = \langle (f_1, \ldots, f_m), (g_1?, \ldots, g_n?) \rangle$. Because $X \trianglelefteq H[h_i \rightsquigarrow h_j]$ holds we know that $r \in (\bigcup_i d_i) \cup (\bigcup_j e_j)$. The first condition in definition 11 state that $((\bigcup_i d_i) \cup (\bigcup_j e_j)) \subseteq ((\bigcup_i f_i) \cup (\bigcup_j g_j))$, thus $r \in ((\bigcup_i f_i) \cup (\bigcup_j g_j))$. Consider any f_i, by the second condition in the definition of $\sqsubseteq$ there is some d_j with $d_j \subseteq f_i$. By the second condition in the definition of $\trianglelefteq$, we know that there is a path $t \in d_j$ that connects h_i to h_j in H. Thus we have $t \in f_i$ satisfying the second required condition for $X' \trianglelefteq H[h_i, h_j]$ to hold. ∎

Lemma 2. If $P \triangleleft (M, env)$ and $P \sqsubseteq P'$ then $P' \triangleleft (M, env)$.

Proof: Suppose P, P', M and *env* are as in the statement. Since P and P' must be defined on the same handles, condition I holds for $P' \triangleleft (M, env)$. For condition *II*, we choose the same handles X for P' as were used for the safety of P. By condition 1 in definition 2 we know that $\forall h_i, h_j \in (L \cup X), P[h_i, h_j] \sqsubseteq P'[h_i, h_j]$. Thus, using lemma 1 we know that the $\trianglelefteq$ conditions *II(1-4)* hold for all entries $P[h_i, h_j]$

and $P'[h_i, h_j]$. Finally, conditions *II(5-6)* hold because the nilness and reference count estimates in P' must be at least as general as those is P. ∎

In discussing soundness we need to look at how statements are interpreted in the two semantics. In the standard semantics they are interpreted as partial functions from memories to memories, we call such functions *memory transformers*, whereas in the abstract semantics they are interpreted as (total) functions from path matrices to path matrices. We call the latter *path-matrix transformers*. The notion of safety extends naturally to memory transformers and path matrix transformers. We will use the symbol $\lhd$ for this also.

Definition 16. Suppose that T is a map between memories and that R is a map between path matrices. We say that R *safely estimates* T if, for any choice of path matrix P, memory M and environment *env* satisfying $P \lhd (M, env)$ we have $RP \lhd (TM, env)$ provided TM is defined.

Similarly we can define safety between abstract and real environments.

Definition 17. Suppose that *aenv* is an abstract environment and *env* is an environment. Suppose also that q is a procedure name defined in the environment *env*. Then we say that $aenv \lhd env$ if:

1. q is also defined in aenv;
2. if $h_1, \ldots, h_k$ are live handles in *env* and P and M satisfy $P \lhd (M, env)$ then $[aenv(q)h_1 \ldots h_k P] \lhd ([env(q)(env(h_1)) \ldots (env(h_k))M], env)$.

A key property of the safety relation on transformers is that it is $\omega - inclusive$[4]. The memory transformers are partial functions and are ordered by inclusion of their graphs.

Lemma 3. Suppose that $\{T_i\}_{i \in N}$ is an increasing sequence of memory transformers and that R is a path-matrix transformer. If $\forall i. R \lhd T_i$ then $R \lhd \sqcup T_i$. If R_i is an increasing sequence of path matrix transformers and T is a memory transformer such that $\forall i. R_i \lhd T$ then $\sqcup R_i \lhd T$.

Proof: Suppose that P, M and *env* satisfy $P \lhd (M, env)$. Suppose that $\sqcup T_i$ is defined on M. Since the least upper bound of chains is given by union, then there is some k such that T_k is defined on M and, in fact, $(\sqcup T_i)M = T_k M$. Since R is safe for T_k we immediately have $RP \lhd (T_k M, env)$. The proof of the second statement follows immediately from the fact that there can be only finite ascending chains of path matrices and from lemma 2. ∎

We are now ready to state and prove the main soundness theorem.

[4]We use this term to mean that a relation, viewed as a function to the domain of booleans is continuous.

Theorem 1. For any statement s and any pair of environments env and $aenv$ such that $aenv \lhd env$

$$\mathcal{A}[\![s]\!]\ aenv \lhd (\mathcal{M}[\![s]\!]\ env\ , env).$$

Proof: The proof proceeds by structural induction on statements. The soundness of the basic handle analysis rules has already been discussed informally in chapter 3. The main point of this proof is to use the machinery for the fixed-point semantics to show that the soundness of the abstract semantic functions for while loops and recursive procedures. This part of the proof is quite independent of the specific analysis rules for the basic statements. We illustrate one simple base case and then discuss the case of while loops, which requires a simple fixed-point induction, nonrecursive procedures and finally recursive procedures.

Simple cases

These cases have all been discussed informally in chapter 3, we treat one case more formally to illustrate the style of the proof.

$$\mathcal{A}[\![A := B]\!]\ aenv\ P = P'$$

where

$$\forall handles\ h_i, h_j.\ P'[h_i, h_j] \mapsto \begin{cases} \{S\} & h_i = A, h_j = B \\ P[B, h_j] & h_i = A\ h_j \neq A, B \\ \{S\} & h_i = B, h_j = A \\ P[h_i, B] & h_i \neq A, B, h_j = A \\ P[h_i, h_j] & h_i, h_j \neq A \end{cases}$$

According to the standard semantics, $store(env(A)) = store(env(B))$ after execution of the above statement. Thus the relations between A and any live handle, X, other than A or B should be exactly the same as that of B and X. The relations between live handles, neither of which are A, are clearly unaffected. Finally the entries for A and B must now say that they are the same, as the entry $\{S\}$ signifies.

While loops

Here we need to unroll the two fixed-point definitions given in figure 4.11 and compare them "stagewise" (figure 4.15). Recall that the standard semantics for the while loop is given by the fixed-point of a functional, W, from memory transformers to memory transformers. We write $W^{(k)}$ for the kth approximant to $fix(W)$. Thus $W^{(0)}$ is everywhere undefined and $W^{(1)}$ is the approximant that is defined on those memories for which the while loop terminates immediately because the condition is false. Similarly, the abstract semantics is defined through a sequence of approximants, the main difference being that rather than taking the least upper bound (which may not be defined) of the path expressions we use the merge operator, $\bowtie$, to determine an upper bound. We call the successive stages in the computation of $\mathcal{A}[\![$**while** e **do** $s]\!]$, $A^{(k)}$ in analogy with the approximants to the while loop.

```
$W^{(0)} = \lambda m \ . \ \bot$
$W^{(1)} = \lambda m \ . \ \underline{\text{if}}\ \mathcal{E}[\![ e ]\!]\ env\ m\ \underline{\text{then}}\ \bot\ \underline{\text{else}}\ m$
$W^{(k+2)} = \lambda m \ . \ \underline{\text{if}}\ \mathcal{E}[\![ e ]\!]\ env\ m\ \underline{\text{then}}\ W^{(k+1)}\ (\ \mathcal{M}[\![ s ]\!]\ env\ m\ )\ \underline{\text{else}}\ m$

$A^{(0)} = \lambda p \ . \ p$
$A^{(k+1)} = \lambda p \ .$
    let
      $new\_p = (\mathcal{A}[\![ s ]\!]\ aenv\ p) \bowtie p$
    in
      if $new\_p = p$ then $p$ else $A^{(k)}\ new\_p$
```

Figure 4.15: Unrolling the semantic functions for **while**.

Note that $A^{(0)}$ represents the effects of zero traversals of the loop body and hence corresponds to $W^{(1)}$ rather than to $W^{(0)}$.

First we show that, for any positive integer k, $A^{(k)} \lhd W^{(k+1)}$. This proceeds by induction on k. The base case, $k = 0$, is immediate from the definitions. For the inductive case we assume that

$$A^{(k)}P \lhd (W^{(k+1)}M, env)$$

for all M, P and env that satisfy $P \lhd (M, env)$ and $W^{(k-1)}M \neq \bot$ and prove that

$$A^{(k+1)}P \lhd (W^{(k+2)}M, env).$$

Thus, we have both a fixed-point induction hypothesis (FIH), and a structural induction hypothesis (SIH) as below:

$$(FIH) : A^{(k)}P \lhd (W^{(k+1)}M, env).$$

$$(SIH) : \mathcal{A}[\![s]\!]\ aenvP \lhd (\mathcal{M}[\![s]\!]\ env\ M), env)$$

We will consider two cases, $\mathcal{E}[\![e]\!]\ env\ M$ = false, and $\mathcal{E}[\![e]\!]\ env\ M$ = true.

Suppose that $\mathcal{E}[\![e]\!]\ env\ M$ is $false$.
In this case we have $W^{(k+2)}M = M$. Since P is one of the path matrices merged to give $A^{(k+1)}P$, we know that $P \sqsubseteq A^{(k+1)}P$. Therefore, $A^{(k+1)}P \lhd (M, env)$, which can be restated as $A^{(k+1)}P \lhd (W^{(k+2)}M, env)$.

Now suppose that $\mathcal{E}[\![e]\!]\ env\ m$ is $true$.
By SIH we have $\mathcal{A}[\![s]\!]\ aenv\ P \lhd (\mathcal{M}[\![s]\!]\ env\ M, env)$.
Consider new_p in the body of $A^{(k+1)}$. Since new_p is the result of $P \bowtie (\mathcal{A}[\![s]\!]\ aenv\ P)$, we know that $\mathcal{A}[\![s]\!]\ aenv\ P \sqsubseteq new\_p$, and therefore $new\_p \lhd (\mathcal{M}[\![s]\!]\ env\ M, env)$.

Now let us consider the result of $A^{(k+1)}P$. If $new\_p = P$, then the result of $A^{(k+1)}P$ is $P = A^{(k)}P = A^{(k)}new\_p$.
If $new\_p \neq P$, then the result of $A^{(k+1)}P$ is also $A^{(k)}new\_p$.
Applying FIH to $new\_p \lhd (\mathcal{M}[\![s]\!]\ env\ M, env)$ gives

$$A^{(k)}new\_p \lhd (W^{(k+1)}(\mathcal{M}[\![s]\!]\ env\ M, env),$$

which is the same as the desired result $A^{(k+1)}P \lhd (W^{(k+2)}M, env)$.

Since the abstract domain is of finite height we conclude that the sequence of approximations, $A^{(k)}$ is finite. Thus we need to show that the last one of these, call it A, safely estimates $fix(W)$. We know by the last paragraph that A safely estimates all the $W^{(k)}$, the desired result now follows from lemma 3.

Nonrecursive Procedures

The next case is nonrecursive procedures. Here we wish to prove that the operations on the environments, both abstract and standard, preserve safety. From this the soundness of the rule for procedure calls follows directly from the structural induction hypothesis. The new ingredient that we need to consider is the role of the abstract environment and the fact that the environment gets modified when the procedure body is executed.

In order to examine the procedure definition case, we must first examine the auxiliary functions *split* and *join*. Note that when a procedure call is made we do not pass the entire path matrix to the procedure body but only that portion of it that contains handles whose relationships may be affected. The relevant operations are carried out by the auxiliary functions *split* and *join*. We need the following condition on *split* that follows from its definition.

Condition 3. Suppose that $(P_V, P_I) = split(P, \chi)$ then for X a handle in P_I and $Y \in \chi$ we have $P[Y, X] = \{\}\ or\ \{S\}$.

We also require the following lemma for the safety of path matrix transformers that use split and join.

Lemma 4. Suppose that we have P, M and env with $P \lhd (M, env)$. Suppose that T_A is a path matrix transformer and T_M is a memory transformer with $T_A \lhd T_M$. Suppose that all the handles accessed by T_A are included in a set of handles χ. We define a new path matrix transformer T'_A by the following equation

$$T'_A = \lambda P. \text{let } (P_V, P_I) = split(P, \chi) \text{ in } join(P_I, T_A(P_V), \chi).$$

Then $T'_A \lhd T_M$.

Proof: Suppose that X and Y are two handles that are in P_I. First we note that T_M cannot affect a handle outside χ since $P \lhd (M, env)$. Second, we claim no path from X to Y can be affected by T_M since there are no paths from the handles that

T_M can affect to *any node on a path between* X *and* Y. Suppose there was some node Z on a path from X to Y such that Z is below one of the handles in χ. This would mean that Y is below some node in χ which is impossible. Thus T'_A gives the right relationships between pairs of handles both of which are in χ or neither of which are in χ. If we have a handle, X, above χ and a handle Y below χ then, because we are looking at tree structures, we can see that a path from X to Y must go through χ. Thus *join* gives a safe estimate in this case also. ∎

Note that in procedure calls we know precisely which handles the procedure body has access to so the situation described in the lemma applies. In view of this fact we can ignore further discussion of the role of *split* and *join* in the rest of the proof.

In demonstrating soundness for definitions we need to compare the effects of $\mathcal{F}$ and $\mathcal{D}$ rather than $\mathcal{A}$ and $\mathcal{M}$. We recall the semantic functions for a procedure definition for both the standard and abstract semantics (figure 4.13). We need to show that D is safe for Q when they are applied to the appropriate arguments. We assume, therefore, that $env(h_i) = v_i$ and strip out the n leading lambda abstractions in Q and D. We see that the abstract function D performs three actions. First, it splits off the portion of the path matrix that cannot be altered. Because the procedure call does not have handles to this invariant part, we know that this is safe because of lemma 4. The next action is to rename the caller's handles to special names local to the procedure body (P_1), and then add new entries to the path matrix to represent the parameters and the local variables (P'_1). Clearly the effect of building P'_1 in F is precisely analogous to the construction of *local_env* and *local_store* in Q. Note that the role of *rename* is to ensure that there is an extra copy of the parameters so that when the procedure returns the relationships that originally held between the parameters is unaltered. Thus we know that $P'_1 \lhd ((local\_store, \ldots), local\_env)$. Now the structural induction hypothesis applied to s gives us that $P'_2 \lhd ((local\_store', heap', \ldots), local\_env)$. Finally the effect of returning *store* in the standard semantics and restoring the original environment is mimicked by the renaming of P_2 to P_3 in D. Thus P_3 is a safe result.

Recursive procedures

The manipulations of the environment and the abstract environment are very similar to the non-recursive case so we will not discuss them except when they differ. Our main point in this case is to show safety by fixed-point induction. We compare the function D (figure 4.14) in the abstract semantics with Q (figure 4.16) in the standard semantics and will ignore *rename*, *split* and other things that we know to be safe.

One subtle aspect of the abstract semantics is that the environment carries a stored estimate (P_{in}, P_{out}) of the desired path matrix transformer. We need to carry in our induction hypothesis the fact that P_{out} gives a safe output if the input is less than P_{in}. The proof involves relating the unwindings of D, and the estimates stored in the environment, to the standard function Q. The proof relates D to Q by fixed-point induction. Since safety is ω-inclusive this amounts to relating $D^{(k)}$ to $Q^{(k)}$, where each of these denote the kth approximant in the unwinding of the recursive definitions.

We begin with a discussion of the unwindings of the recursive function Q in the standard semantics, and the recursive functions F and D in the abstract semantics. The meaning of a recursively defined procedure in the standard semantics is shown in figure 4.16. Let $\mathcal{R}$ stand for fix(R). Using the fixed-point rule on $\mathcal{R}(env)$ gives

$$(\lambda T.\lambda e.e[r \mapsto Q])(\mathcal{R})env$$

and carrying out the indicated β-reductions gives

$$env[r \mapsto (Q[T \mapsto \mathcal{R}])].$$

Using these "unwound" expressions we see that the approximants to $\mathcal{R}$ and Q are as shown in figure 4.17.

```
$\mathcal{D}[\![$ recproc $r$ ( $x_1$:$t$ ; ... ; $x_n$:$t$ ) $l_1$:$t$ ; ... ; $l_m$:$t$ ; $s$ $]\!]$ $env$ = $fix(R)$ $env$
  where
    $R = \lambda T \,.\, \lambda e \,.\, e[r \mapsto Q]$
    $Q = \lambda v_1 \,.\, \lambda v_2 \,.\, \ldots \,.\, \lambda v_n \,.\, \lambda\ (store,heap,sp,fl,in,out)$ .
      let
        $local\_env$ =
          $(T\ e)[\ x_1 \mapsto sp+1,\ \ldots\ ,\ x_n \mapsto sp+n,$
                $l_1 \mapsto sp+n+1,\ \ldots\ ,\ l_m \mapsto sp+n+m\ ]$
        $local\_store$ =
          $store[local\_env(x_1) \mapsto v_1,\ \ldots,\ local\_env(x_n) \mapsto v_n,$
                $local\_env(l_1) \mapsto \underline{undef},\ \ldots,\ local\_env(l_m) \mapsto \underline{undef}\,]$
        $new\_mem$ =
          $\mathcal{M}[\![\ s\ ]\!]\ local\_env\ (local\_store,\ heap,\ sp+n+m,\ fl,\ in,\ out)$
      in
        if $new\_mem = \underline{error}$ then
          $\underline{error}$
        else
          let
          $(local\_store',heap',sp',fl',in',out') = new\_mem$
          in
            $(store,heap',sp,fl',in',out')$
```

Figure 4.16: Recursive procedures in the Standard Semantics.

Similarly, we can define the unwindings of the functions in the abstract semantics. We define a sequence of $F^{(k)}$s and $D^{(k)}$s as in figure 4.18.

```
$\mathcal{R}^{k+1}(env) = env[r \mapsto Q^{(k)}]$
  $Q^{(k)} = \lambda v_1 \ . \ \lambda v_2 \ . \ \ldots \ . \ \lambda v_n \ . \ \lambda \ (store,heap,sp,fl,in,out)$ .
      let
        $local\_env =$
          $(\mathcal{R}^{(k)} \ env)[\ x_1 \mapsto sp+1, \ \ldots \ , \ x_n \mapsto sp+n,$
                  $l_1 \mapsto sp+n+1, \ \ldots \ , \ l_m \mapsto sp+n+m \ ]$
        $local\_store =$
          $store[local\_env(x_1) \mapsto v_1, \ \ldots, \ local\_env(x_n) \mapsto v_n,$
                  $local\_env(l_1) \mapsto \underline{undef}, \ \ldots, \ local\_env(l_m) \mapsto \underline{undef}\,]$
        $new\_mem =$
          $\mathcal{M}[\![ \ s \ ]\!] \ local\_env \ (local\_store, \ heap, \ sp+n+m, \ fl, \ in, \ out)$
      in
        if $new\_mem = \underline{error}$ then $\underline{error}$ else
          let
          $(local\_store', heap', sp', fl', in', out') = new\_mem$
          in
            $(store, heap', sp, fl', in', out')$
```

Figure 4.17: The k'th unwinding of $\mathcal{R}$ and Q.

The base case case of the fixed-point induction, $k = 1$, is straightforward. The inductive case is rather more complicated. In this case, we need to examine the unwindings of the auxiliary function F. This function, however, combines two quite different actions; it refines the estimate stored in the environment both by making the input path matrix more general and by making the output estimate more general. These interact in a fairly complicated way and it is not easy to relate a given $F^{(k)}$ with some number of refinements of each kind. Accordingly, we introduce a labeled family of approximants to F written $F^{p,m}_n$ where m indicates the number of times the output estimates may be refined, and n indicates how many times the input estimate may be refined. We use these approximants to argue that the estimators are being refined appropriately and, finally, we argue that the $F^{p,m}_n$ have the same least upper bound as the $F^{(k)}$, so that one can safely use $F^{p,m}_n$ rather than $F^{(k)}$ in the fixed-point induction. The rest of this subsection spells out the details of this proof.

First we discuss the base case and the associated auxiliary functions in the abstract semantics. We display the first unwinding $F^{(1)}$ and $D^{(1)}$ in figure 4.19. We introduce the symbol Ω to stand for a hypothetical "least general path matrix". From the information point of view it represents inconsistency; in terms of the ordering $\sqsubseteq$ it is bottom and hence serves as the starting point of recursive unfoldings. The definition of the abstract semantics does not involve Ω so it will not appear in any estimate, it merely serves as a convenient way to denote the partial unwindings

```
$D^{(k)} = \lambda h_1 \,.\, \ldots \,.\, \lambda h_n \,.\, \lambda e \,.\, \lambda P \,.$
  $\underline{\text{let}}$
    $(P_{var}, P_{inv}) = split\ P\ [h_1, \ldots, h_n]$
    $P_1 = rename([h_1 \mapsto \underline{x_1}, \ldots, h_n \mapsto \underline{x_n}], P_{var})$
    $P_2 = \underline{\text{if}}\ e(r.new\_est)\ \underline{\text{then}}$
                $F^{(k)}\ e[r.est \mapsto (P_1, base(P_1))]\ P_1$
              $\underline{\text{else}}\ F^{(k)}\ e\ P_1$
    $P_3 = rename([\underline{x_1} \mapsto h_1, \ldots, \underline{x_n} \mapsto h_n], P_2)$
  $\underline{\text{in}}$
    $join(P_{inv}, P_3)\ [h_1, \ldots, h_n]$
$F^{(k+1)} = \lambda e \,.\, \lambda P \,.$
  $\underline{\text{let}}\ (P_{in}, P_{out}) = e(r.est)\ \underline{\text{in}}$
    $\underline{\text{if}}\ e(r.new\_est)\ \underline{\text{then}}$
      { ---- CASE 1 (generalize output?) ---- }
      $\underline{\text{let}}$
        $P_1 = insert([x_1 \mapsto \underline{x_1}, \ldots, x_n \mapsto \underline{x_n},$
                        $l_1 \mapsto \underline{nil}, \ldots, l_m \mapsto \underline{nil}], P)$
        $P_2 = \mathcal{A}[\![\ s\ ]\!]\ e[r.new\_est \mapsto false, r.D \mapsto D^{(k)}]\ \ P_1$
        $NP_{out} = remove([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2)$
      $\underline{\text{in}}$
        $\underline{\text{if}}\ NP_{out} \sqsubseteq P_{out}\ \underline{\text{then}}$
          $P_{out}$   { -- 1(a) -- }
        $\underline{\text{else}}$
          $F^{(k)}\ e[r.est \mapsto (P, (NP_{out} \bowtie P_{out}))]\ P$ { -- 1(b) -- }
    $\underline{\text{else}}$
      { ---- CASE 2 (generalize input?) ---- }
      $\underline{\text{if}}\ P \sqsubseteq P_{in}\ \underline{\text{then}}$
        $P_{out}$   { -- 2(a) -- }
      $\underline{\text{else}}$
        $\underline{\text{let}}$
          $NP_{in} = (P \bowtie P_{in})$
        $\underline{\text{in}}$ { -- 2(b) -- }
          $F^{(k)}\ e[r.new\_est \mapsto true, r.est \mapsto (NP_{in}, base(NP_{in}))]\ NP_{in}$
```

Figure 4.18: The k'th unwinding of F.

```
$D^{(1)} = \lambda h_1 \; . \; \ldots \; . \; \lambda h_n \; . \; \lambda e \; . \; \lambda P \; .$
  let
    $(P_{var}, P_{inv}) = split\ P\ [h_1, \ldots, h_n]$
    $P_1 = rename([h_1 \mapsto \underline{x_1}, \ldots, h_n \mapsto \underline{x_n}], P_{var})$
    $P_2 =$ if $e(r.new\_est)$ then
                 $F^{(1)}\ e[r.est \mapsto (P_1, base(P_1))]\ P_1$
              else $F^{(1)}\ e\ P_1$
    $P_3 = rename([\underline{x_1} \mapsto h_1, \ldots, \underline{x_n} \mapsto h_n], P_2)$
  in
    $join(P_{inv}, P_3)\ [h_1, \ldots, h_n]$
$F^{(1)} = \lambda e \; . \; \lambda P \; .$
  let $(P_{in}, P_{out}) = e(r.est)$ in
    if $e(r.new\_est)$ then
      { ---- CASE 1 (generalize output?) ---- }
      let
        $P_1 = insert([x_1 \mapsto \underline{x_1}, \ldots, x_n \mapsto \underline{x_n},$
                        $l_1 \mapsto \underline{nil}, \ldots, l_m \mapsto \underline{nil}], P)$
        $P_2 = \mathcal{A}[\![\ s\ ]\!]\ e[r.new\_est \mapsto false, r.D \mapsto D^{(0)}]\ \ P_1$
        $NP_{out} = remove([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2)$
      in
        if $NP_{out} \sqsubseteq P_{out}$ then
          $P_{out}$  { -- 1(a) -- }
        else
          $\Omega$  { -- 1(b) -- }
    else
      { ---- CASE 2 (generalize input?) ---- }
      if $P \sqsubseteq P_{in}$ then
        $P_{out}$  { -- 2(a) -- }
      else
        $\Omega$  { -- 2(b) -- }
```

Figure 4.19: The first unwinding of F, $F^{(1)}$.

of the recursive definition. The termination of $\mathcal{A}$ guarantees that it is everywhere defined. The function *base* estimates the effect of a procedure call on all computation paths that *do not involve a recursive call.* The construct $s \ominus r$ represents the body of the procedure with all calls to r replaced with a new syntactic construct ω, a path matrix transformer that maps any path matrix to Ω. In compound statements it is handled as follows:

1. $s; \omega = \omega; s = \omega$,
2. **if** b **then** s else $\omega = s$, and similarly for the else case,
3. **if** b **then** ω else $\omega = \omega$,
4. **while** b **do** $\omega =$ *skip*.
5. **repeat** ω **until** $e = \omega$.

Note that the while loop yields *skip*, since the body may never be executed, whereas the repeat loop yields ω.
Recall that $Q^{(1)}$ is the function that defines the effect of the procedure call provided there are no recursive calls. If we use $F^{(1)}$ instead of F in D we get $D^{(1)}$. We claim that $D^{(1)}$ applied to *aenv* gives a safe path matrix transformer for the first non-trivial unwinding, $Q^{(1)}$, of the recursive definition in the standard semantics. To see this note that $Q^{(1)}$ is a memory transformer that diverges if any subsequent call to r is made but otherwise terminates. In other words $Q^{(1)}$ terminates exactly when $D^{(1)}$ returns a non-Ω result. The former, when it converges, results from applying $\mathcal{M}[\![s \ominus r]\!]$ to a suitable local environment and local store whereas the latter, when it is not Ω, results from applying $\mathcal{A}[\![s \ominus r]\!]$ to the corresponding path matrix. Thus, by the structural induction hypothesis $D^{(1)}$ is safe for $Q^{(1)}$.

The property of F that we need to prove by induction is somewhat delicate. We need to say not only that F returns safe results but that the estimates "stored in the environment by F" are safe. The proof proceeds in two stages. First we show, by fixed-point induction, that the estimates are refined correctly, then we establish, again by fixed-point induction, that the function F computes a safe path matrix. The latter step uses the former.

The way F is defined, it performs modifications of the estimates in the environment in two ways. First, if the input path matrix to F is not less general than the estimated input, the computation is restarted with generalized input and the base case as the output estimate. Second, if the output produced is not less general than the estimated output, then the output estimate must be redefined. In order to carry out this fixed-point induction perspicuously we need to define a triply-indexed family of approximants to F. We do this by using a labeling technique similar to, but much simpler than, that used in the analysis of the λ-calculus [Bar84]. Note that the definition of F contains two recursive invocations of F. We define $F_0^{0,0}$ to be the function that just returns Ω, the totally undefined path matrix. We show the inductive definition of $F_n^{p,m}$, and D_n^m in figure 4.20. We define $F_n^{p,m}$ by replacing, in the definition of F, the first recursive invocation by $F_n^{p-1,m}$ and the second by $F_{n-1}^{m,m}$. In the computation of P_2 we use an environment that has $r.D$ bound to D_n^{m-1}. We define D_n^m analogously to D using $F_n^{m,m}$ instead of F. Thus, we are defining the labeled expressions $F_n^{p,m}$ and D_n^m together. Intuitively, $F_n^{p,m}$ represents the approximant to F obtained by permitting n refinements of the input estimate and for each such refinement permitting m refinements of the output estimate.

In order to discuss the correspondence between the standard semantics and the abstract semantics we need a convenient way to refer to the portions of the unwound terms. In general, the approximants to a given recursively defined function can be expressed as labelled terms. Suppose that H is some recursively defined function. The partial unwindings of the recursive definition of H are typically denoted by $H^{(n)}$ as we have been doing with Q and D. These partial unwindings are approximants to H, where a given approximant, $H^{(n)}$, can be obtained by substituting $H^{(n-1)}$, for occurrences of H, in the recursive definition of H. In calculating the effect of $H^{(n)}$ on an argument we encounter calls to $H^{(n-1)}$, $H^{(n-2)}$, an so on. We call the collection of such lower approximants encountered during the evaluation of $H^{(n)}a$, where a is the argument to $H^{(n)}$, the *call tree* of $H^{(n)}a$. One can use such operational notions to discuss the fixed-point approximation process in view of the well known correspondence between fixed-points of recursive functionals and computation by substitution into recursive definitions [Man74]. In the following discussion, we will establish properties of the indexed approximants to F by referring to the call trees that result when the $F_n^{p,m}$ are applied to path matrices.

Lemma 5. Suppose that P is a path matrix, M a memory and env an environment with $P \lhd (M, env)$. Suppose that $aenv$ is an abstract environment such that $aenv \lhd env$ and that $aenv$ has Ω as the input estimate for calculating the effects of r. Consider the call tree of the expression $F_n^{p,m}\ aenv\ P$ and suppose that it contains a term of the form $F_0^{i,m}\ aenv'\ P'$. Let the input estimate stored in $aenv'$ be P_1. Now consider the corresponding call tree of $\mathcal{R}^{(n)}\ env$ arising as the nth approximant to $\mathcal{M}[\![r(a_1,\ldots,a_n)]\!]envM$. Consider any memory M' that occurs in the call tree in a term of the form $Q^{(k)}v_1\ldots v_nM'$ where $k \leq n$. Let env' be the environment at this point in the call tree. For all such (M', env') we have $P_1 \lhd (M', env')$.

Proof: Note that the statement is independent of i and m. The base case is obvious since we are comparing an undefined memory to an undefined path matrix. We analyze the structure of the call tree. When $F_n^{p,m}$ makes a call to $F_{n-1}^{m,m}$ we are in case 2(b). This is the only way to diminish the lower label so we must enter this case n times in order to eventually reach a call to $F_0^{i,m}$. To get to this case we must be in a recursive invocation of r in the body s. By updating the input estimate to $P \bowtie P_{in}$ we have incorporated the effect of one more level of nesting of recursive calls to r than is captured by P_{in}. The inductive hypothesis states that when the term $F_0^{i,m}$ appears in the call tree generated by $F_{n-1}^{p,m}$, the abstract environment has an input estimator, in this case P_{in}, that incorporates the effect of $n-1$ levels of

```
$D^m_n = \lambda h_1 \; . \; \ldots \; . \; \lambda h_n \; . \; \lambda e \; . \; \lambda P \; .$
  let
    $(P_{var}, P_{inv}) = split\ P\ [h_1, \ldots, h_n]$
    $P_1 = rename([h_1 \mapsto \underline{x_1}, \ldots, h_n \mapsto \underline{x_n}], P_{var})$
    $P_2 =$ if $e(r.new\_est)$ then
                  $F^{m,m}_n\ e[r.est \mapsto (P_1,\ base(P_1))]\ P_1$
              else $F^{m,m}_n\ e\ P_1$
    $P_3 = rename([\underline{x_1} \mapsto h_1, \ldots, \underline{x_n} \mapsto h_n], P_2)$
  in
    $join(P_{inv}, P_3)\ [h_1, \ldots, h_n]$

$F^{p,m}_n = \lambda e \; . \; \lambda P \; .$
  let $(P_{in}, P_{out}) = e(r.est)$ in
    if $e(r.new\_est)$ then
      { ---- CASE 1 (generalize output?) ---- }
      let
        $P_1 = insert([x_1 \mapsto \underline{x_1}, \ldots, x_n \mapsto \underline{x_n},$
                       $l_1 \mapsto \underline{nil}, \ldots, l_m \mapsto \underline{nil}], P)$
        $P_2 = \mathcal{A}[\![\ s\ ]\!]\ e[r.new\_est \mapsto false,\ r.D \mapsto D^{m-1}_n]\ \ P_1$
        $NP_{out} = remove([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2)$
      in
        if $NP_{out} \sqsubseteq P_{out}$ then
          $P_{out}$  { -- 1(a) -- }
        else
          $F^{p-1,m}_n\ e[r.est \mapsto (P, (NP_{out} \bowtie P_{out}))]\ P$ { -- 1(b) -- }
    else
      { ---- CASE 2 (generalize input?) ---- }
      if $P \sqsubseteq P_{in}$ then
        $P_{out}$  { -- 2(a) -- }
      else
        let
          $NP_{in} = (P \bowtie P_{in})$
        in { -- 2(b) -- }
          $F^{m,m}_{n-1}\ e[r.new\_est \mapsto true,\ r.est \mapsto (NP_{in},\ base(NP_{in}))]\ NP_{in}$
```

Figure 4.20: $F^{p,m}_n$ approximants.

nesting of recursive calls of r. Thus, $F_n^{i,m}$ incorporates the effect of any computation that encounters at least n nested calls. ∎

Because the domains are of finite height we will reach a general enough input estimate in some finite number of iterations of F. Now we can consider a family of iterates, $F_n^{p,m}$, with n assumed fixed and p and m varying. We assume that n has been chosen large enough that in all subsequent iterations of F the input estimates in the environment do not need to be generalized. We say that n is large enough for $F_n^{p,m}$ to *cover* all the memories encountered in the call tree describing the unwindings of the standard semantic definition of the recursive procedure.

We say that an environment e has a *m-safe estimator*, $e(r.est) = (P_{in}, P_{out})$ for r if for any path matrix P that is safe for M, env and $P \sqsubseteq P_{in}$ we have that P_{out} is a safe estimate for the memory and environment after $Q^{(m)}$ has been applied to (M, env). We call an environment that has (Ω, Ω) as the estimator an *initialized environment.* The key fact that we need to prove is the following.

Lemma 6. Suppose that $P \lhd (M, env)$. Suppose that $aenv \lhd env$ and that $aenv$ is an initialized environment. Suppose n is chosen large enough that $F_n^{p,m}$ covers $Qv_1 \ldots v_l M$, where the $v_1 \ldots v_l$ are fixed arguments to the procedure r. Then the following statements hold.

1. $F_n^{p,m} aenv P \lhd (Q^{(p)} v_1 \ldots v_l M, env)$.

2. Any leaf in the call tree of $F_n^{p,m} aenv P$ is either undefined or has an abstract environment containing an p-safe estimator.

Proof: Note that Part 1 follows immediately from part 2 because the only way that F only returns results that are stored estimators. We need, however, to carry both assertions in our inductive hypothesis since the proof of part 2 relies on the safety asserted in part 1.

The proof is by induction on p. The first part of the base case (the fact that the estimate is safe) has been done above. The second part of the base case (the statement that the estimator stored at the end of the base case is 1-safe) is as follows. Note that the pair stored in the environment as $e(r.est)$ is essentially $(P, base(P))$. Thus, if the stored estimate is used to compute the effect of a procedure call it will give the correct result for the first unwinding of the standard semantics provided the input is less than P. In other words it is a 1-safe estimator.

Assume both assertions true for $p-1$. Now consider the call tree of $F_n^{p,m}\, aenv\, P$. Any path through this tree that terminates in mn or fewer steps. If the end of such a path is undefined, then the corresponding path through the call tree of the corresponding expression in the standard semantics is also undefined. Note that our assumption on n assures us that we will *not* produce an undefined estimate because of an insufficiently generalized input approximation.

Consider the case where the last expression in the path through the call tree is a final path matrix. The last conditional encountered in the path must have resulted in case 1(a) or 2(a). In case 1(a), NP_{out} results from evaluating the body s in an environment that contains a $p-1$ safe estimator. Using the structural induction hypothesis on the body s and the induction hypothesis for recursive calls to r within s we see that NP_{out} is safe for $Q^{(p)}$. Since $NP_{out} \sqsubseteq P_{out}$, P_{out} is also safe for $Q^{(p)}$ and, in fact, P_{out} is a p-safe estimator. In case 2(a), the path through the call tree is via a call to r embedded in the body s. Thus P_{out} needs to be safe for $Q^{(p-1)}$ only, which it is by the inductive hypothesis.

Finally, we need to show that we have safe estimators at the leaves of the call tree. Consider the call tree for $F_n^{p,m}$ *aenv* P. Note that the first conditional encountered in the call tree must be case 1, where $e(r.new\_est = true)$. Thus, all paths through the call tree must either terminate here in one step (case 1(a)) or continue with further refinements (case 1(b)). We have already discussed case 1(a). Now consider the term for case 1(b)

$$F_n^{p-1,m}\ aenv[r.est \mapsto (P, (NP_{out} \bowtie P_{out}))]\ P.$$

By the inductive hypothesis, $F_n^{p-1,m}$ will produce at the leaves of the call tree rooted at it estimates that are $p-1$ safer than the initial estimate. The initial estimate at this term is, however, itself 1-safe because we heve merged in the effect of evaluating the body, s, of r. Thus, the leaves of the call tree contain p-safe estimators. ∎

Unfortunately the $F_n^{p,m}$ do not correspond to $F^{(k)}$ in any simple way. It is the latter that arise when we compare the effects of the $D^{(k)}$ and the $Q^{(k)}$ by induction on k. We require the more complex $F_n^{p,m}$ because $F^{(k)}$ gives us insufficient information about the quality of the estimators used. We handle this in the following way. First, we note that by choosing n sufficiently large we can say that $F_n^{k,k}$ is safe for $Q^{(k)}$, this is lemma 6. Lemma 5 assures us that such an n exists. Thus in effect we fix n and work with $F_n^{k,k}$. This does not tell us that F is safe for Q because we do not know that the $F_n^{p,m}$ converge to F; we only know that the $F^{(k)}$ converge to F. Thus, to complete the argument, we establish the following lemma.

Lemma 7. The least upper bound of the $F_n^{p,m}$ is F.

Proof: We work with terms of the form $F_n^{m,m}$ as these certainly converge to the same least upper bound as the $F_n^{p,m}$. First note that the $F_n^{m,m}$ are a directed set as, clearly, $F_{n_1}^{m_1,m_1}$ and $F_{n_2}^{m_2,m_2} \sqsubseteq F_{max(n_1,n_2)}^{max(m_1,m_2),max(m_1,m_2)}$. Thus the $F_n^{m,m}$ have a least upper bound. Now we claim that the $F_n^{m,m}$ dominate the $F^{(k)}$. Given any k, pick an n large enough for the input estimate to have become general enough and pick m to be bigger than k, clearly we have $F^{(k)} \sqsubseteq F_n^{m,m}$. Thus, the least upper bound of the $F^{(k)}$ (i.e. F) is less than the least upper bound of the $F_n^{m,m}$. Finally, each of the $F_n^{m,m}$ is clearly less than F so the least upper bound of the $F_n^{m,m}$ is F.∎

The desired safety result now follows from the fact that safety is an ω-inclusive relation. The overall structural induction is now complete. ∎

Chapter 5

Interference and Parallelization

In this chapter we present three methods for interference analysis and parallelization of SIL programs. These methods use the path matrices computed by the interference analysis tools described in the previous chapters.

5.1 Interference between Basic Statements

The first interference analysis method is used to determine if n basic handle statements interfere. As illustrated by figure 5.1, we can use such an interference analysis method to determine if several sequential statements can be transformed into a single parallel statement.

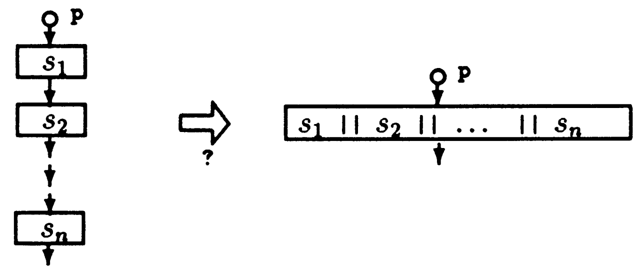


Figure 5.1: Transforming sequential statements to a parallel statement.

We first consider interference between two statements. More precisely, given two handle statements s_i and s_j and a path matrix p, we determine if one statement writes to a location that the other statement reads or writes. We then extend this method to handle n statements.

For this analysis, we define the following abstraction for a location. A *location* is denoted by a pair *(name, kind)*, where *name* is the name of a variable and *kind* is either the special designation *var* (variable), or a field name. For binary trees the possible kinds are: *var* - a variable, *left* - the left field of a node, *right* - the right field of a node, or *value* - the value field of a node.

We also define an alias function, $\mathcal{A}(a, f, p)$. Given a name a, a field kind f, and a path matrix p, the alias function returns the set of locations that may be aliased to location (a, f). Location (x, f) is an element of $\mathcal{A}(a, f, p)$ iff the path matrix entry $p[a, x]$ contains the path expression S, $(S + e)$, $S?$, or $(S + e)?$. Note that the path S indicates that locations (x, f) and (a, f) are *definite aliases*, while the other three possibilities indicate that locations (x, f) and (a, f) are *possible aliases*.

For each kind of basic handle statement, we have defined functions $\mathcal{R}(s, p)$ and $\mathcal{W}(s, p)$. Given a statement s and a path matrix p, $\mathcal{R}(s, p)$ defines a set of locations possibly read by s. Similarly, $\mathcal{W}(s, p)$ defines a set of locations possibly written by s. These functions are presented in figure 5.2.

Statement	Read Set $\mathcal{R}(s,p)$	Write Set $\mathcal{W}(s,p)$
$a := nil$	$\{\}$	$\{(a, var)\}$
$a := new()$	$\{\}$	$\{(a, var)\}$
$a := b$	$\{(b, var)\}$	$\{(a, var)\}$
$a := b.f$	$\{(b, var)\} \cup \mathcal{A}(b, f, p)$	$\{(a, var)\}$
$a.f := b$	$\{(a, var), (b, var)\}$	$\mathcal{A}(a, f, p)$

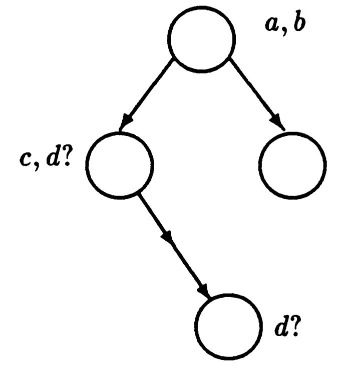
Figure 5.2: Functions for read and write sets of statement s relative to path matrix p.

The *interference set*, $\mathcal{I}(s_i, s_j, p)$, is defined as the set of locations through which statements s_i and s_j may interfere when executed at a program point with path matrix p. If $\mathcal{I}(s_i, s_j, p) = \{\}$, then there is no interference between s_i and s_j, and it is safe to execute s_i and s_j in parallel.

$$\begin{aligned}\mathcal{I}(s_i, s_j, p) &= \\ &\quad \left[\mathcal{W}(s_i,p) \cap \left(\mathcal{R}(s_j,p) \cup \mathcal{W}(s_j,p) \right)\right] \\ &\cup \left[\mathcal{W}(s_j,p) \cap \left(\mathcal{R}(s_i,p) \cup \mathcal{W}(s_i,p) \right)\right]\end{aligned}$$

Three examples of interfering statements are given in figure 5.3. The first example illustrates variable interference; the statement $x := a.left$ writes variable x, while the statement $y := x$ reads variable x. The second example illustrates two statements that interfere by accessing the *left* field of the same node. Since a and b are handles to the same node, the statement $x := a.left$ reads the same location that statement $b.left := nil$ writes. The third example illustrates the conservative nature of the interference analysis. Note that handles c and d may be handles to

A tree and corresponding path matrix



	a	b	c	d
a	S	S	L^1	D^+
b	S	S	L^1	D^+
c			S	$S?, R^+?$
d			$S?$	S

Example 1

	Statement	$\mathcal{R}(s_i, p)$	$\mathcal{W}(s_i, p)$	$\mathcal{I}(s_1, s_2, p)$
s_1	x := a.left	{(a,var),(a,left),(b,left)}	{(x,var)}	{(x,var)}
s_2	y := x	{(x,var)}	{(y,var)}	

Example 2

	Statement	$\mathcal{R}(s_i, p)$	$\mathcal{W}(s_i, p)$	$\mathcal{I}(s_1, s_2, p)$
s_1	x := a.left	{(a,var),(a,left),(b,left)}	{(x,var)}	{(a,left),(b,left)}
s_2	b.left := nil	{(b,var)}	{(b,left),(a,left)}	

Example 3

	Statement	$\mathcal{R}(s_i, p)$	$\mathcal{W}(s_i, p)$	$\mathcal{I}(s_1, s_2, p)$
s_1	n := d.value	{(d,var),(d,value),(c,value)}	{(n,var)}	{(c,value),
s_2	c.value := 0	{(c,var)}	{(c,value),(d,value)}	(d,value)}

Figure 5.3: Examples of interfering statements.

the same node, or handle d may be some number of *right* links below handle c. In the first case, the statements $n := d.value$ and $c.value := 0$ would interfere on the *value* field. However, in the second case c and d refer to different nodes, and the statements would not interfere. Thus, uncertainty in the path matrix results in a conservative approximation of interference.

We can generalize this method to determine if n basic statements $[s_1, \ldots, s_n]$ interfere. $\mathcal{R}_n$, $\mathcal{W}_n$, and $\mathcal{I}_n$ are defined as follows.

$$\mathcal{R}_n([s_1, \ldots, s_n], p) = \bigcup_{i=1,n} \mathcal{R}(s_i, p)$$

$$\mathcal{W}_n([s_1, \ldots, s_n], p) = \bigcup_{i=1,n} \mathcal{W}(s_i, p)$$

$$\begin{aligned} \mathcal{I}_n([s_1, \ldots, s_n], s_j, p) \; = & \\ & \left[\mathcal{W}_n([s_i, \ldots, s_n], p) \cap \begin{pmatrix} \mathcal{R}(s_j, p) \cup \\ \mathcal{W}(s_j, p) \end{pmatrix}\right] \cup \\ & \left[\mathcal{W}(s_j, p) \cap \begin{pmatrix} \mathcal{R}_n([s_1, \ldots, s_n], p) \cup \\ \mathcal{W}_n([s_1, \ldots, s_n], p) \end{pmatrix}\right] \end{aligned}$$

Statements $[s_1, \ldots, s_n]$ do not interfere at a program point with path matrix p if

$$\left(\bigcup_{i=1,n-1} \mathcal{I}_n([s_1, \ldots, s_i], s_{i+1}, p)\right) = \{\}.$$

Note that we can incrementally build the set of statements that can be executed in parallel by first computing the interference set for s_1 and s_2. If this set is empty, then we can schedule s_1 and s_2 in parallel. We can continue to add statements to the parallel schedule until we reach a statement that results in a non-empty interference set.

5.2 Interference between Procedure Calls

In the previous section we outlined a fine-grain analysis that is used to detect interference among single statements. In this section we outline a coarse-grain approach to interference analysis between procedure calls. Given two procedure calls $f(x_1, x_2, \ldots, x_m)$ and $g(y_1, y_2, \ldots, y_n)$ at a program point with path matrix p, we wish to determine if the calls to f and g interfere.

A first approximation of the analysis can be obtained by examining the relationship between the handle arguments of f and the handle arguments of g at the program point just before the procedure call. The only nodes that a procedure can access are those accessed via a path from a handle argument. Recall that data structures of type *TREE* have the property that if two handles h_i and h_j are unrelated, then all of the nodes accessed from h_i are unrelated to those accessed from h_j. Thus, if all handle arguments x_i of the call to f are unrelated to all of the handle arguments y_j of the call to g, we can conclude that the two procedure calls do *not* interfere. Since the path matrix at the point before the procedure calls is guaranteed to contain all possible relationships among handles, we can conclude that handles x_i and y_j are unrelated if $p[x_i, y_j] = p[y_j, x_i] = \{\}$. If the data structure is a *DAG* we must also check that there is no *DAG* handles d_k such that d_k is related to both x_i and y_j.

As an example of using this method, consider the program presented in figure 5.4. This program adds 1 to the left sub-tree, adds -1 to the right sub-tree and then reverses the whole tree. Note that the procedure *add_n* updates the nodes of a tree and that *reverse* actually changes the structure of the tree. The three critical program points for parallelization of procedure calls occur at program points A, B, and C. First, consider the path matrix p_A at program point A. By examining p_A we can determine that handles *l_side* and *r_side* are not related ($p_A[l\_side, r\_side] = p_A[r\_side, l\_side] = \{\}$). Therefore, the procedure calls $add\_n(l\_side, 1)$ and $add\_n(r\_side, -1)$ may be executed in parallel.

A more interesting sort of parallelism is exhibited at program points B and C. Consider the path matrix p_B at program point B. The handles in p_B can be divided into two groups: (1) *h*2* is used as a symbolic name for the calling procedure's argument handle, and (2) h, l, and r contain information about the handles local to the current invocation of *add_n*. The path matrix p_B summarizes all possible relationships between handles for the recursive calls of *add_n*. Since handles l and r are not related in p_B, it is always safe to execute the recursive calls $add\_n(l, n)$ and $add\_n(r, n)$ in parallel. A similar result is obtained for the recursive calls to *reverse* at program point C.

A more accurate analysis of procedure call interference can be performed by using further information about how handle arguments to a procedure are used in the body of the procedure. With the previous method, we assumed that nodes accessed through a handle argument may be updated. This is an overly conservative assumption, since some procedures may only read nodes. By adding further analysis it can be determined that a handle argument is either a *read-only* argument or an *update* argument. Handle argument x_i is *read-only* if all nodes accessed through x_i are read and not written, otherwise x_i is an *update* argument. The interference analysis can now be restricted to checking for interference due to update arguments. Let f_{update} be the set of update arguments of the call $f(x_1, \ldots, x_m)$ and g_{update} be the set of update arguments of the call $g(y_1, \ldots, y_n)$. The calls to f and g will not interfere if all handles in f_{update} are unrelated to all arguments of g and all handles in g_{update} are unrelated to all arguments of f.

Using the technique for detecting basic statement interference presented in the previous section and the technique for detecting procedure call interference pre-

```
program add_and_reverse
nodedef left,right: handle; value: int end;
procedure main()
  root,l_side,r_side: handle; i: int
begin
  { ... build a tree at root ... }
  l_side := root.left;
  r_side := root.right;
  { ⇐ PROGRAM POINT A – $p_A$ }
  add_n(l_side,1);
  add_n(r_side,–1);
  reverse(root)
end;
procedure add_n(h:handle; n: int)
  l,r: handle
begin
  if h ≠ nil then
    begin
      h.value := h.value + n;
      l := h.left;
      r := h.right;
      { ⇐ PROGRAM POINT B – $p_B$ }
      add_n(l,n);
      add_n(r,n)
    end
end;
procedure reverse(h:handle)
  l,r: handle
begin
  if h ≠ nil then
    begin
      l := h.left;
      r := h.right;
      { ⇐ PROGRAM POINT C – $p_C$ }
      reverse(l);
      reverse(r);
      h.left := r;
      h.right := l
    end
end;
```

p_A

	root	*l_side*	*r_side*
root	S	L^1	R^1
l_side		S	
r_side			S

p_B

	h∗1	*l*	*r*
h∗1	S	L^1	R^1
l		S	
r			S

p_C

	h∗2	*h*	*l*	*r*
h∗2	S	S	L^1	R^1
h	S	S	L^1	R^1
l			S	
r				S

Figure 5.4: Example program and path matrices.

sented in this section, the example program of figure 5.4 can be transformed into the parallel program shown in figure 5.5.

5.3 Interference between Statement Sequences

In this section we examine the problem of determining if two statement sequences interfere. As illustrated in figure 5.6, we want to determine if it is safe to execute statements sequences U and V in parallel (given the same initial program point). More precisely, given statement sequences $U = [u_1, \ldots, u_m]$ and $V = [v_1, \ldots, v_n]$ at an initial program point with path matrix p, we want to determine if one statement sequence writes to a location that the other statement sequence reads or writes. This sort of analysis is useful both in checking that the parallel specification of $U \parallel V$ is safe and in determining that the statement sequence U; V can be transformed into the parallel statement $U \parallel V$.

For this analysis we require a new notion of location. Let $\mathcal{L}$ be the set of handles that are used before they are defined in either U or V. All nodes that can be accessed in both U and V must be accessed along some path from a handle in $\mathcal{L}$ [1]. Therefore, we will refer to locations by their access path from handles in $\mathcal{L}$. A *relative location* is a triple (*name*,*field_type*,*access_path*) where *name* is the name of a handle in $\mathcal{L}$, *field_type* is one of var, $left$, $right$, or $value$, and *access_path* is a set of path expressions describing the path from *name* to the node which is being read or updated.

The relative read and write functions $\mathcal{W}^r(s, p, \mathcal{L})$ and $\mathcal{R}^r(s, p, \mathcal{L})$ are defined as in figure 5.7. In these rules we use the relative alias function $\mathcal{A}^r$. Given a handle h, a field name f, a set of live handles $\mathcal{L}$, and a path matrix p, $\mathcal{A}^r(h, f, \mathcal{L}, p)$ returns the set of relative locations which are possibly aliased to the location referred to by $h.f$. $\mathcal{A}^r(h, f, \mathcal{L}, p)$ contains the relative location (l, f, r) iff l is one of the handles in $\mathcal{L}$ and the path matrix entry $p[l, h]$ contains the path expression r.

We define $\mathcal{R}^r_n([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L})$ to be the *relative read set* for statement sequence $[s_1, \ldots, s_n]$ and $\mathcal{W}^r_n([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L})$ to be the *relative write set* for statement sequence $[s_1, \ldots, s_n]$.

$$\mathcal{R}^r_n([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L}) = \bigcup_{i=1,n} \mathcal{R}^r(s_i, p_i, \mathcal{L})$$

$$\mathcal{W}^r_n([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L}) = \bigcup_{i=1,n} \mathcal{W}^r(s_i, p_i, \mathcal{L})$$

Let $P = [p_1, \ldots, p_m]$ be the sequence of path matrices associated with the statements u_i of U, and $Q = [q_1, \ldots, q_n]$ be the sequence of path matrices associated with

[1]For ease of presentation, we will assume that the handles in $\mathcal{L}$ are not redefined in U or V. This restriction can be lifted by automatic renaming of variables.

```
program add_and_reverse

nodetype left,right: handle; value: int end;

procedure main()
  root,l_side,r_side: handle; i: int
begin
  { ... build a tree at root ... }
  l_side := root.left || r_side := root.right;
  add_n(l_side,1) || add_n(r_side,−1);
  reverse(root)
end;

procedure add_n(h:handle; n: int)
  l,r: handle
begin
  if h ≠ nil then
    begin
      h.value := h.value + n;
      l := h.left || r := h.right;
      add_n(l,n) || add_n(r,n)
    end
end;

procedure reverse(h:handle)
  l,r: handle
begin
  if h ≠ nil then
    begin
      l := h.left || r := h.right;
      reverse(l) || reverse(r);
      h.left := r || h.right := l
    end
end;
```

Figure 5.5: Parallel version of example program.

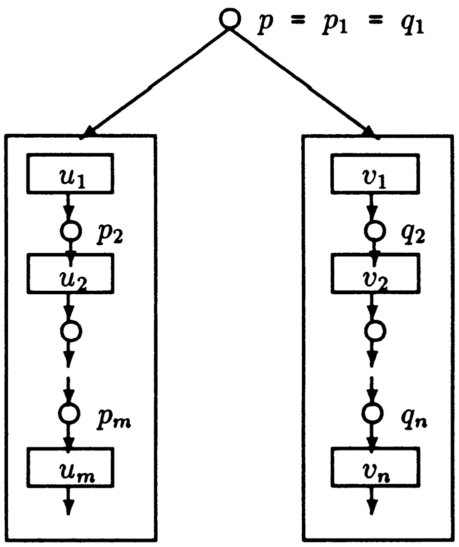


Figure 5.6: Initial path matrix p with two parallel statement sequences U and V.

Statement	Relative Read Set $\mathcal{R}^r(s,p,\mathcal{L})$	Relative Write Set $\mathcal{W}^r(s,p,\mathcal{L})$
$a := nil$	$\{\}$	$\{(a, var, S)\}$
$a := new()$	$\{\}$	$\{(a, var, S)\}$
$a := b$	$\{(b, var, S)\}$	$\{(a, var, S\}$
$a := b.f$	$\{(b, var, S)\} \cup \mathcal{A}^r(b, f, \mathcal{L}, p)$	$\{(a, var, S)\}$
$a.f := b$	$\{(a, var, S), (b, var, S)\}$	$\mathcal{A}^r(a, f, \mathcal{L}, p)$

Figure 5.7: Relative read and write sets given path matrix p and live handles $\mathcal{L}$.

the statements v_j of V (as illustrated in figure 5.6). The *relative interference set*, $\mathcal{I}^r(U, P, V, Q, \mathcal{L})$, is defined as follows.

$$\begin{aligned}
&\mathcal{I}^r(U, P, V, Q, \mathcal{L}) \;= \\
&\quad \Big[\mathcal{W}^r(U, P, \mathcal{L}) \cap \big(\; \mathcal{R}^r(V, Q, \mathcal{L}) \cup \mathcal{W}^r(V, Q, \mathcal{L}) \;\big)\Big] \cup \\
&\quad \Big[\mathcal{W}^r(V, Q, \mathcal{L}) \cap \big(\; \mathcal{R}^r(U, P, \mathcal{L}) \cup \mathcal{W}^r(U, P, \mathcal{L}) \;\big)\Big]
\end{aligned}$$

If the data structure is a *TREE* at the program point just before $U \parallel V$, then U and V do not interfere if $\mathcal{I}^r(U, P, V, Q, \mathcal{L}) = \{\}$. The proof of this observation is an induction on the height of the tree. To extend this method to *DAGS* it is necessary to include all handles to *DAG* nodes in the set of live handles $\mathcal{L}$.

Chapter 6

Generating Parallel Programs

In the previous chapters we presented interference analysis methods and parallelization techniques to support automatic parallelization of sequential SIL programs. As illustrated in figure 6.1, interference analysis and parallelization form the first two stages of our parallelizing system. To provide a complete experimental system, we implemented a back-end that translates parallelized SIL programs into parallel C programs for a shared memory machine. The complete parallelizing system provides a concrete use of our interference analysis and parallelization techniques. In addition, the system provides an experimental framework in which we can investigate further areas of research.

In the first section of this chapter, we present our approach to generating parallel programs for the BBN GP1000 (Butterfly[1]) Parallel Processor running Mach 1000. In the second section we present, as an example, the parallel program generated for adaptive bitonic sort [BN89]. In the final section, we discuss new areas of research that were suggested by the results of these experiments.

6.1 Generating Parallel Code for the Butterfly

The GP1000 parallel processor is an MIMD shared memory machine [BBN88a]. Each processor has a local memory, and may access any other processor's memory through the Butterfly switch. The switch is a collection of switching nodes that interconnects processor nodes and provides each processor with access to shared memory. Some important timing information for memory accesses is given in figure 6.2[BBN89]. Note that global reads and writes are substantially slower than local reads and writes. Synchronization on the Butterfly is supported through a collection of atomic operations. A 16-bit atomic operation takes about 30 microseconds, or approximately 30 times as long as a local memory access.

[1]Butterfly is a trademark of Bolt Beranek and Newman Inc.

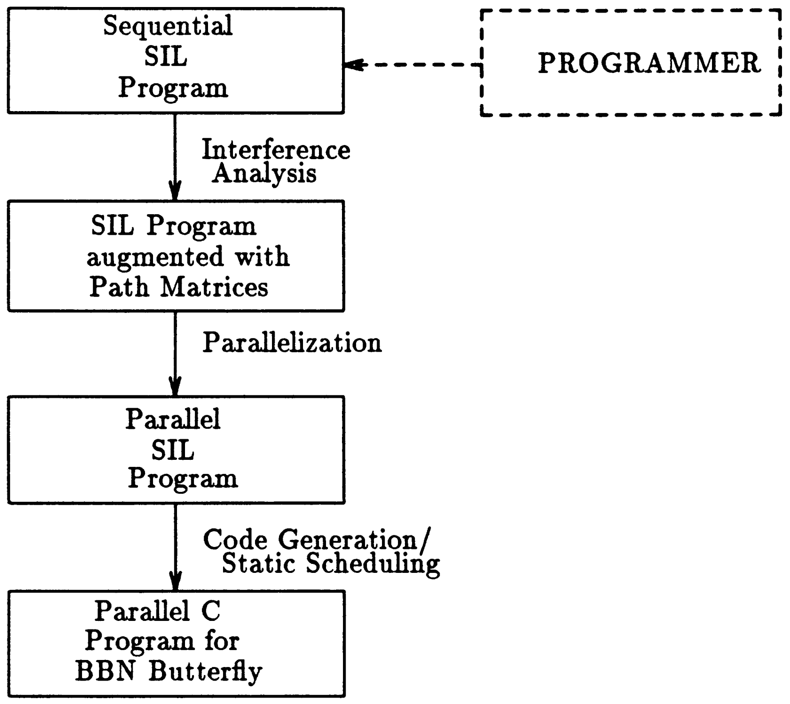

Sequential SIL Program
PROGRAMMER
Interference Analysis
SIL Program augmented with Path Matrices
Parallelization
Parallel SIL Program
Code Generation/ Static Scheduling
Parallel C Program for BBN Butterfly

Figure 6.1: Current System

	Global (microseconds)	Local (microseconds)	Global/ Local
Read	8.12	1.22	6.7
Write	3.52	.81	4.3

Figure 6.2: Global/Local memory access times (unloaded switch).

6.1.1 Strategy

The basic strategy is to exploit coarse-grain parallelism at the procedure and function call level. More specifically, if there are two procedure calls $f(x)$ and $g(y)$ that do not interfere, we wish to execute these in parallel. Since our main objective was parallelizing programs that use recursive data structures, we were particularly interested in designing code generation strategies that handle recursive unfoldings of parallel procedure calls in a reasonable manner.

Two common types of unfoldings have been identified. Examples of procedures exhibiting these types of unfoldings are given in figure 6.3. The first example is a procedure that reverses trees. Note that each invocation of *reverse* leads to another opportunity for parallel procedure calls *(reverse(l) || reverse(r))*. Thus, the total number of possible parallel procedure calls is unbounded (denoted by $*$). We refer to the recursive unfolding in reverse as $(*,*)$*-unfolding.*

The second procedure, *process_next*, processes a data-structure that has two links, a head link and a tail link. For each invocation of *process_next*, procedure *foo* is executed on the head, and a recursive call to *process_next* is executed on the tail. If foo contains no parallel procedure calls, then each invocation of *process_next* gives one more opportunity for a parallel call. We refer to the unfolding of $foo(h) \| process\_next(t)$ as $(1,*)$*-unfolding.*

Given these two types of unfoldings, we designed a strategy for statically generating programs that dynamically schedule reasonable unfoldings of parallel procedure calls. For $(*,*)$-unfoldings we use an *interval-based* processor allocation scheme which is similar to the *team-split* mechanism in PCP [Bro88]. For the $(1,*)$-unfoldings we use a *pool-based* allocation scheme.

Figure 6.4(a) illustrates the interval approach. Each processor p is associated with an interval that indicates the set of idle processors. Initially processor 0 is allocated the set of all processors, and as each parallel procedure call is encountered the interval is split. For example, given an initial interval of [0-7], the recursive calls in $reverse$ would split such that $reverse(r)$ would continue on processor 0 with interval [0-3], and $reverse(l)$ would be given to processor 4 with interval [4-7]. When the intervals become singletons, no more interval subdividing is possible, and execution reverts to the sequential case. This method is obviously well suited to problems which naturally sub-divide into equal sub-tasks. By allocating intervals to processors, the cost for scheduling processors is minimized by avoiding contention for synchronization locks, and a breadth-first expansion of the parallel call graph is obtained.

Clearly, the interval scheme is not well-suited to all sorts of unfoldings. In the case of $(1,*)$-unfoldings, we use the pool-based method. In this case, all processors in an interval are put into a pool, and processors are removed from and inserted into the pool. For example, in $process\_next$ each invocation of $foo(h)$ requests a processor from the pool, and when the invocation terminates the processor is returned to the pool. As indicated in figure 6.4(b), the interval and pool-based

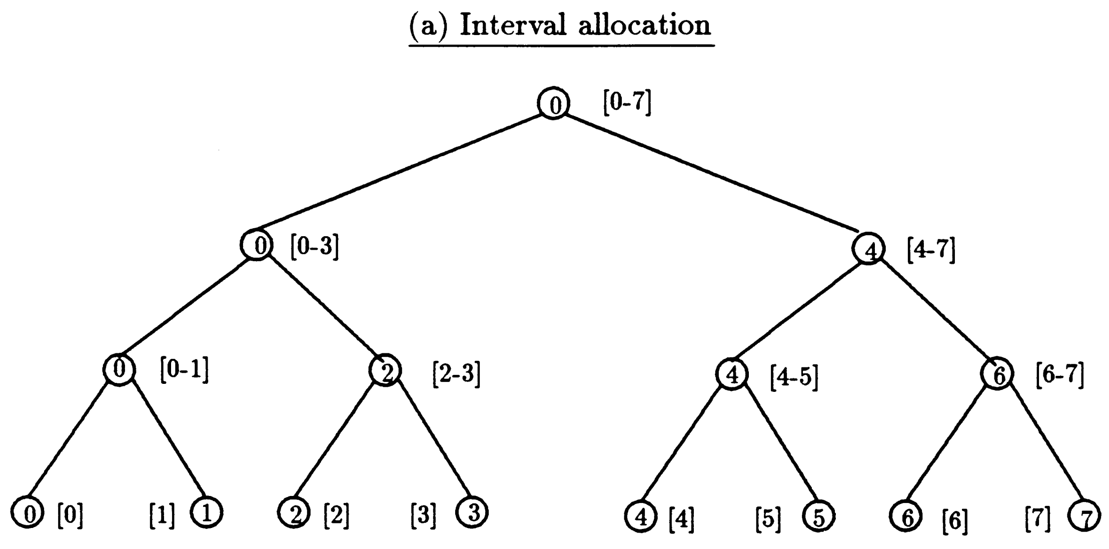
```
nodedef left, right: handle; value: int end;

procedure reverse(h: handle)
  l,r: handle
begin
  if $h \neq$ nil then
    begin
      $l$ := h.left;
      $r$ := h.right;
      reverse(l) || reverse(r);
      h.left := r;
      h.right := l
    end
end

nodedef head, tail: handle; value: int end;

procedure process_next(l: handle)
  h,t: handle
begin
  if $l \neq$ nil then
    begin
      $h$ := l.head;
      $t$ := l.tail;
      foo(h) || process_next(t)
    end
end
```

Figure 6.3: Examples of unfoldings.



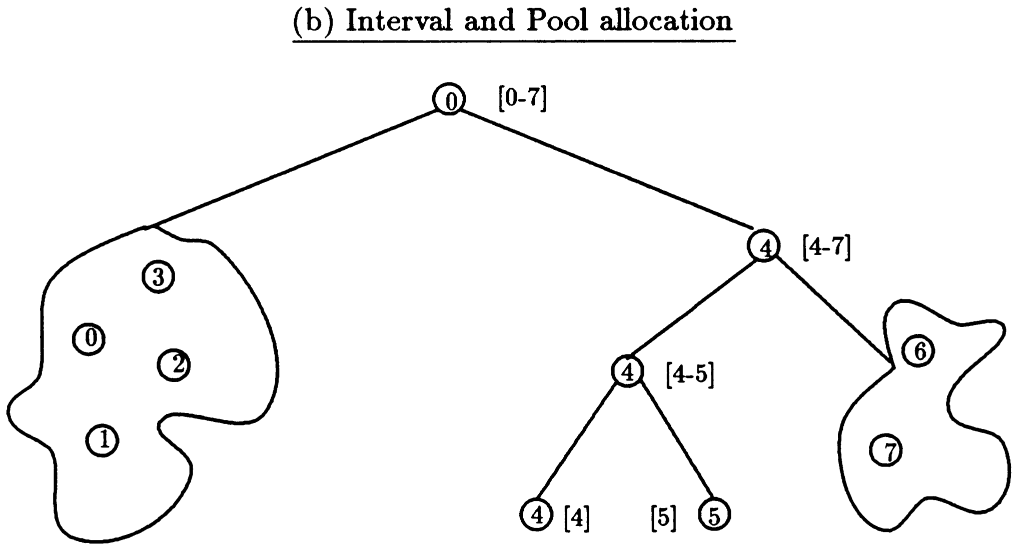


Figure 6.4: Interval and Pool allocation.

methods may be combined, with any interval being treated as a pool.

6.2 An example: Adaptive Bitonic Sort

The adaptive bitonic sort program, which is used to sort binary trees, consists of two main procedures: a recursive sort procedure *Bisort*, and a recursive merge procedure *Bimerge*. The *Bisort* procedure (figure 6.5) has three arguments: a *TREE* with *n-1* nodes (*root*), a *TREE* with 1 node (*sp_r*), and an integer direction (*dir*). *Bisort* recursively sorts two smaller problems, and then merges the result using *Bimerge*. The procedure *Bimerge* (figure 6.6) is more complex. The computation consists of two steps. The first step is to traverse down two paths in the tree, and at each step in the traversal possibly swap sub-trees. The second step consists of recursive calls to *Bimerge* on two smaller sub-problems.

The adaptive bitonic sort program is a good representative test of our interference analysis because it includes nested recursion, while loops, and nested conditionals. In addition, it relies on imperative tree updating for the swapping of sub-trees in *SwapLeft* and *SwapRight*, and it illustrates the problem of temporary *DAGS* being created (see the comments in *SwapLeft*).

6.2.1 Path Matrix Computation

For each statement in the program, our analysis gives the relationships among all live handles. This information can be used to determine that the recursive calls in *Bimerge* can be safely executed in parallel, and that the recursive calls in *Bisort* can be safely executed in parallel. Consider the recursive calls *Bimerge(rl,sp_l,dir)* and *Bimerge(rr,sp_r,dir)* in the body of the procedure *Bimerge*. Our analysis produces the path matrix given in figure 6.7 for the program point just before the recursive calls. The relationships in this path matrix indicate that the handles *rl*, *rr*, *sp_l*, and *sp_r* are all unrelated, and that there are no *DAG* nodes. From this information, we use the coarse-grain parallelizing rules from chapter 5 to infer that the two recursive calls do not interfere and it is safe to execute them in parallel. A similar result is easily obtained for the recursive calls of *Bisort(l,sp_l,dir)* and *Bisort(r,sp_r,!dir)* in the body of the procedure *Bisort*.

6.2.2 Generating the Parallel Program

Given a parallelized SIL program in which the recursive calls to *Bimerge* and *Bisort* have been scheduled in parallel, we automatically generated a parallel C program for the Butterfly. Both the sequential SIL program and the parallel C program are given in appendix D. The generated program uses the facilities provided by the BBN

```
procedure SwapValue(l,r: handle)
  temp: int
begin
  temp := l.value; l.value := r.value; r.value := temp
end; { SwapValue }

procedure SwapLeft(l,r: handle)
  ll,rl: handle
begin
  ll := l.left; rl := r.left;
  l.left := rl; { at this point rl will have two parents: r and l (DAG) }
  r.left := ll  { rl now has only one parent: l (TREE) }
end; { SwapLeft }

procedure SwapRight(l,r: handle)
  lr,rr: handle
begin
  lr := l.right; rr := r.right;
  l.right := rr; r.right := lr
end; { SwapRight }

procedure Bisort(root,sp_r: handle; dir:int)
  l, r, sp_l: handle
begin
  if root.left = nil then
    begin
      if (root.value > sp_r.value) xor dir then
        SwapValue(root,sp_r)
    end
  else
    begin
      l := root.left; r := root.right;
      sp_l := NewNode(root.value);
      Bisort(l,sp_l,dir); Bisort(r,sp_r,!dir);
      root.value := sp_l.value;
      Bimerge(root,sp_r,dir)
    end
end; { Bisort }
```

Figure 6.5: Bitonic Sort and Swap procedures.

```
procedure Bimerge(root: handle; sp_r:handle ; dir: int)
  rightexchange, elementexchange, temp : int;
  sp_l, pl, pr, rl, rr: handle
begin
  rightexchange := (root.value > sp_r.value) xor dir;
  if rightexchange then SwapValue(root,sp_r);
  pl := root.left; pr := root.right;
  while (pl ≠ nil) do
    begin
      elementexchange := (pl.value > pr.value) xor dir;
      if rightexchange then
        if elementexchange then
        { swap values and right subtrees, search path goes left }
          begin
            SwapValue(pl,pr); SwapRight(pl,pr);
            pl := pl.left; pr := pr.left
          end
        else
        { search path goes right }
          begin pl := pl.right; pr := pr.right end
      else
        if elementexchange then
        { swap values and left subtrees, search path goes right }
          begin
            SwapValue(pl,pr); SwapLeft(pl,pr);
            pl := pl.right; pr := pr.right
          end
        else
        { search path goes left }
          begin pl := pl.left; pr := pr.left end
    end; { while }
  if (root.left ≠ nil) then
    begin
      rl := root.left; rr := root.right;
      sp_l := NewNode(root.value);
      Bimerge(rl,sp_l,dir); Bimerge(rr,sp_r,dir);
      root.value := sp_l.value
    end
end; { Bimerge }
```

Figure 6.6: Bitonic Merge procedure.

	*root*9* (⊙,⊙⊙)	*sp_r*9* (●,oo)	*root* (●,⊙⊙)	*sp_r* (●,oo)	*sp_l* (●,oo)	*rl* (⊙,⊙⊙)	*rr* (⊙,⊙⊙)
*root*9*	S		S			L^1	R^1
*sp_r*9*		S		S			
root	S		S			L^1	R^1
sp_r		S		S			
sp_l					S		
rl						S	
rr							S

Figure 6.7: Path Matrix before recursive calls in Bimerge.

Uniform System[2] to organize the memory and data structures as illustrated in figure 6.8. Note that each processor's memory is divided into one section local to that processor, and another section that may be accessed by other processors. The stack and program code reside in the local section, while the heap and data structures required for synchronization and communication reside in the global section. The heap is managed as a distributed structure with locks. Each processor allocates from all the heaps in a round-robin fashion. Thus, the nodes in the data structures are evenly distributed over all processors.

Since both *Bisort* and *Bimerge* are $(*,*)$-*unfoldings*, the interval-based method is used in the generated code. Figure 6.9 gives the parallel procedure generated for *Bisort*. The body of the procedure supports two modes. If the interval allocated to the processor executing call to *Bisort* contains only one processor (`my_index == my_endpoint`), then further recursive calls use the sequential version (*SS_Bisort*), and no further scheduling overhead is incurred. When the interval contains more that one processor, the interval is divided between a parallel call to `Bisort(l,sp_l,dir)` and a local call to `Bisort(r,sp_r,!dir)`. Note that the overhead for starting the parallel call is quite small. It consists of some integer arithmetic to calculate the intervals, a small number of global memory writes, and an atomic operation to signal the processor to start.

6.2.3 Speedup results

The results of running the parallel program is given in figures 6.10, 6.11, 6.12, and 6.13. The parallel program was run on 1 to 32 processors. For each run, 10 random trees of sizes 128, 512, and 8192 were sorted.

First consider figures 6.10 and 6.11. In these graphs *speedup* is the ratio (execu-

[2]The Uniform System is a collection of macros and functions that support global memory allocation, data sharing, task generation, and synchronization [BBN88b].

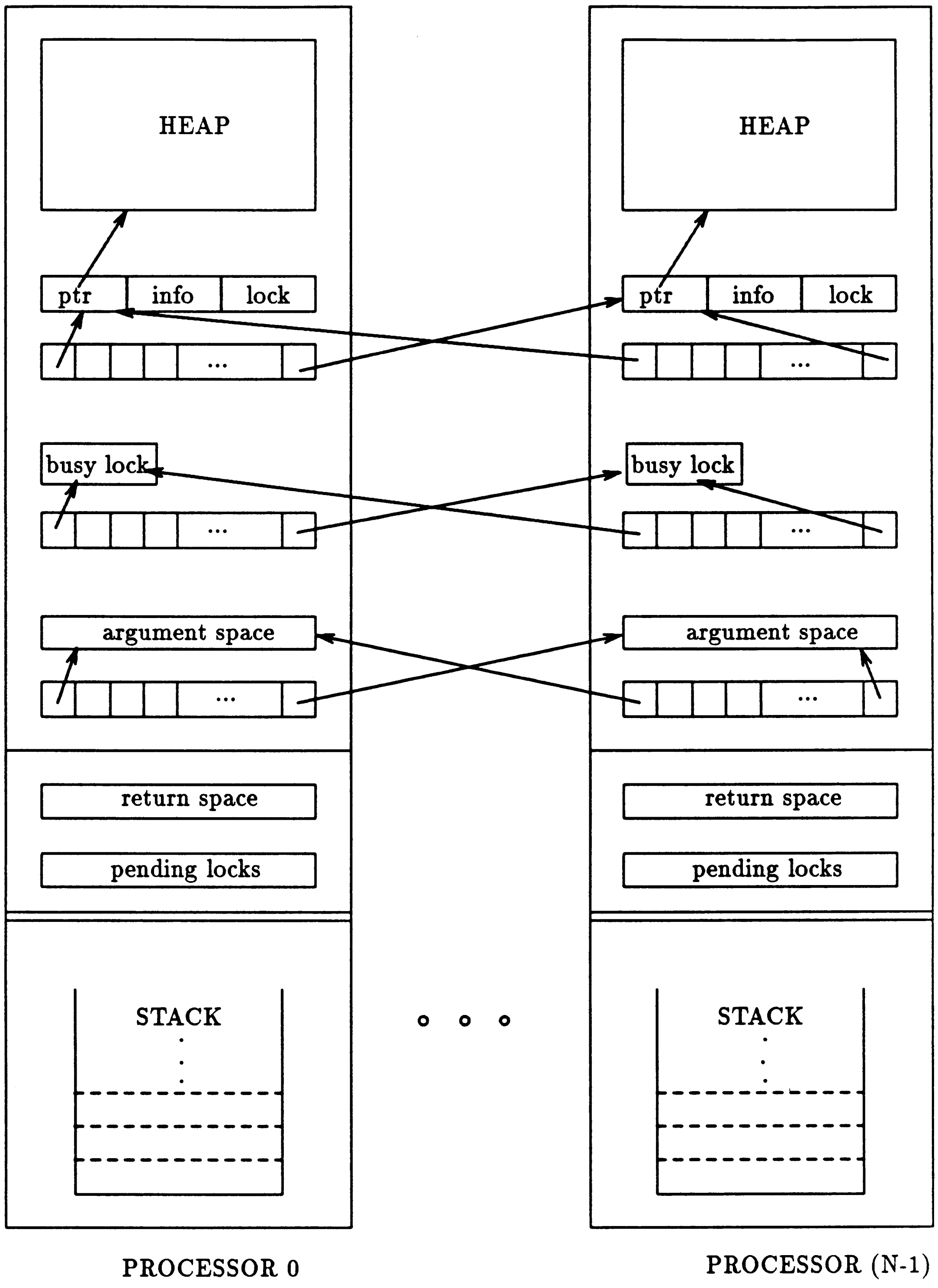


Figure 6.8: Global/Local memory layout.

```
int my_index, my_endpoint; /* interval */

void
/*******************/
Bisort(root,sp_r,dir)
/*******************/
HANDLE *root, *sp_r;
int dir;
{ HANDLE *l, *r, *sp_l;
  if ((root->left == NIL))
    { if (((root->value > sp_r->value) ^ dir))
        SwapValue(root,sp_r);
    }
  else
    { l = root->left; r = root->right;
      sp_l = NewNode(root->value);
      { int input_endpoint;
        input_endpoint = my_endpoint;
        if (my_endpoint == my_index)  /* only 1 proc */
          { SS_Bisort(l,sp_l,dir); SS_Bisort(r,sp_r,(! dir));
          }
        else /* more than one proc, fork off a procedure call */
          /* ======= set up for fork of Bisort(l,sp_l,dir); ========== */
          { int p, p_endpoint, *p_arglist, interval; short *p_pending;
            p_endpoint = my_endpoint;
            interval = (my_endpoint - my_index + 1) >> 1;
            my_endpoint -= interval; p = my_endpoint + 1;
            GET_PENDING(p_pending,1); /* get pending lock for p */
            /* set up proc num, interval, and args for call */
            p_arglist = arg_list_ptrs[p];
            *((short **) p_arglist)++ = p_pending;
            *((int*) p_arglist)++ = DD_Bisort;
            *((int*) p_arglist)++ = p_endpoint;
            *((HANDLE **) p_arglist)++ = l;
            *((HANDLE **) p_arglist)++ = sp_l;
            *((int *) p_arglist)++ = dir;
            START(p);  /* let processor p start */
            Bisort(r,sp_r,(! dir));
            WAIT_ZERO(*p_pending);
            RELEASE_PENDING; /* release pending lock */
          }
        my_endpoint = input_endpoint;
      }
      root->value = sp_l->value;
      Bimerge(root,sp_r,dir);
    }
}
```

Figure 6.9: Parallel C version of Bisort.

tion time for 1 processor) / (execution time for n processors). In figure 6.11 we plot only the points for 1,2,4,8,16, and 32 processors. For each run, the nodes in the tree were allocated equally to each processor involved in the run. This means that the 1-processor run accesses only local memory, while all other runs must access some non-local memory. Even with the increased activity to non-local memory, we see that speedup is achieved for all problem sizes. For trees of size 128, maximum speedup is achieved at 16 processors. At this point the sub-problems given to each processor are of size 8. For sub-problems smaller than 8 the overhead of starting a new processor is prohibitive. For larger problem sizes, we see that the speedup is still increasing at 32 processors. It should be noted that these data sets are relatively small, and it is quite encouraging to see sustained speedup for 32 and more processors.

In order to study the effect of non-local memory accesses, we ran an experiment where the data structure nodes were always equally distributed among the 32 memories. Thus, the time to access the data is equal for all test cases, regardless of the number of processors involved in the test. Figure 6.12 gives the comparison of speedups for this test, and the previous case (where data was allocated only on the processors actually involved in the test). Note that the test in which all memory accesses are equally expensive gives us a better speedup. From this we conclude that locality of reference is an important factor that should be considered.

The final graph (figure 6.13) gives the execution time for on 1 processor, versus the execution time on 32 processors. Note that a problem size of at least 32 is required before the 32-processor case exhibits speedup over the 1-processor case. With problem sizes of 512 or more, scheduling overhead is insignificant and the 32-processor executes about 7 times faster than the 1-processor case. This graph demonstrates that there is some minimal problem size that is required before the benefit of adding a new processor to the problem outweighs the overhead associated in starting a new computation.

6.3 Further Issues

As outlined in the previous section, there are two major problems that need to be addressed. The first problem concerns the lack of locality of data. Since the data structures are dynamically allocated and imperatively updated, it is difficult to exploit any locality of reference. Thus, when comparing the parallel program (many heaps) to a sequential program (one local heap) there is a large penalty for non-local memory accesses in the parallel case. The second problem concerns the overhead associated with allocating a task (parallel procedure call) to an idle processor. Allocating idle processors to work on a problem is only beneficial when the amount of work (problem size) is large enough to mask the overhead.

Dealing with these issues is difficult in the presence of dynamic data structures. Unlike arrays, the size and shape of the recursive data structures is unknown at compile-time, and in fact may vary drastically throughout the execution. In addition, small changes to the data structure, like swapping sub-trees can greatly affect locality of data.

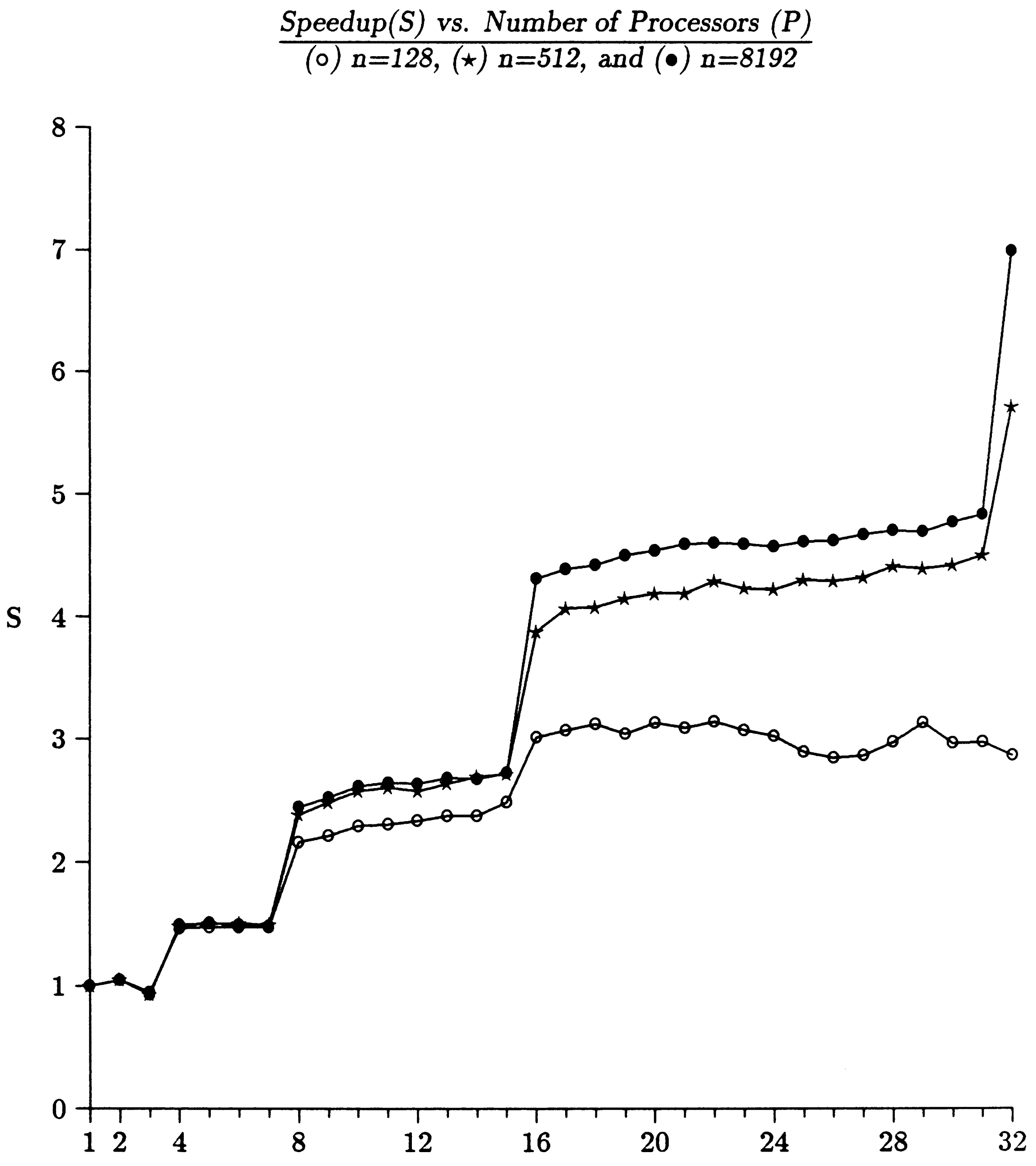


Figure 6.10: Speedup for 1 ... 32 processors.

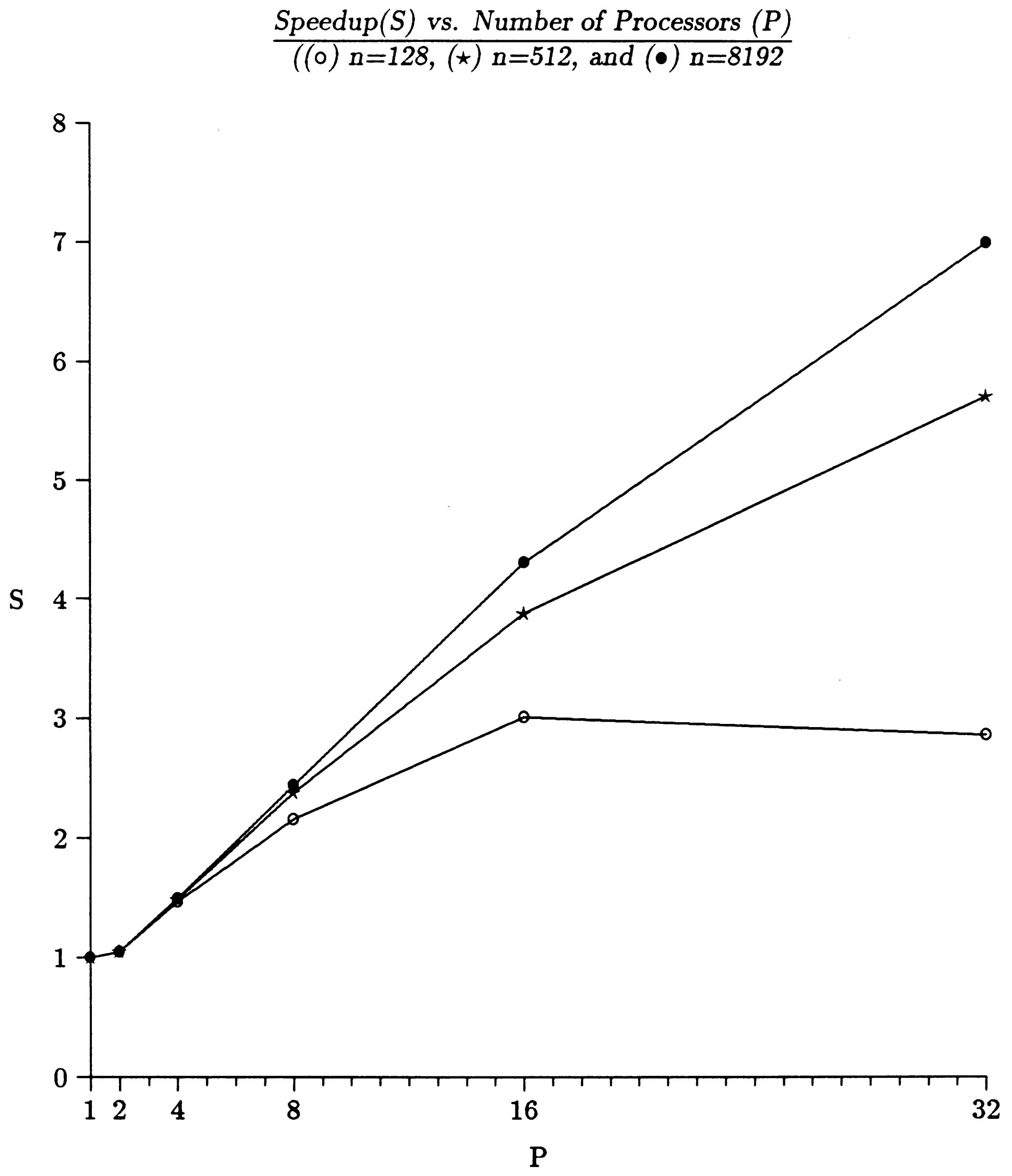


Figure 6.11: Speedup for 1 ,2, 4, 8, 16, and 32 processors.

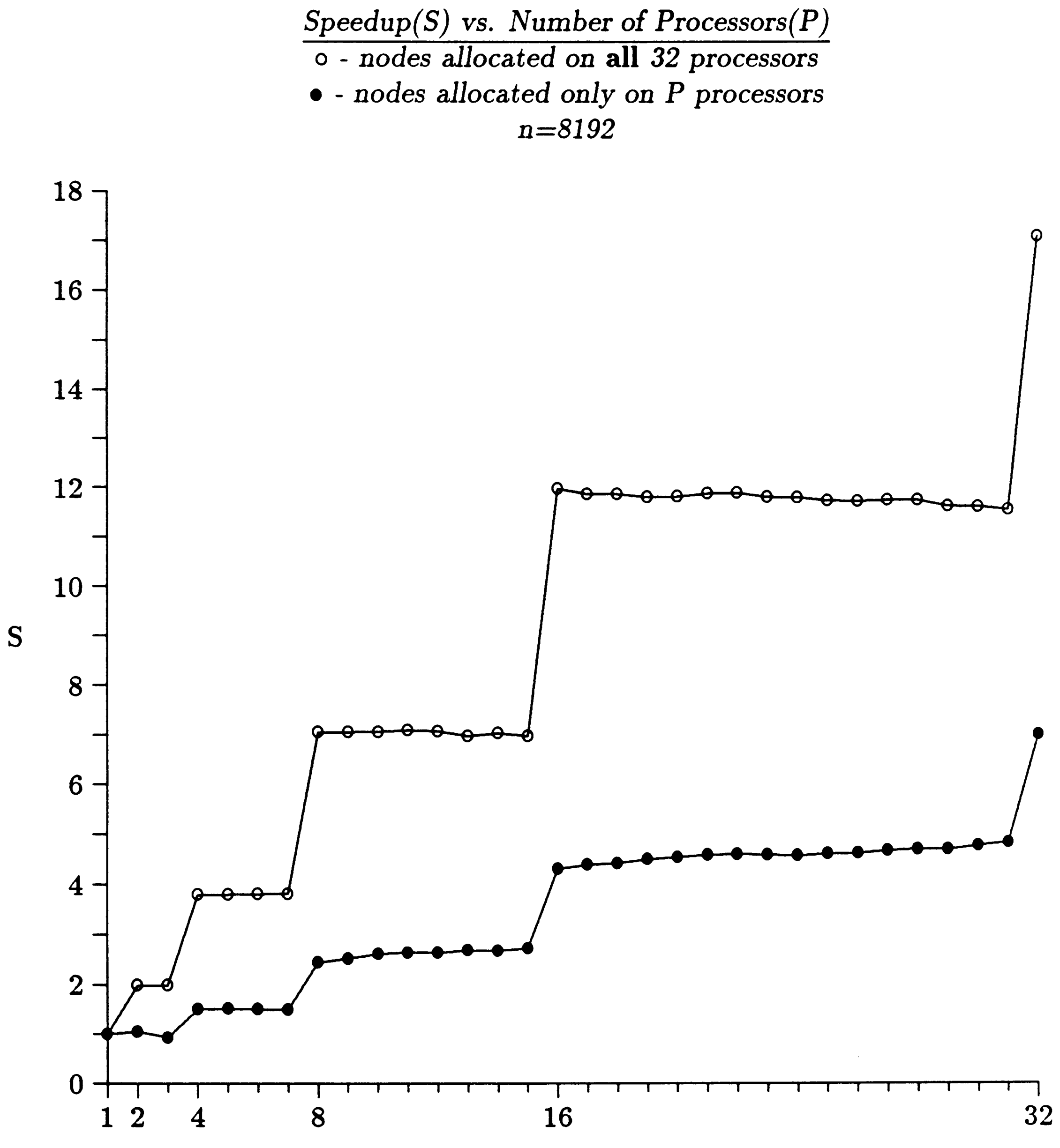


Figure 6.12: The effect of locality.

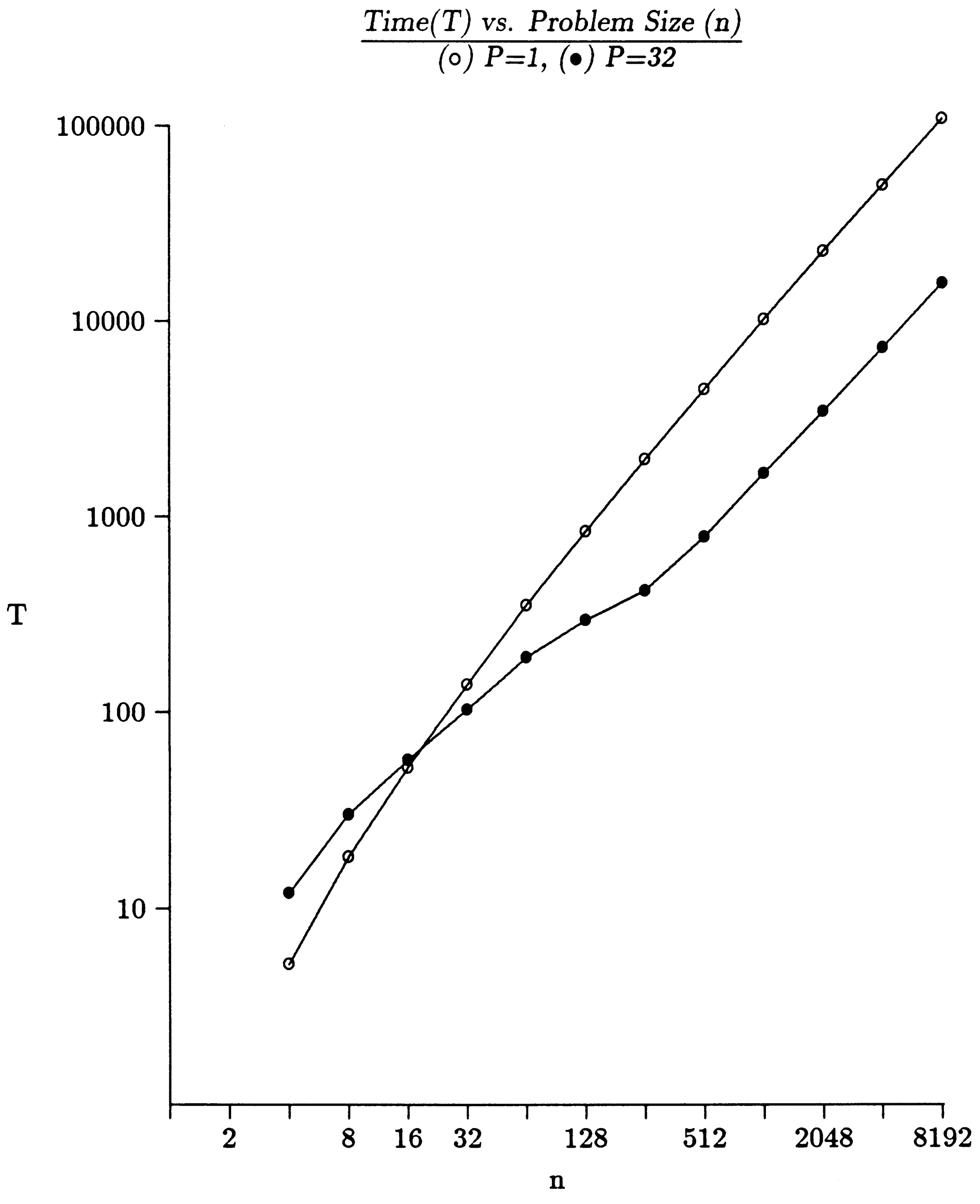


Figure 6.13: The effect of processor allocation overhead.

Chapter 7

Conclusions and Further Work

In this thesis, we have presented a new approach to interference analysis and parallelization of imperative programs with dynamic data structures. Our approach concentrates on providing compile-time analyses that approximate the relationships between accessible nodes in large aggregate data structures. These relationships are represented by path expressions, a restricted form of regular expressions, that provide both useful information, and efficient operations such as merging and equality testing. The relationships calculated by the interference analysis, encoded in path matrices, are used to determine that the data structures built by a program are in the special classes of *TREE* or *DAG*. In addition, the relationships are used for parallelization. We presented three parallelization methods: (1) a fine-grain analysis to determine if n basic handle statements can be executed in parallel, (2) a more coarse-grain analysis to determine if procedure calls can be executed in parallel, and (3) an analysis to detect if two statement sequences interfere.

We have provided both an informal description of the techniques and a formal model for describing the interference analysis as an abstract interpretation. Using the formal model we demonstrated soundness of the method with respect to a standard semantics.

By focusing our approach on data structures with regular properties, we were able to provide an efficient abstract representation (path matrices) for recursive data structures and we were able to develop effective parallelization methods based on the abstract representation. A prototype of the system dealing with *TREES* and *DAGS* is operational. We illustrated one use of the parallelization information by implementing a back-end that uses the coarse-grain parallelization information to generate parallel C programs for the BBN butterfly. As well, we provided numerous examples of the interference analysis, and we illustrated the effectiveness of our system with concrete examples.

The overall structure of the parallelizing system presented in this thesis is illustrated in figure 6.1. Although originally intended as an experiment in analysing programs with recursive data structures, we can view the system as providing a kernel of interference analysis and parallelization techniques. Our future work involves building upon this kernel to provide parallelization systems for languages less constrained than SIL. Consider the system outlined in figure 7.1. The top-level lists various programming languages that support recursively defined data structures, while the bottom-level lists some parallel architectures. Interference analysis, parallelization, and code generation/scheduling connect the two levels. If we view SIL as a high-level intermediate representation of C, then we have provided a major piece of the path from sequential C to parallel Butterfly C. However, there are many opportunities for extensions and further work at each level.

First consider the interference analysis level. We would like to express our analysis in terms of an intermediate representation that can be related to other languages. Some obvious extensions to the method include other forms of parameter passing (call-by-reference), and support for higher-order functions.

At the parallelization level we have provided three parallelization methods. We would like to extend the use of path matrices to develop more sophisticated parallelization techniques such as partial overlapping and pipelining of procedure calls, and pipelining of loop iterations.

At the code generation level, we have provided some basic techniques for exploiting coarse-grain parallelism on a shared memory machine. As outlined in chapter 6, there are still many issues to address. Scheduling the parallel computations in a balanced fashion, and maximizing locality of reference, are both important problems. As was the case with static estimation of interference, the dynamic nature of the size, shape, and connectivity of the data-structures makes static scheduling and exploiting locality particularly difficult.

Another important issue is programmer interaction at various levels. In the system outlined in this thesis, the programmer specified a sequential program, and the interference analysis, parallelization, and parallel code generation was performed by the compiler without further interaction with the programmer. We would like to investigate how the programmer can guide the parallelization. One possibility is to augment the programming language so that the programmer can specify additional information about the computation. Such an approach has been used in the context of domain decomposition for scientific programs using arrays [RP89, CK88, KMR87]. In these systems, the programmer specifies both the program and a mapping for the data, and the compiler produces a parallel program based on that mapping. By allowing the programmer to encode some knowledge about the problem (what mapping leads to good locality of reference), the compiler can generate efficient parallel programs. We would like to explore the possibilities of analogous mechanisms for programs with dynamic data structures. Is it possible to specify more information about a dynamic data structure so that the compiler can perform more precise interference analysis, improve locality of reference, or produce a better parallel schedule? Other levels of programmer interaction have been developed in the context of parallelizing FORTRAN compilers. Programming environments such as Faust [GGGJ88], PAT [SA88], Superb [ZBG88], Parafrase-2 [PGH+89], and ParaScope [BKK+89] allow the programmer to interact with the

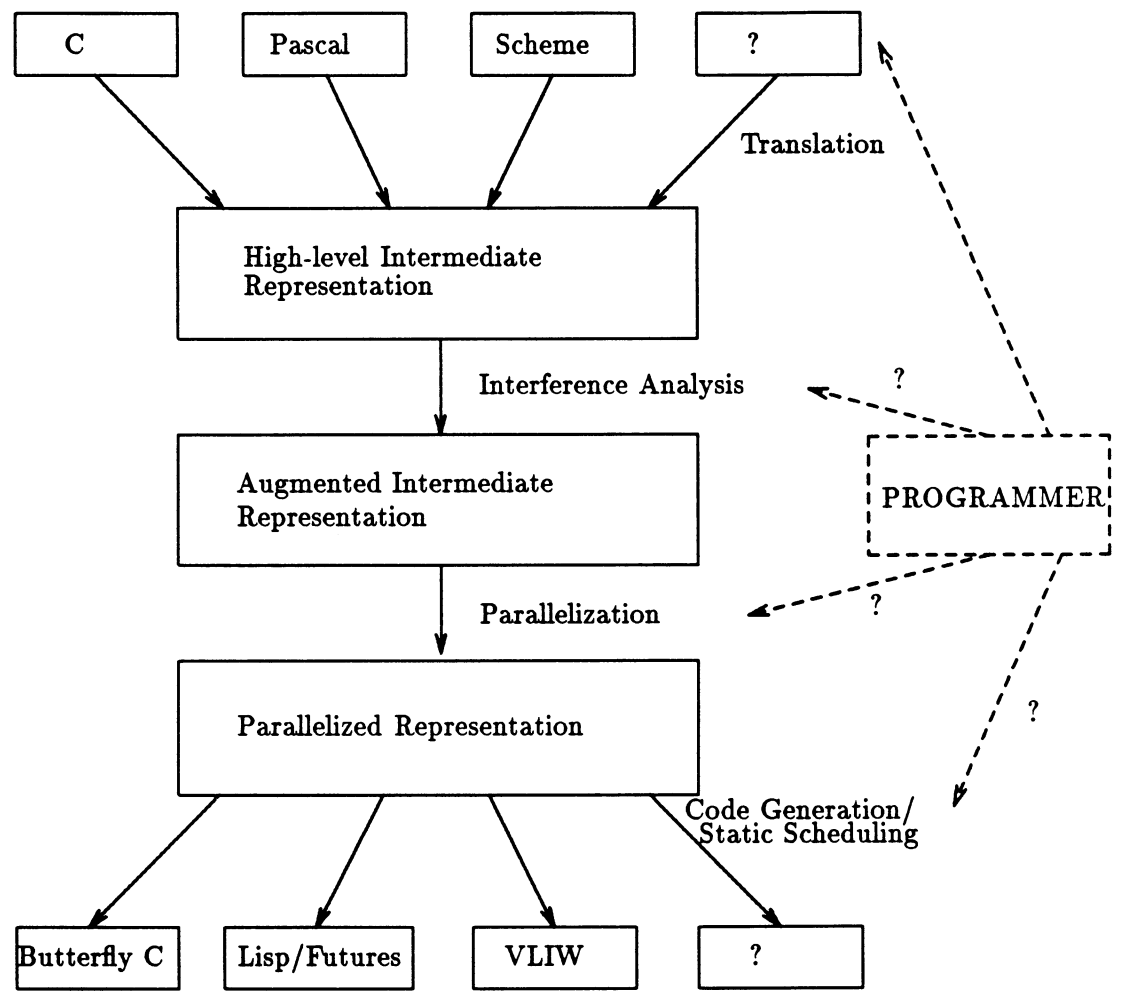

C
Pascal
Scheme
?
Translation
High-level Intermediate Representation
Interference Analysis
?
Augmented Intermediate Representation
PROGRAMMER
Parallelization
?
Parallelized Representation
?
Code Generation/ Static Scheduling
Butterfly C
Lisp/Futures
VLIW
?

Figure 7.1: Future Plans

interference analyzer and parallelizing transformation system. However, unlike the FORTRAN environments that deal mostly with loops and arrays, we need to determine what tools would be useful in the context of recursive data structures, and recursive programs. Programmer interaction with the scheduling strategy is one instance of potentially useful programmer input.

In summary, we have developed the core of an interference analysis and parallelization system for programs containing recursive data structures. Our initial experiments with the methods have been very encouraging and have suggested interesting new areas of research.

Appendix A

Standard Sematics for SIL

A.1 Semantic Domains

$$\begin{gathered}
\mathcal{E} : Expression \rightarrow (Env \rightarrow (Mem \rightarrow Value)) \\
\mathcal{M} : Statement \rightarrow (Env \rightarrow (Mem \rightarrow Mem)) \\
\mathcal{D} : Definition \rightarrow (Env \rightarrow Env) \\
\mathcal{P} : Program \rightarrow (File \rightarrow File + \underline{error})
\end{gathered}$$

where

$$\begin{gathered}
Env = Id \rightarrow Loc + Procedure \\
Store = Loc \rightarrow Value \\
Heap = NodePtr \rightarrow ((Int + \underline{undef}) \times (NodePtr + \underline{nil}) \times (NodePtr + \underline{nil})) \\
Value = Int + NodePtr + \underline{nil} + \underline{undef} \\
Procedure = Value^n \rightarrow (Mem \rightarrow Mem) \\
File = Int \textbf{ list} \\
NodePtr = Int \\
Loc = Int \\
Mem = (Store \times Heap \times Loc \times NodePtr \times File \times File) + \underline{error}
\end{gathered}$$

A.2 Expressions

$\mathcal{E}[\![\ constant \ ]\!] \ env \ mem = \underline{constant}$

$\mathcal{E}[\![\ X \ ]\!] \ env \ (store, heap, sp, fl, in, out) = store(env(X))$

```
$\mathcal{E}[\![ \ X.f \ ]\!] \ env \ (store, heap, sp, fl, in, out) =$
  $\underline{\textbf{let}}$
    $h = (store(env(X))$
  $\underline{\textbf{in}}$
    $heap(h).f$
```

```
$\mathcal{E}[\![\ e_1\ op\ e_2\ ]\!]\ env\ mem =$
  $\underline{\textbf{let}}$
    $v_1 = (\ \mathcal{E}[\![\ e_1\ ]\!]\ env\ mem\ )$
    $v_2 = (\ \mathcal{E}[\![\ e_2\ ]\!]\ env\ mem\ )$
  $\underline{\textbf{in}}$
    $\underline{\textbf{if}}\ v_1 = \underline{undef}$ or $v_2 = \underline{undef}\ \underline{\textbf{then}}$
      $\underline{undef}$
    $\underline{\textbf{else}}$
      $v_1\ \underline{op}\ v_2$
```

A.3 Scalar Statements

$\mathcal{M}[\![\ s\ ]\!]\ env\ \underline{error} = \underline{error}$

```
$\mathcal{M}[\![\ X := e\ ]\!]\ env\ (store,\ heap,\ sp,\ fl,\ in,\ out) =$
  $\underline{\textbf{let}}$
    $v = \mathcal{E}[\![e]\!]\ env\ (store,\ heap,\ sp,\ fl,\ in,\ out)$
  $\underline{\textbf{in}}$
    $(store[env(X) \mapsto v],\ heap,\ sp,\ fl,\ in,\ out)$
```

$\mathcal{M}[\![\ X := get()]\!]\ env\ (store,\ heap,\ sp,\ fl,\ in,\ out) =$
$\quad (store[env(X) \mapsto first(in)],\ heap,\ sp,\ fl,\ rest(in),\ out)$

```
$\mathcal{M}[\![\ put(e)\ ]\!]\ env\ (store,\ heap,\ sp,\ fl,\ in,\ out) =$
  $\underline{\textbf{let}}$
    $v = \mathcal{E}[\![\ e\ ]\!]\ env\ mem$
  $\underline{\textbf{in}}$
    $(store,\ heap,\ sp,\ fl,\ in,\ out::v)$
```

A.4 Handle Statements

$\mathcal{M}[\![\ A := nil\ ]\!]\ env\ (store,\ heap,\ sp,\ fl,\ in,\ out) =$
$\quad (store[env(A) \mapsto \underline{nil}],\ heap,\ sp,\ fl,\ in,\ out)$

$\mathcal{M}[\![\ A := new()\ ]\!]\ env\ (store,\ heap,\ sp,\ fl,\ in,\ out) =$
$\quad (store[env(A) \mapsto fl],\ heap[fl \mapsto (\underline{undef},\ \underline{nil},\ \underline{nil})],\ sp,\ fl{+}1,\ in,\ out)$

$\mathcal{M}[\![\ A := B\ ]\!]\ env\ (store,\ heap,\ sp,\ fl,\ in,\ out) =$
$\quad (store[env(A) \mapsto store(env(B))],\ heap,\ sp,\ fl,\ in,\ out)$

```
$\mathcal{M}[\![ A := B.f ]\!]\ env\ (store, heap, sp, fl, in, out) =$
  $\underline{\textbf{let}}$
    $h = store(env(B))$
  $\underline{\textbf{in}}$
    $\underline{\textbf{if}}\ h = \underline{nil}\ \underline{\textbf{then}}\ \underline{error}\ \underline{\textbf{else}}\ (store[env(A) \mapsto h.f], heap, sp, fl, in, out)$
```

```
$\mathcal{M}[\![ A.f := e ]\!]\ env\ (store, heap, sp, fl, in, out) =$
  $\underline{\textbf{let}}$
    $h = store(env(A))$
    $v = \mathcal{E}[\![ exp ]\!]\ env\ mem$
  $\underline{\textbf{in}}$
    $\underline{\textbf{if}}\ h = \underline{nil}\ \textbf{or}\ v = \underline{verror}\ \underline{\textbf{then}}$
      $\underline{error}$
    $\underline{\textbf{else}}$
      $(store, heap[h.f \mapsto v], sp, fl, in, out)$
```

A.5 Compound Statements

$\mathcal{M}[\![\textbf{begin end}]\!]\ env\ mem = mem$

```
$\mathcal{M}[\![ \textbf{begin}\ s_1 ; s_2 ; \ldots ; s_n\ \textbf{end} ]\!]\ env\ mem =$
  $\mathcal{M}[\![ \textbf{begin}\ s_2 ; \ldots ; s_n\ \textbf{end} ]\!]\ env\ (\ \mathcal{M}[\![ s_1 ]\!]\ env\ mem\ )$
```

```
$\mathcal{M}[\![ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 ]\!]\ env\ mem =$
  $\underline{\textbf{if}}\ \ \mathcal{E}[\![ e ]\!]\ env\ mem\ \underline{\textbf{then}}$
    $\mathcal{M}[\![ s_1 ]\!]\ env\ mem$
  $\underline{\textbf{else}}$
    $\mathcal{M}[\![ s_2 ]\!]\ env\ mem$
```

```
$\mathcal{M}[\![ \textbf{while}\ e\ \textbf{do}\ s ]\!]\ env\ mem = fix(W)\ mem$
  where
    $W = \lambda T\ .\ \lambda m\ .$
            $\underline{\textbf{if}}\ \mathcal{E}[\![ e ]\!]\ env\ m\ \underline{\textbf{then}}\ T(\ \mathcal{M}[\![ s ]\!]\ env\ m\ )\ \underline{\textbf{else}}\ m$
```

```
$\mathcal{M}[\![ \textbf{repeat}\ s\ \textbf{until}\ e ]\!]\ env\ mem = fix(W)\ mem$
  where
    $W = \lambda T.\ \lambda m\ .$
          $\underline{\textbf{let}}\ m_2 = \mathcal{M}[\![ s ]\!]\ env\ m\ \underline{\textbf{in}}$
            $\underline{\textbf{if}}\ \mathcal{E}[\![ e ]\!]\ env\ m_2\ \underline{\textbf{then}}\ m_2\ \underline{\textbf{else}}\ T(\mathcal{M}[\![ s ]\!]\ env\ m_2)$
```

A.6 Procedures

```
$\mathcal{M}[\![\ p(\ a_1\ ,\ a_2, \ldots ,\ a_n\ )\ ]\!]\ env\ mem =$
  $\underline{\textbf{let}}$
    $v_1 =\ \ \mathcal{E}[\![\ a_1\ ]\!]\ env\ mem$
    $\ldots$
    $v_n =\ \ \mathcal{E}[\![\ a_n\ ]\!]\ env\ mem$
  $\underline{\textbf{in}}$
    $env(p)(v_1 \ldots v_n)\ mem$

$\mathcal{D}[\![$ proc $q$ ( $x_1{:}t$ ; ... ; $x_n{:}t$ ) $l_1{:}t$ ; ... ; $l_m{:}t$ ; $s$ $]\!]\ env = env\ [q \mapsto Q]$
  where $Q =$
    $\lambda v_1\ .\ \lambda v_2\ .\ \ldots .\ \lambda v_n\ .\ \lambda\ (store,heap,sp,fl,in,out)\ .$
      $\underline{\textbf{let}}$
        $local\_env =$
          $env[\ x_1 \mapsto sp+1, \ldots ,\ x_n \mapsto sp+n,$
                $l_1 \mapsto sp+n+1, \ldots ,\ l_n \mapsto sp+n+m\ ]$
        $local\_store =$
              $store[local\_env(x_1) \mapsto v_1, \ldots, local\_env(x_n) \mapsto v_n,$
                    $local\_env(l_1) \mapsto \underline{undef}, \ldots, local\_env(l_m) \mapsto \underline{undef}\,]$
        $new\_mem = \mathcal{M}[\![\ s\ ]\!]\ local\_env\ (local\_store,\ heap,\ sp+m+n,\ fl,\ in,\ out)$
      $\underline{\textbf{in}}$
        $\underline{\textbf{if}}\ new\_mem = \underline{error}\ \underline{\textbf{then}}$
          $\underline{error}$
        $\underline{\textbf{else}}$
          $\underline{\textbf{let}}$
            $(local\_store',heap',sp',fl',in',out') = new\_mem$
          $\underline{\textbf{in}}$
            $(store,heap',sp,fl',in',out')$

$\mathcal{D}[\![$ recproc $r$ ( $x_1{:}t$ ; ... ; $x_n{:}t$ ) $l_1{:}t$ ; ... ; $l_m{:}t$ ; $s$ $]\!]\ env = fix(R)\ env$
  where
    $R = \lambda T\ .\ \lambda e\ .\ e[r \mapsto Q]$
    $Q = \lambda v_1\ .\ \lambda v_2\ .\ \ldots .\ \lambda v_n\ .\ \lambda\ (store,heap,sp,fl,in,out)\ .$
      $\underline{\textbf{let}}$
        $local\_env =$
          $(T\ e)[\ x_1 \mapsto sp+1, \ldots ,\ x_n \mapsto sp+n,$
                $l_1 \mapsto sp+n+1, \ldots ,\ l_m \mapsto sp+n+m\ ]$
        $local\_store =$
          $store[local\_env(x_1) \mapsto v_1, \ldots, local\_env(x_n) \mapsto v_n,$
                $local\_env(l_1) \mapsto \underline{undef}, \ldots, local\_env(l_m) \mapsto \underline{undef}\,]$
        $new\_mem =$
          $\mathcal{M}[\![\ s\ ]\!]\ local\_env\ (local\_store,\ heap,\ sp+n+m,\ fl,\ in,\ out)$
    in
      if new_mem = error then
        error
      else
        let
        (local_store',heap',sp',fl',in',out') = new_mem
        in
          (store,heap',sp,fl',in',out')
```

```
$\mathcal{D}[\![ d_1; d_2; \ldots; d_n ]\!]\ env =$
    $\mathcal{D}[\![ d_2; \ldots; d_n ]\!]\ newenv$
      where $newenv = \mathcal{D}[\![ d_1 ]\!]\ env$
```

A.7 Programs

```
$\mathcal{P}[\![$ program $id\ def\_list ]\!]\ in\_file =$
  let
    final_mem =
      $\mathcal{M}[\![ main() ]\!]$
        ( $\mathcal{D}[\![ def\_list ]\!]\ empty\_env$ )
        ( empty_store,empty_heap,0,0,in_file,[] )
  in
    if final_mem = error then
      error
    else
        final_mem.out_file
```

Appendix B

Translating SIL to C

In this appendix, we outline the correspondence between SIL and C. The following sections give the SIL-to-C translations for node type definitions, statements, and procedures and functions.

B.1 Node Type Definition

nodedef
  s_1 : **int**;
  ...
  s_n : **int**;
  h_1 : **handle**;
  ...
  h_m : **handle**
end
	`struct node {`
	  `int s1;`
	  `...`
	  `int sn;`
	  `struct node *h1;`
	  `...`
	  `struct node *hm;`
	  `};`
	
	`typedef struct node HANDLE;`
	
	`#define NIL ((HANDLE *) 0)`

B.2 Statements

$a := b$	`a = b`
$a := b.f$	`a = b->f`
$a.f := b$	`a->f = b`
begin	`{ s1;`
s_1;	`s2;`
s_2;	`...`
...	`sn;`
s_n	`}`
end	
if *exp* **then**	`if (exp)`
s_1	`s1`
else	`else`
s_2	`s2`
while *exp* **do**	`while (exp)`
s_1	`s1`
repeat	`do`
s_1	`s1`
until *exp*	`while (! exp)`

B.3 Procedures and Functions

The following rules give the translations for procedures, functions returning one value, and functions return more that one value. Note that functions returning one value map directly to C functions, while functions returning more than one value in SIL are translated to C procedures with reference parameters.

$p(x_1,x_2,\ldots,x_n)$	`p(x1,x2, ... ,xn)`
$r_1 := f(x_1,x_2,\ldots,x_n)$	`r1 = f(x1,x2, ... ,xn)`
$r_1, \ldots, r_m := f(x_1,x_2,\ldots,x_n)$	`f(x1,x2, ... ,xn,&r1, ... ,&rm)`

procedure $p(x_1{:}t_1,\ldots,x_n{:}t_n)$
  l_1: lt_1;
  ...
  l_m: lt_m
begin
  s_1;
  ...
  s_t
end
	`void p(x1,x2, ... ,xn)`
	`  t1 x1; ...; tn xn;`
	`{ lt1 l1;`
	`  ...`
	`  ltm lm;`
	
	`  s1;`
	`  ...`
	`  st;`
	`}`

function $f(x_1{:}t_1,\ldots,x_n{:}t_n)$ rt
  l_1: lt_1;
  ...
  l_m: lt_m
begin
  s_1;
  ...
  s_t
**end => return**(r)
	`rt f(x1,x2, ... ,xn)`
	`  t1 x1; ...; tn xn;`
	`{ lt1 l1;`
	`  ...`
	`  ltm lm;`
	
	`  s1;`
	`  ...`
	`  st;`
	`  return(r);`
	`}`

function $f(x_1{:}t_1,\ldots,x_n{:}t_n)$
            $rt_1, \ldots, rt_q$
  l_1: lt_1;
  ...
  l_m: lt_m
begin
  s_1;
  ...
  s_t
**end => return**($r_1, \ldots, r_q$)
	`void f(x1,x2, ... ,xn,`
	`                r_1, ..., r_q)`
	`  t1 x1; ...; tn: xn;`
	`  tr_1 *r_1; ...; tr_q *r_q;`
	`{ lt1: l1;`
	`  ...`
	`  ltm: lm;`
	
	`  s1;`
	`  ...`
	`  st;`
	
	`  *r_1 = r1;`
	`  ...`
	`  *r_q = rq;`
	`}`

Appendix C

Illustrative Interference Analysis Computations

Our interference analysis tools include a trace facility[1] for path matrix computations. In this appendix we present portions of traces to illustrate the analysis for a wide variety of program constructs.

C.1 SIL program

The following program defines a linked list node type, and defines functions for building, sorting, and printing lists.

```
program quicksort
constant
  RANDOMDIV = 100000
end;
nodedef
  value: int;
  next: handle
end;
function new_node(v: int) handle
{ Allocate a new node, initialize value field to v,   }
{   link field to nil }
  h: handle
begin
  h := new();
  h.next := nil;
  h.value := v
```

[1]Including a latex mode which is useful for thesis writing.

```
end => return(h);

function build_nonnil_list(n:int)   handle
{ build a random list of at least one element }
  next_val: int;
  root, last_node, next_node : handle
begin
  next_val := random();
  next_val := next_val / RANDOMDIV;
  root := new_node(next_val);
  last_node := root;
  while n > 1 do
    begin
      next_val := random();
      next_val := next_val / RANDOMDIV;
      next_node := new_node(next_val);
      last_node.next := next_node;
      last_node := next_node;
      n := n - 1
    end;
end => return(root);

function build_list(n:int) handle
{ build a random list }
  list: handle
begin
  if n > 0 then
    list := build_nonnil_list(n)
  else
    list := nil
end => return(list);

procedure print_list(h: handle)
{ print a linked list }
begin
  while h ≠ nil do
    begin
      put(h.value);
      h := h.next
    end;
  puts("\n");
end;

function link_up(a_head,a_tail,b_head,b_tail:handle) handle, handle
```

```
{ given two lists with head and tail pointers, }
{ link them together and return the head and tail  }
{ pointers to linked result }
  res_head, res_tail: handle
begin
  if a_head = nil then
    begin
      res_head := b_head;
      res_tail := b_tail
    end
  else if b_head = nil then
    begin
      res_head := a_head;
      res_tail := a_tail
    end
  else
    begin
      a_tail.next := b_head;
      res_head := a_head;
      res_tail := b_tail
    end
end => return(res_head,res_tail);

function partition(h:handle) handle,handle,handle,handle
{ partition a non-nil list into  three pieces, }
{ return pointer to head of left list (< v), head and tail of  }
{ middle list (= v), and head of right list (> v) }
  left_h,left_t, p_h, p_t, right_h, right_t: handle;
  next_node, last_node: handle;
  v: int
begin
  { h is first item of list, next_node is rest, v is partition value }
  v := h.value;
  next_node := h.next;
  h.next := nil;

  { initialize left list to empty, partition list to first item, }
  { right list to empty }
  left_h := nil; left_t := nil;
  p_h := h; p_t := h;
  right_h := nil; right_t := nil;

  { partition into three lists }
  while next_node ≠ nil do
    begin
```

```
        { chop head of list off }
        last_node := next_node;
        next_node := next_node.next;
        last_node.next := nil;
        { append to correct list, left, partition, or right }
        if last_node.value < v then
          if left_h = nil then
            begin
              left_h := last_node;
              left_t := last_node
            end
          else
            begin
              left_t.next := last_node;
              left_t := last_node
            end
        else if last_node.value > v then
          if right_h = nil then
            begin
              right_h := last_node;
              right_t := last_node;
            end
          else
            begin
              right_t.next := last_node;
              right_t := last_node
            end
        else
          begin
            p_t.next := last_node;
            p_t := last_node
          end
      end
  end => return (left_h,p_h,p_t,right_h);

  function qsort(h:handle) handle, handle
  { given a pointer to unsorted list, return head and tail pointer of }
  { sorted list }
    q_h, q_t: handle;
    l,l_h,l_t,r,r_h,r_t: handle;
    p_h, p_t: handle
  begin
    if h = nil then
```

```
        begin
          q_h := nil;
          q_t := nil
        end
      else
        begin
          l,p_h,p_t,r := partition(h);
          l_h,l_t := qsort(l);
          r_h,r_t := qsort(r);
          q_h,q_t := link_up(l_h,l_t,p_h,p_t);
          q_h,q_t := link_up(q_h,q_t,r_h,r_t)
        end
    end => return(q_h,q_t);

    procedure main(n:int)
      unsorted_head,sorted_head,sorted_tail : handle
    begin
      unsorted_head := build_list(n);
      print_list(unsorted_head);
      sorted_head, sorted_tail := qsort(unsorted_head);
      print_list(sorted_head)
    end;
```

C.2 Analysis of the main program

We first look at the interference analysis at the level of the main procedure. For each statement in the body of *main*, we display the path matrix that estimates the effect of executing the statement.

```
    procedure main(n:int)
      unsorted_head,sorted_head,sorted_tail : handle
    begin
      unsorted_head := build_list(n);
      print_list(unsorted_head);
      sorted_head, sorted_tail := qsort(unsorted_head);
      print_list(sorted_head)
    end;
```

```
==================== BEGIN main ====================

begin ... end
unsorted_head:=build_list(n)
```

	unsorted_head$(\odot,\odot)$
unsorted_head	S

```
print_list(unsorted_head)
```

	unsorted_head$(\odot,\odot)$
unsorted_head	S

```
sorted_head,sorted_tail:=qsort(unsorted_head)
```

	sorted_head$(\odot,\odot)$	*sorted_tail*$(\odot,\circ)$
sorted_head	S	$(S+N^+)?$
sorted_tail	$S?$	S

```
print_list(sorted_head)
```

□

```
=================== END main ===================
```

Given the trace of *main* above , we observe the following:

`unsorted_head := build_list(n)` - This statement produces a list that does not contain any cycles or *DAG* nodes. If a cycle had been detected an exception would have been raised, and if a *DAG* node had been created, a *DAG* handle would appear in the resulting path matrix. The nilness estimate of $(\odot,\odot)$ indicates that the handle *unsorted_head* may be either **nil** or a handle to a node.

`print_list(unsorted_head)` - This statement does not change the structure of the list, and the path matrix estimate remains the same.

`sorted_head, sorted_tail := qsort(unsorted_head)` - After executing this statement, two handles are live: *sorted_head*, and *sorted_tail*. No cycles or *DAGS* were created, and the path matrix entries indicate that *sorted_head* and *sorted_tail* may be the same node, or there may be a path of length 1 or more between them. Also note that the analysis detects that the *next* field for *sorted_tail* must be **nil**.

`print_list(sorted_tail)` - After executing this statement, there are no live handles, so an empty path matrix is displayed.

C.3 Analysis of *build_list*

We now take a closer look at the analysis of the *build_list* and *build_nonnil_list* functions, and the resulting trace of the path matrix computations. This trace illustrates the iterative approximation of a the while loop in *build_nonnil_list*, and the merging of path matrices after conditionals. For example, note the merging of the path matrices for both parts of the conditional statement in *build_list*.

```
function build_list(n:int) handle
{ build a random list }
  list: handle
begin
  if n > 0 then
    list := build_nonnil_list(n)
  else
    list := nil
end => return(list);

function build_nonnil_list(n:int)  handle
{ build a random list of at least one element }
  next_val: int;
  root, last_node, next_node : handle
begin
  next_val := random();
  next_val := next_val / RANDOMDIV;
  root := new_node(next_val);
  last_node := root;
  while n > 1 do
    begin
      next_val := random();
      next_val := next_val / RANDOMDIV;
      next_node := new_node(next_val);
      last_node.next := next_node;
      last_node := next_node;
      n := n - 1
    end;
end => return(root);
```

```
================= BEGIN build_list ==================
begin ... end
if (n>0) then ... else ...
================= COND (THEN PART) =================
list:=build_nonnil_list(n)
================= BEGIN build_nonnil_list ==================
begin ... end
next_val:=random()
```

```
next_val := (next_val/100000)
root:=new_node(next_val)
```

	root(●,○)
root	*S*

```
last_node := root
```

	root(●,○)	*last_node*(●,○)
root	*S*	*S*
last_node	*S*	*S*

```
while (n>1) do ....
=============== BEGIN WHILE/REPEAT  =============
begin ... end
next_val:=random()
next_val := (next_val/100000)
next_node:=new_node(next_val)
```

	root(●,○)	*last_node*(●,○)	*next_node*(●,○)
root	*S*	*S*	
last_node	*S*	*S*	
next_node			*S*

```
last_node.next := next_node
```

	root(●,●)	*next_node*(●,○)
root	*S*	N^1
next_node		*S*

```
last_node := next_node
```

	root(●,●)	*last_node*(●,○)
root	*S*	N^1
last_node		*S*

```
n := (n-1)

------- results of iteration #1 -------

IN:
```

	root(●,○)	*last_node*(●,○)
root	*S*	*S*
last_node	*S*	*S*

```
OUT:
```

	root(•,•)	*last_node*(•,∘)
root	S	N^1
last_node		S

```
=================== ITERATION # 2 ==================
begin ... end
next_val:=random()
next_val := (next_val/100000)
next_node:=new_node(next_val)
```

	root(•,•)	*last_node*(•,∘)	*next_node*(•,∘)
root	S	N^1	
last_node		S	
next_node			S

```
last_node.next := next_node
```

	root(•,•)	*next_node*(•,∘)
root	S	N^2
next_node		S

```
last_node := next_node
```

	root(•,•)	*last_node*(•,∘)
root	S	N^2
last_node		S

```
n := (n-1)

------- results of iteration #2 -------

IN:
```

	root(•,•)	*last_node*(•,∘)
root	S	N^1
last_node		S

```
OUT:
```

	root(•,•)	*last_node*(•,∘)
root	S	N^2
last_node		S

```
IN merge OUT => next(IN):
```

	root(•,•)	*last_node*(•,∘)
root	S	N^+
last_node		S

```
=================== ITERATION # 3 ==================
begin ... end
next_val:=random()
next_val := (next_val/100000)
next_node:=new_node(next_val)
```

	root(•,•)	*last_node(•,∘)*	*next_node(•,∘)*
root	S	N^+	
last_node		S	
next_node			S

```
last_node.next := next_node
```

	root(•,•)	*next_node(•,∘)*
root	S	N^1N^+
next_node		S

```
last_node := next_node
```

	root(•,•)	*last_node(•,∘)*
root	S	N^1N^+
last_node		S

```
n := (n-1)

------- results of iteration #3 -------

IN:
```

	root(•,•)	*last_node(•,∘)*
root	S	N^+
last_node		S

```
OUT:
```

	root(•,•)	*last_node(•,∘)*
root	S	N^1N^+
last_node		S

```
IN merge OUT => next(IN):
```

	root(•,•)	*last_node(•,∘)*
root	S	N^+
last_node		S

```
================== END  WHILE/REPEAT =================
```

	root(•,⊙)	*last_node*(•,○)
root	S	$(S+N^+)$
last_node	$S?$	S

```
================= END build_nonnil_list =================
```

	list(•,⊙)
list	S

```
================ (ELSE PART) =====================

list := nil
```

	list(○,○)
list	S

```
=================== END COND ================
```

	list(⊙,⊙)
list	S

```
================= END build_list =================
```

C.4 Analysis of *qsort*

We now examine the analysis for *qsort*. Figure C.1 gives a pictorial outline of the behaviour of *qsort* and figure C.2 gives a trace of the iterative approximation. Given an unsorted linked list *h*, *qsort* partitions *h* into three sub-lists, sorts the the left and right sub-lists, and then links the sorted sub-lists together.

```
function qsort(h:handle) handle, handle
{ given a pointer to unsorted list, return head and tail pointer of }
{ sorted list }
  q_h, q_t: handle;
  l,l_h,l_t,r,r_h,r_t: handle;
  p_h, p_t: handle
begin
  if h = nil then
    begin
      q_h := nil;
      q_t := nil
    end
  else
```

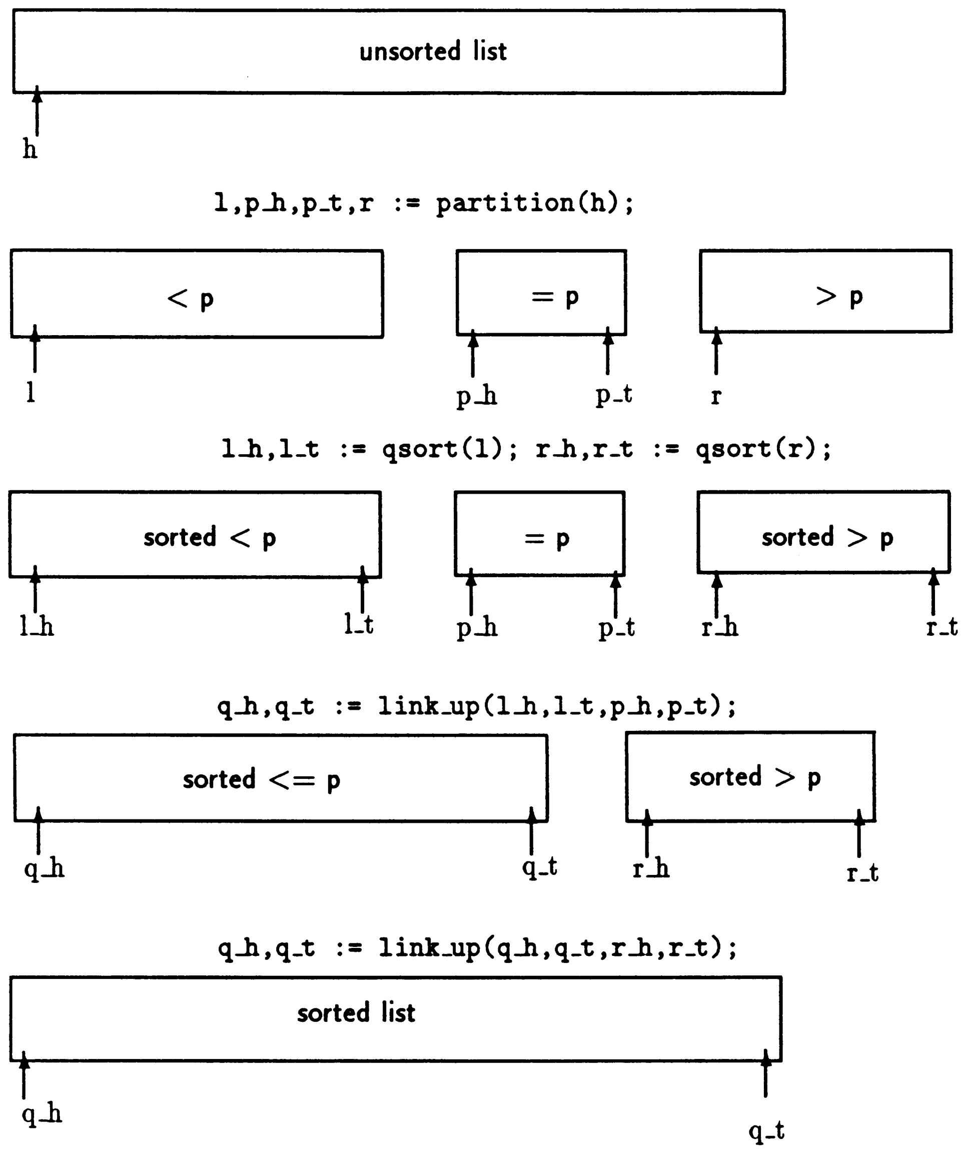

unsorted list
h
l,p_h,p_t,r := partition(h);
< p
= p
> p
l
p_h
p_t
r
l_h,l_t := qsort(l); r_h,r_t := qsort(r);
sorted < p
= p
sorted > p
l_h
l_t
p_h
p_t
r_h
r_t
q_h,q_t := link_up(l_h,l_t,p_h,p_t);
sorted <= p
sorted > p
q_h
q_t
r_h
r_t
q_h,q_t := link_up(q_h,q_t,r_h,r_t);
sorted list
q_h
q_t

Figure C.1: Diagrammatic outline for *qsort*.

```
************* [ 1 ] qsort - LEVEL 0 *****************

INPUT APPROX:
```

	h*6(⊙,⊙)	h(⊙,⊙)
h*6	S	S
h	S	S

```
OUTPUT APPROX:
```

	h*6(⊙,⊙)	q_h(∘,∘)	q_t(∘,∘)
h*6	S		
q_h		S	
q_t			S

```
************* [ 2 ] qsort - LEVEL 0 *****************

INPUT APPROX:
```

	h*6(⊙,⊙)	h(⊙,⊙)
h*6	S	S
h	S	S

```
OUTPUT APPROX:
```

	h*6(⊙,⊙)	q_h(⊙,⊙)	q_t(⊙,∘)
h*6	S	$S?$	$(S+N^+)?$
q_h	$S?$	S	$(S+N^+)$
q_t	$S?$	$S?$	S

```
************* [ 3 ] qsort - LEVEL 0 *****************

INPUT APPROX:
```

	h*6(⊙,⊙)	h(⊙,⊙)
h*6	S	S
h	S	S

```
OUTPUT APPROX:
```

	h*6(⊙,⊙)	q_h(⊙,⊙)	q_t(⊙,∘)
h*6	S	$S?$	$(S+N^+)?$
q_h	$(S+N^+)?$	S	$(S+N^+)?$
q_t	$S?$	$S?$	S

Figure C.2: Iterative Approximation for qsort.

```
    begin
      l,p_h,p_t,r := partition(h);
      l_h,l_t := qsort(l);
      r_h,r_t := qsort(r);
      q_h,q_t := link_up(l_h,l_t,p_h,p_t);
      q_h,q_t := link_up(q_h,q_t,r_h,r_t)
    end
  end => return(q_h,q_t);
```

The following trace of *qsort* gives the path matrices computed for the last iteration the iterative approximation. Compare the relationships encoded in the path matrices with the diagram of pointers given in figure C.1. Note that the path matrix just before the recursive calls to *qsort* indicates that *l* and *r* are unrelated. Also, note that the resulting path matrix is equal to the last output approximation in the iterative approximation in figure C.2.

```
================ BEGIN qsort ======================

begin ... end
if (h=nil) then ... else ...

================= COND (THEN PART) =================
begin ... end
q_h := nil
```

	$h*6(\odot,\odot)$	$q\_h(\circ,\circ)$
$h*6$	S	
q_h		S

```
q_t := nil
```

	$h*6(\odot,\odot)$	$q\_h(\circ,\circ)$	$q\_t(\circ,\circ)$
$h*6$	S		
q_h		S	
q_t			S

```
================ (ELSE PART) ======================
begin ... end
l,p_h,p_t,r:=partition(h)
```

	$h*6(\odot,\odot)$	$l(\odot,\odot)$	$r(\odot,\odot)$	$p\_h(\bullet,\odot)$	$p\_t(\bullet,\circ)$
$h*6$	S			$S?$	$(S+N^+)?$
l		S			
r			S		
p_h	$S?$			S	$(S+N^+)$
p_t	$S?$			$S?$	S

```
l_h,l_t:=qsort(l)
```

	$h*6$ $(\odot,\odot)$	l_h $(\odot,\odot)$	l_t $(\odot,\circ)$	r $(\odot,\odot)$	p_h $(\bullet,\odot)$	p_t $(\bullet,\circ)$
$h*6$	S				$S?$	$(S+N^+)?$
l_h		S	$(S+N^+)?$			
l_t		$S?$	S			
r				S		
p_h	$S?$				S	$(S+N^+)$
p_t	$S?$				$S?$	S

```
r_h,r_t:=qsort(r)
```

	$h*6$ $(\odot,\odot)$	l_h $(\odot,\odot)$	l_t $(\odot,\circ)$	r_h $(\odot,\odot)$	r_t $(\odot,\circ)$	p_h $(\bullet,\odot)$	p_t $(\bullet,\circ)$
$h*6$	S					$S?$	$(S+N^+)?$
l_h		S	$(S+N^+)?$				
l_t		$S?$	S				
r_h				S	$(S+N^+)?$		
r_t				$S?$	S		
p_h	$S?$					S	$(S+N^+)$
p_t	$S?$					$S?$	S

```
q_h,q_t:=link_up(l_h, l_t, p_h, p_t)
```

	$h*6(\odot,\odot)$	$q\_h(\odot,\odot)$	$q\_t(\odot,\circ)$	$r\_h(\odot,\odot)$	$r\_t(\odot,\circ)$
$h*6$	S	$S?$	$(S+N^+)?$		
q_h	$S?,(S+N^+)?$	S	$(S+N^+)?$		
q_t	$S?$	$S?$	S		
r_h				S	$(S+N^+)?$
r_t				$S?$	S

```
q_h,q_t:=link_up(q_h, q_t, r_h, r_t)
```

	$h*6(\odot,\odot)$	$q\_h(\odot,\odot)$	$q\_t(\odot,\circ)$
$h*6$	S	$S?$	$(S+N^+)?$
q_h	$S?,(S+N^+)?$	S	$(S+N^+)?$
q_t	$S?$	$S?$	S

```
==================== END COND =================
```

	$h*6(\odot,\odot)$	$q\_h(\odot,\odot)$	$q\_t(\odot,\circ)$
$h*6$	S	$S?$	$(S+N^+)?$
q_h	$(S+N^+)?$	S	$(S+N^+)?$
q_t	$S?$	$S?$	S

```
================= END qsort =================
```

Appendix D

Generating Parallel C Programs: An Example

D.1 Sequential SIL Program

```
program bitonic

constant
  UP = 0;      { sort in ascending order }
  DOWN = 1      { sort in descending order }
end;

nodedef
  value: int;
  left: handle;
  right: handle
end;

{ --- AUXILLIARY PROCEDURES --- }

function NewNode(v: int) handle
  h: handle
begin
  h := new();
  h.value := v;
  h.left := nil;
  h.right := nil
end => return(h);

{ --- PRINT TREE PROCEDURES --- }

procedure PrintIndent(n: int; path: int)
  divisor: int
begin
  if n = 0 then puts("x"); { if the root then print out root symbol }
  divisor := 1 << (n - 1);
  while n > 0 do
    begin
      if n = 1 then { if last link }
        if path = 0 then
          puts("/------>")  { right link }
        else
          puts("\------>") { left link }
      else
        if (path / divisor) =  (path
          puts("        ")
        else
          puts("|       ");
      path := path
      divisor := divisor / 2;
      n := n - 1
    end
end;

procedure PrintSubTree(h: handle; n,path: int)
  l,r: handle
begin
  if h ≠ nil then
    begin
      { print right sub-tree }
      r := h.right;
      PrintSubTree(r,n+1,path × 2);

      { print root }
      puts("("); put(n); puts(")");
      PrintIndent(n,path);
      puts(" "); put(h.value); puts("\n");

      { print left sub-tree }
      l := h.left;
      PrintSubTree(l,n+1,(path × 2) + 1)
    end
end;

procedure PrintTree(h: handle)
{ Print the tree rooted at h }
begin
  PrintSubTree(h,0,0)
end;

{ Print the inorder traversal of tree rooted at h }
```

```
procedure InOrder(h: handle)
  l,r: handle
begin
  if h ≠ nil then
    begin
      l := h.left;
      r := h.right;
      InOrder(l);
      put(h.value);
      puts(" ");
      InOrder(r)
    end
end;
{ --- CREATE A RANDOM TREE --- }
function RandTree(n: int) handle
  next_val: int;
  h,l,r: handle
begin
  if n > 1 then
    begin
      next_val := random();
      h := NewNode(next_val);
      l := RandTree(n/2);
      r := RandTree(n/2);
      h.left := l;
      h.right := r;
    end
  else
    h := nil
end => return(h);
{ --- SWAPPING PROCEDURES --- }
procedure SwapValue(l,r: handle)
  temp: int
begin
  temp := l.value;
  l.value := r.value;
  r.value := temp
end; { SwapValue }
procedure SwapLeft(l,r: handle)
  ll,rl: handle
begin
```

```
  ll := l.left;
  rl := r.left;
  l.left := rl;
  r.left := ll
end; { SwapLeft }
procedure SwapRight(l,r: handle)
  lr,rr: handle
begin
  lr := l.right;
  rr := r.right;
  l.right := rr;
  r.right := lr
end; { SwapRight }
{ --- BITONIC MERGE --- }
procedure Bimerge(root: handle; sp_r:handle ; dir: int)
  rightexchange, elementexchange, temp : int;
  sp_l, pl, pr, rl, rr: handle
begin
  rightexchange := (root.value > sp_r.value) xor dir;
  if rightexchange then SwapValue(root,sp_r);
  pl := root.left;
  pr := root.right;
  while (pl ≠ nil) do
    begin
      elementexchange := (pl.value > pr.value) xor dir;
      if rightexchange then
        if elementexchange then
        { swap values and right subtrees, search path goes left }
          begin
            SwapValue(pl,pr);
            SwapRight(pl,pr);
            pl := pl.left;
            pr := pr.left
          end
        else
        { search path goes right }
          begin
            pl := pl.right;
            pr := pr.right
          end
      else
        if elementexchange then
        { swap values and left subtrees, search path goes right }
          begin
            SwapValue(pl,pr);
            SwapLeft(pl,pr);
            pl := pl.right;
            pr := pr.right
          end
        else
        { search path goes left }
          begin
            pl := pl.left;
            pr := pr.left
          end
    end; { while }
  if (root.left ≠ nil) then
    begin
      rl := root.left;
      rr := root.right;
      sp_l := NewNode(root.value);
      Bimerge(rl,sp_l,dir);   Bimerge(rr,sp_r,dir);
      root.value := sp_l.value
    end
end; { Bimerge }
{ --- BITONIC SORT --- }
procedure Bisort(root,sp_r: handle; dir:int)
  l, r, sp_l: handle
begin
  if root.left = nil then
    begin
      if (root.value > sp_r.value) xor dir then
        SwapValue(root,sp_r)
    end
  else
    begin
      l := root.left;
      r := root.right;
      sp_l := NewNode(root.value);
      Bisort(l,sp_l,dir);   Bisort(r,sp_r,!dir);
      root.value := sp_l.value;
      Bimerge(root,sp_r,dir)
    end
end;
```

```
{ --- MAIN --- }
procedure main(n: int)
  h,s: handle;
  sval, height: int;
begin
  { Build original tree }
  h := RandTree(n);
  sval := random();
  s := NewNode(sval);

  { Print out tree }
  puts("Original Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s.value); puts("\n");

  { Sort tree }
  Bisort(h,s,UP);

  { Print out sorted tree }
  puts("Sorted Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s.value);
  puts("\n");

  { sort in desending order }
  Bisort(h,s,DOWN);

  { Print out sorted tree }
  puts("Sorted Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s.value);
  puts("\n")
end;
```

D.2 Parallel C Program

D.2.1 File: bitonicp_node.h

```
/* =============== NODE STRUCTURE ================== */

struct node {
  int value;
  struct node *left;
  struct node *right;
  };
```

```
typedef struct node HANDLE;

#define NIL ((HANDLE *) 0)
```

D.2.2 File: bitonicp_procs.h

```
/* ========== PROCEDURE TYPES/NUMS ================== */

HANDLE *NewNode();
void PrintIndent();
void PrintSubTree();
void PrintTree();
void InOrder();
HANDLE *RandTree();
void SwapValue();
void SwapLeft();
void SwapRight();
void Bimerge();
void SS_Bimerge();
void Bisort();
void SS_Bisort();
void sil_main();
void sil_child();

#define DD_Bimerge 0
#define DD_Bisort 1
#define DD_EXIT 2

/* ================= PROC NAMES ===============*/

#ifdef EXTERN
  extern char *procnames[];
#else
  static char *procnames[] =
  {
    "Bimerge",
    "Bisort",
    "EXIT"
  };
#endif
```

D.2.3 File: bitonicp.c

```
/* =================== PROGRAM bitonic===================== */

#include <us.h>       /* Uniform System */
#include <stdio.h>    /* Standard I/O */

#include NODEH   /* Node Definition */
#include PROCH   /* Procedure Types/Nums */

#include "include/util.h"  /* Utility macros */

HANDLE *
/********/
NewNode(v)
/********/
int v;

{ HANDLE *h;
```

```
  h = makenode();
  h->value = v;
  h->left = NIL;
  h->right = NIL;
  return(h);
}

void
/*****************/
PrintIndent(n,path)
/*****************/
int n;
int path;

{ int divisor;

  if ((n == 0))
    puts("*");
  divisor = (1 << (n - 1));
  while ((n > 0))
    { if ((n == 1))
        if ((path == 0))
          puts("/----->");
        else
          puts("\\----->");
      else
        if (((path / divisor) == ((path % divisor) / (divisor / 2))))
          puts("       ");
        else
          puts("|      ");
      path = (path % divisor);
      divisor = (divisor / 2);
      n = (n - 1);
    }
}

void
/********************/
PrintSubTree(h,n,path)
/********************/
HANDLE *h;
int n;
int path;

{ HANDLE *l;
  HANDLE *r;

  if ((h != NIL))
    { r = h->right;
      PrintSubTree(r,(n + 1),(path * 2));
      puts("(");
      put(n);
      puts(")\t");
      PrintIndent(n,path);
      puts(" ");
      put(h->value);
      puts("\n");
      l = h->left;
      PrintSubTree(l,(n + 1),((path * 2) + 1));
    }
}

void
/**********/
```

```
PrintTree(h)
/**********/
HANDLE *h;

{
  PrintSubTree(h,0,0);
}

void
/********/
InOrder(h)
/********/
HANDLE *h;

{ HANDLE *l;
  HANDLE *r;

  if ((h != NIL))
    { l = h->left;
      r = h->right;
      InOrder(l);
      put(h->value);
      puts(" ");
      InOrder(r);
    }
}

HANDLE *
/*********/
RandTree(n)
/*********/
int n;

{ int next_val;
  HANDLE *h;
  HANDLE *l;
  HANDLE *r;

  if ((n > 1))
    { next_val = random();
      h = NewNode(next_val);
      l = RandTree((n / 2));
      r = RandTree((n / 2));
      h->left = l;
      h->right = r;
      /* EmptyStatement */
    }
  else
    h = NIL;
  return(h);
}

void
/************/
SwapValue(l,r)
/************/
HANDLE *l;
HANDLE *r;

{ int temp;

  temp = l->value;
  l->value = r->value;
  r->value = temp;
}
```

```
void
/************/
SwapLeft(l,r)
/************/
HANDLE *l;
HANDLE *r;

{ HANDLE *ll;
  HANDLE *rl;

  ll = l->left;
  rl = r->left;
  l->left = rl;
  r->left = ll;
}

void
/*************/
SwapRight(l,r)
/*************/
HANDLE *l;
HANDLE *r;

{ HANDLE *lr;
  HANDLE *rr;

  lr = l->right;
  rr = r->right;
  l->right = rr;
  r->right = lr;
}

void
/*********************/
Bimerge(root,sp_r,dir)
/*********************/
HANDLE *root;
HANDLE *sp_r;
int dir;

{ int rightexchange;
  int elementexchange;
  int temp;
  HANDLE *sp_l;
  HANDLE *pl;
  HANDLE *pr;
  HANDLE *rl;
  HANDLE *rr;

  rightexchange = ((root->value > sp_r->value) ^ dir);
  if (rightexchange)
    SwapValue(root,sp_r);
  pl = root->left;
  pr = root->right;
  while ((pl != NIL))
    { elementexchange = ((pl->value > pr->value) ^ dir);
      if (rightexchange)
        if (elementexchange)
          { SwapValue(pl,pr);
            SwapRight(pl,pr);
            pl = pl->left;
            pr = pr->left;
          }
        else
```

```
          { pl = pl->right;
            pr = pr->right;
          }
      else
        if (elementexchange)
          { SwapValue(pl,pr);
            SwapLeft(pl,pr);
            pl = pl->right;
            pr = pr->right;
          }
        else
          { pl = pl->left;
            pr = pr->left;
          }
    }
  if ((root->left != NIL))
    { rl = root->left;
      rr = root->right;
      sp_l = NewNode(root->value);
      { int input_endpoint;
        input_endpoint = my_endpoint;
        if (my_endpoint == my_index)  /* only 1 proc */
          { SS_Bimerge(rl,sp_l,dir);
            SS_Bimerge(rr,sp_r,dir);
          }
        else /* more than one proc, fork off a procedure call */
          /* ======= set up for fork of Bimerge(rl,sp_l,dir); ========== */
          { int p, p_endpoint, *p_arglist, interval; short *p_pending;

            p_endpoint = my_endpoint;
            interval = (my_endpoint - my_index + 1) >> 1;
            my_endpoint -= interval;
            p = my_endpoint + 1;
            GET_PENDING(p_pending,1); /* get pending lock for p */
            p_arglist = arg_list_ptrs[p];
            *((short **) p_arglist)++ = p_pending;
            *((int*) p_arglist)++ = DD_Bimerge;
            *((int*) p_arglist)++ = p_endpoint;
            *((HANDLE **) p_arglist)++ = rl;
            *((HANDLE **) p_arglist)++ = sp_l;
            *((int *) p_arglist)++ = dir;
            START(p);  /* let processor p start */
            Bimerge(rr,sp_r,dir);
            WAIT_ZERO(*p_pending);
            RELEASE_PENDING; /* release pending lock */
          }
        my_endpoint = input_endpoint;
      }
      root->value = sp_l->value;
    }
}

void
/************************/
SS_Bimerge(root,sp_r,dir)
/************************/
HANDLE *root;
HANDLE *sp_r;
int dir;

{ int rightexchange;
  int elementexchange;
  int temp;
  HANDLE *sp_l;
  HANDLE *pl;
  HANDLE *pr;
  HANDLE *rl;
  HANDLE *rr;

  rightexchange = ((root->value > sp_r->value) ^ dir);
  if (rightexchange)
    SwapValue(root,sp_r);
  pl = root->left;
  pr = root->right;
  while ((pl != NIL))
    { elementexchange = ((pl->value > pr->value) ^ dir);
      if (rightexchange)
        if (elementexchange)
          { SwapValue(pl,pr);
            SwapRight(pl,pr);
            pl = pl->left;
            pr = pr->left;
          }
        else
          { pl = pl->right;
            pr = pr->right;
          }
      else
        if (elementexchange)
          { SwapValue(pl,pr);
            SwapLeft(pl,pr);
            pl = pl->right;
            pr = pr->right;
          }
        else
          { pl = pl->left;
            pr = pr->left;
          }
    }
  if ((root->left != NIL))
    { rl = root->left;
      rr = root->right;
      sp_l = NewNode(root->value);
      { SS_Bimerge(rl,sp_l,dir);
        SS_Bimerge(rr,sp_r,dir);
      }
      root->value = sp_l->value;
    }
}

void
/*******************/
Bisort(root,sp_r,dir)
/*******************/
HANDLE *root;
HANDLE *sp_r;
int dir;

{ HANDLE *l;
  HANDLE *r;
  HANDLE *sp_l;

  if ((root->left == NIL))
    { if (((root->value > sp_r->value) ^ dir))
        SwapValue(root,sp_r);
    }
  else
    { l = root->left;
      r = root->right;
      sp_l = NewNode(root->value);
      { int input_endpoint;
        input_endpoint = my_endpoint;
        if (my_endpoint == my_index)  /* only 1 proc */
          { SS_Bisort(l,sp_l,dir);
            SS_Bisort(r,sp_r,(! dir));
          }
        else /* more than one proc, fork off a procedure call */
          /* ======= set up for fork of Bisort(l,sp_l,dir); ========== */
          { int p, p_endpoint, *p_arglist, interval; short *p_pending;

            p_endpoint = my_endpoint;
            interval = (my_endpoint - my_index + 1) >> 1;
            my_endpoint -= interval;
            p = my_endpoint + 1;
            GET_PENDING(p_pending,1); /* get pending lock for p */
            p_arglist = arg_list_ptrs[p];
            *((short **) p_arglist)++ = p_pending;
            *((int*) p_arglist)++ = DD_Bisort;
            *((int*) p_arglist)++ = p_endpoint;
            *((HANDLE **) p_arglist)++ = l;
            *((HANDLE **) p_arglist)++ = sp_l;
            *((int *) p_arglist)++ = dir;
            START(p);  /* let processor p start */
            Bisort(r,sp_r,(! dir));
            WAIT_ZERO(*p_pending);
            RELEASE_PENDING; /* release pending lock */
          }
        my_endpoint = input_endpoint;
      }
      root->value = sp_l->value;
      Bimerge(root,sp_r,dir);
    }
}

void
/**********************/
SS_Bisort(root,sp_r,dir)
/**********************/
HANDLE *root;
HANDLE *sp_r;
int dir;

{ HANDLE *l;
  HANDLE *r;
  HANDLE *sp_l;

  if ((root->left == NIL))
    { if (((root->value > sp_r->value) ^ dir))
        SwapValue(root,sp_r);
    }
  else
    { l = root->left;
      r = root->right;
      sp_l = NewNode(root->value);
      { SS_Bisort(l,sp_l,dir);
        SS_Bisort(r,sp_r,(! dir));
      }
      root->value = sp_l->value;
      SS_Bimerge(root,sp_r,dir);
    }
}

void
/*********/
sil_main(n)
```

```
/*********/
int n;

{ HANDLE *h;
  HANDLE *s;
  int sval;
  int height;

  h = RandTree(n);
  sval = random();
  s = NewNode(sval);
  puts("Original Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s->value);
  puts("\n");
  Bisort(h,s,0);
  puts("Sorted Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s->value);
  puts("\n");
  Bisort(h,s,1);
  puts("Sorted Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s->value);
  puts("\n");
}

void
/********/
SS_main(n)
/********/
int n;

{ HANDLE *h;
  HANDLE *s;
  int sval;
  int height;

  h = RandTree(n);
  sval = random();
  s = NewNode(sval);
  puts("Original Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s->value);
  puts("\n");
  SS_Bisort(h,s,0);
  puts("Sorted Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s->value);
  puts("\n");
  SS_Bisort(h,s,1);
  puts("Sorted Tree: ");
  PrintTree(h);
  InOrder(h);
  put(s->value);
  puts("\n");
}

void
/*********/
```

```
sil_child()
/*********/
{ short *pending_addr;
  int my_exit, procedure_num;
  my_exit = 0;
  Atomic_add(start_counter,-1); /* signal this processor started */
  while (my_exit == 0)  /* while this processor not killed */
    { WAIT_NONZERO(*my_busy_lock); /* wait for work to do */
      pending_addr = (short *) my_arg_list[0];
      procedure_num = (int) my_arg_list[1];
      my_endpoint = (int) my_arg_list[2];
      switch (procedure_num)
        {
          case DD_Bimerge:
            Bimerge(
              (HANDLE *) my_arg_list[3],
              (HANDLE *) my_arg_list[4],
              (int) my_arg_list[5]); break;
          case DD_Bisort:
            Bisort(
              (HANDLE *) my_arg_list[3],
              (HANDLE *) my_arg_list[4],
              (int) my_arg_list[5]); break;
          case DD_EXIT:
            my_exit = 1;  break;
        }
      *my_busy_lock = 0;
      Atomic_add(pending_addr,-1);
    }
}
```

Bibliography

[AH87] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, 1987.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4), October 1987.

[Ban79a] U. Banerjee. *Speedup of Ordinary Programs.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, October 1979. Dept. of Computer Science Rpt. 79-989.

[Ban79b] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, 1979.

[Bar78] J. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21:724–736, 1978.

[Bar84] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* Studies in Logic. North-Holland, Amsterdam, revised edition, 1984.

[BBN88a] BBN Advanced Computers Inc. *Inside the GP1000*, 1.0 edition, October 1988.

[BBN88b] BBN Advanced Computers Inc. *Programming in C with the Uniform System*, 1.0 edition, October 1988.

[BBN89] BBN Advanced Computers Inc. *Private Communication*, September 1989. BBN Hotline.

[BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *ACM SIGPLAN Notices, Vol 21,7*, 1986.

[BKK$^+$89] Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, Kathryn McKinley, and Jaspal Subholk. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings Supercomputing '89*, pages 540–550, 1989.

[BN89] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing*, 18(2):216–228, 1989.

[Bro88] Eugene D. Brooks III. PCP: A parallel extension to C that is 99 percent fat free. Technical Report UCRL-99673 PREPRINT, Universtiy of California, Lawrence Livermore National Laboratory, 1988. Revised September 25, 1989.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–169, October 1988.

[DK85] Henry Dietz and David Klappholz. Refined C: A sequential language for parallel processing. In *ICPP*, pages 442–449, 1985.

[GGGJ88] Vincent A. Guarna Jr., D. Gannon, Y. Gaur, and D. Jablonowski. Faust: An environment for programming parallel scientific applications. In *Supercomputing '88*, 1988.

[Gor79] M.J.C. Gordon. *The Denotational Description of Programming languages.* Springer-Verlag, 1979.

[GS] Carl A. Gunter and Dana S. Scott. *Handbook of Theoretical Computer Science*, chapter on Semantic Domains. North Holland. To appear.

[Gua88] Vincent 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.

[Har86] W. Ludwell Harrison III. Compiling Lisp for evaluation on a tightly coupled multiprocessor. Technical Report CSRD Rpt. No. 565, Center for Supercomputing Research and Development, University of Illinois at Urbanan-Champaign, March 1986.

[Har89] W. Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. Technical Report CSRD Rpt. No. 860, Center for Supercomputing Research and Development, University of Illinois at Urbanan-Champaign, February 1989.

[HP88] W. Ludwell Harrison III and David A. Padua. Parcel: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp. In *Proceedings of the 1988 International Conference on Supercomputing*, July 1988.

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas 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.

[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. 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.

[KKK89] D. Klappholz, A. Kallis, and X. Kong. Refined C - An Update. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, 1989.

[KMR87] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic domain decomposition in Blaze. In *Proceedings of the International Conference on Parallel Processing*, 1987.

[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.

[LH88a] James R. Larus and Paul 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.

[LH88b] James R. Larus and Paul N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proceedings of the ACM/SIGPLAN PPEALS 1988 - Parallel Programming: Experience with Applications, Languages and Systems*, pages 100–110, July 1988.

[Luc87] J. M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming.* Ph.D. dissertation, MIT, 1987.

[Man74] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. dissertation, University of Edinburgh, Scotland, 1981.

[Nei88] A. Neirynck. *Static Analysis of Aliasing and Side Effects in Higher-Order Languages.* Ph.D. dissertation, Cornell University, January 1988.

[Nic84] A. Nicolau. *Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers.* Ph.D. dissertation, Yale University, 1984.

[NPD87] A. Neirynck, P. Panangaden, and A.J. Demers. Computation of aliases and support sets. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 274–283, 1987.

[PGH+89] Constantine D. Polychronopoulos, Miland Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Paraphrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 39–48, 1989.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[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.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the '89 ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[SA88] Kevin Smith and William F. Appelbe. PAT - An Interactive Fortran Parallelizing Assitant Tool. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 285–293, 1988.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Ten81] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, October 1982.

[ZBG88] H. Zima, H-J. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.