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 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

1 Introduction
   1.1 Related Work ........................................... 2
   1.2 Our Approach ........................................... 5
   1.3 Organization of the Thesis ............................. 6

2 Setting
   2.1 SIL ....................................................... 7
   2.2 Trees and DAGs .......................................... 9

3 Interference Analysis Tools ........................................ 11
   3.1 Data Structure Abstraction .............................. 12
      3.1.1 Path Matrices and Path Expressions ................. 12
      3.1.2 Operations on Path Expressions ..................... 14
   3.2 Analysing Simple Statements (TREES) .................. 18
      3.2.1 Handle Assignment Statements ....................... 18
      3.2.2 Handle Update Statements ........................... 23
   3.3 Estimating nil handles. ................................... 25
      3.3.1 Data abstraction for nil handle estimates. ........ 27
      3.3.2 Analysis rules for nil handle estimates. .......... 27
   3.4 Analysing Simple Statements (DAGS) .................... 30
      3.4.1 Data abstraction for reference counts ............... 33
      3.4.2 Analysis Rules for DAG estimates ................... 33
   3.5 Analysing Compound Statements .......................... 36
      3.5.1 Merging Path Matrices ............................... 38
   3.6 Analysing Procedure Calls ................................ 40

4 Formal Description of Interference Analysis ..................... 46
   4.1 A Formal Description of SIL ............................. 46
   4.2 Path Matrices Revisited .................................. 48
      4.2.1 Defining a merge for Path Matrices .................. 52
   4.3 Abstract Semantics ...................................... 54
   4.4 Soundness Theorem ...................................... 62
5 Interference and Parallelization
   5.1 Interference between Basic Statements 78
   5.2 Interference between Procedure Calls 81
   5.3 Interference between Statement Sequences 84

6 Generating Parallel Programs
   6.1 Generating Parallel Code for the Butterfly 88
       6.1.1 Strategy 90
   6.2 An example: Adaptive Bitonic Sort 93
       6.2.1 Path Matrix Computation 93
       6.2.2 Generating the Parallel Program 93
       6.2.3 Speedup results 96
   6.3 Further Issues 99

7 Conclusions and Further Work 105

A Standard Semantics for SIL
   A.1 Semantic Domains 109
   A.2 Expressions 109
   A.3 Scalar Statements 110
   A.4 Handle Statements 110
   A.5 Compound Statements 111
   A.6 Procedures 112
   A.7 Programs 113

B Translating SIL to C
   B.1 Node Type Definition 114
   B.2 Statements 115
   B.3 Procedures and Functions 115

C Illustrative Interference Analysis Computations
   C.1 SIL program 117
   C.2 Analysis of the main program 121
   C.3 Analysis of build_list 123
   C.4 Analysis of qsort 127

D Generating Parallel C Programs: An Example
   D.1 Sequential SIL Program 132
   D.2 Parallel C Program 137
       D.2.1 File: bitonicp_node.h 137
       D.2.2 File: bitonicp_procs.h 138
       D.2.3 File: bitonicp.c 138

Bibliography 147
# List of Figures

2.1 Abstract Syntax of SIL ........................................... 8
2.2 Examples of recursive handle types ................................ 8
2.3 A TREE and a DAG .............................................. 10

3.1 Structure of the analysis ........................................ 11
3.2 An example path matrix ......................................... 14
3.3 Path expression equality, merging, and concatenation .......... 17
3.4 Rules for $A := \text{nil}$, $A := \text{new()}$, and $A := B$ .... 19
3.5 Example application of the simple handle assignment rules ... 20
3.6 Rule for $A := B.left$ .......................................... 21
3.7 Functions used in the analysis of $A := B.left$ .............. 22
3.8 An example of handle assignments ................................ 24
3.9 Analysis for $A.left := \text{nil}$ ................................... 25
3.10 Analysis for $A.left := B$ ...................................... 26
3.11 Data abstraction for nil estimates ............................. 27
3.12 Nil estimate for $A := \text{nil}$, $A := \text{new()}$, and $A := B$ . 28
3.13 Example application of nil estimate ............................ 29
3.14 Nil estimate for $A := B.left$ .................................. 30
3.15 Nil estimate for $A.left := \text{nil}$ and $A.left := B$ ....... 31
3.16 An example program that creates DAG nodes .................. 32
3.17 Operations on abstract reference counts ....................... 33
3.18 Extended rule for $A := B.left$ ................................. 34
3.19 Extended rule for $A.left := \text{nil}$ ............................ 35
3.20 Extended rule for $A.left := B$ ................................ 37
3.21 Iterative approximation for a simple while loop ............ 38
3.22 An example of the merge operation ............................. 39
3.23 Example of a procedure call ................................... 41
3.24 Example of a function call .................................... 42
3.25 Estimating the path matrices for reverse ...................... 44
3.26 Estimating path matrices for tail-recursive FindMin .......... 45

4.1 Semantic domains for SIL ....................................... 47
4.2 The model for the environment and memory .................... 47
4.3 A tree and the corresponding path matrix ..................... 48
4.4 Partial ordering of path expressions of length 1 ............. 49
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Partial ordering of path expressions of length 1 or 2</td>
<td>50</td>
</tr>
<tr>
<td>4.6</td>
<td>Partial ordering for nil estimates and reference counts.</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Merging path matrices.</td>
<td>53</td>
</tr>
<tr>
<td>4.8</td>
<td>Abstract semantic domains.</td>
<td>54</td>
</tr>
<tr>
<td>4.9</td>
<td>Abstract semantic functions for handle assignment statements.</td>
<td>55</td>
</tr>
<tr>
<td>4.10</td>
<td>Abstract semantic functions for compound statements.</td>
<td>56</td>
</tr>
<tr>
<td>4.11</td>
<td>Semantic functions for while loops.</td>
<td>57</td>
</tr>
<tr>
<td>4.12</td>
<td>Abstract functions for procedure calls.</td>
<td>57</td>
</tr>
<tr>
<td>4.13</td>
<td>Semantic functions for non-recursive procedures.</td>
<td>58</td>
</tr>
<tr>
<td>4.14</td>
<td>Abstract semantic function for recursive procedures.</td>
<td>60</td>
</tr>
<tr>
<td>4.15</td>
<td>Unrolling the semantic functions for while.</td>
<td>66</td>
</tr>
<tr>
<td>4.16</td>
<td>Recursive procedures in the Standard Semantics.</td>
<td>69</td>
</tr>
<tr>
<td>4.17</td>
<td>The k'th unwinding of $R$ and $Q$.</td>
<td>70</td>
</tr>
<tr>
<td>4.18</td>
<td>The k'th unwinding of $F$.</td>
<td>71</td>
</tr>
<tr>
<td>4.19</td>
<td>The first unwinding of $F$, $F^{(1)}$.</td>
<td>72</td>
</tr>
<tr>
<td>4.20</td>
<td>$F_{n,m}^{p,m}$ approximants.</td>
<td>75</td>
</tr>
<tr>
<td>5.1</td>
<td>Transforming sequential statements to a parallel statement.</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>Functions for read and write sets of statement $s$ relative to path</td>
<td></td>
</tr>
<tr>
<td></td>
<td>matrix $p$.</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>Examples of interfering statements.</td>
<td>80</td>
</tr>
<tr>
<td>5.4</td>
<td>Example program and path matrices.</td>
<td>83</td>
</tr>
<tr>
<td>5.5</td>
<td>Parallel version of example program.</td>
<td>85</td>
</tr>
<tr>
<td>5.6</td>
<td>Initial path matrix $p$ with two parallel statement sequences $U$</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>and $V$.</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>Relative read and write sets given path matrix $p$ and live handles</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>$L$.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Current System</td>
<td>89</td>
</tr>
<tr>
<td>6.2</td>
<td>Global/Local memory access times (unloaded switch).</td>
<td>89</td>
</tr>
<tr>
<td>6.3</td>
<td>Examples of unfoldings.</td>
<td>91</td>
</tr>
<tr>
<td>6.4</td>
<td>Interval and Pool allocation.</td>
<td>92</td>
</tr>
<tr>
<td>6.5</td>
<td>Bitonic Sort and Swap procedures.</td>
<td>94</td>
</tr>
<tr>
<td>6.6</td>
<td>Bitonic Merge procedure.</td>
<td>95</td>
</tr>
<tr>
<td>6.7</td>
<td>Path Matrix before recursive calls in Bmerge.</td>
<td>96</td>
</tr>
<tr>
<td>6.8</td>
<td>Global/Local memory layout.</td>
<td>97</td>
</tr>
<tr>
<td>6.9</td>
<td>Parallel C version of Bisort.</td>
<td>98</td>
</tr>
<tr>
<td>6.10</td>
<td>Speedup for 1 ... 32 processors.</td>
<td>101</td>
</tr>
<tr>
<td>6.11</td>
<td>Speedup for 1, 2, 4, 8, 16, and 32 processors.</td>
<td>102</td>
</tr>
<tr>
<td>6.12</td>
<td>The effect of locality.</td>
<td>103</td>
</tr>
<tr>
<td>6.13</td>
<td>The effect of processor allocation overhead.</td>
<td>104</td>
</tr>
<tr>
<td>7.1</td>
<td>Future Plans</td>
<td>107</td>
</tr>
<tr>
<td>C.1</td>
<td>Diagrammatic outline for $qsort$.</td>
<td>128</td>
</tr>
<tr>
<td>C.2</td>
<td>Iterative Approximation for $qsort$.</td>
<td>129</td>
</tr>
</tbody>
</table>
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 interest-
1.1 Related Work

ing 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 Neiryck [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 analyzing 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
### Figure 2.1: Abstract Syntax of SIL.

```plaintext
<Program> ::= program <ProgId>
[ constant <ConstantList> 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 2.2: Examples of recursive handle types.

```plaintext
nodedef
value: int;
link: handle
end;

nodedef
key: int;
left, right: handle
end;

nodedef
lowofsecond, lowofthird: int;
first, second, third: handle
end;

type handle =
nil |
(value: int, link: handle)

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

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

\[
\begin{align*}
a & := \text{nil}; \\
a & := \text{new}(); \\
a & := b; \\
a & := b.h; \\
a.h & := b; \\
x & := a.s; \\
\text{and} \ a.s & := x.
\end{align*}
\]

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.

![Diagram](image)

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. $S + \text{link expression}$, 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.

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( S )</td>
<td>( L^1L^+L^1 )</td>
<td>( R^1D^+ )</td>
<td>( R^1 )</td>
<td>( R^1L^1 )</td>
</tr>
<tr>
<td>( b )</td>
<td>( S )</td>
<td>( S )</td>
<td>( S^? )</td>
<td>( D^+ )</td>
<td>( S )</td>
</tr>
<tr>
<td>( c )</td>
<td>( D^+ )</td>
<td>( S )</td>
<td>( L^1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( e )</td>
<td>( S + D^+ )</td>
<td>( S )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 \((L,R,\text{or} D)\) 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 \((L, R, \text{or} D \text{ for binary trees})\) to a subsequence of the form \( \chi^i \), \( \chi^+ \), or \( \chi^i\chi^+ \). First-order normalization is based on the observation that \( \chi^i\chi^j \) denotes the same set of paths at \( \chi^{(i+j)} \), and that \( \chi^+\chi^+ \) denotes the same set as \( \chi^i\chi^+ \). Given a subsequence \( S \) containing links of the same kind, the reduced subsequence \( S_{\text{red}} \) is computed as follows. \( S \) is divided into \( S_{\text{link}} = \chi_{i}^{l_i} \ldots \chi_{m}^{l_m} \) and \( S_{\text{plus}} = \chi_{i}^{+} \ldots \chi_{n}^{+} \), where \( S_{\text{link}} \) collects all links of the form \( \chi^i \) and \( S_{\text{plus}} \) collects all links of the form \( \chi^+ \). Given that \( n_{\text{plus}} \) is the number of links in \( \chi_{\text{plus}} \) and \( n_{\text{link}} \) is the sum of the \( l_i \)'s in \( S_{\text{link}} \), the reduced subsequence \( S_{\text{red}} \) is given by:
### 3.1 Data Structure Abstraction

\[ S_{red} = \begin{cases} 
\lambda^{n_{\text{link}}} & (n_{\text{link}} > 0, n_{\text{plus}} = 0) \\
\lambda^{(n_{\text{plus}} - 1)} \lambda^+ & (n_{\text{link}} = 0, n_{\text{plus}} > 0) \\
\lambda^{(n_{\text{plus}} + n_{\text{link}} - 1)} \lambda^+ & (n_{\text{link}} > 0, n_{\text{plus}} > 0).
\end{cases} \]

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 \( \lambda^+ D^+ \) (\( \lambda = L \) or \( R \)) denote the same set of paths as \( \lambda^1 D^+ \), and similarly \( D^+ \lambda^+ \) denotes the same set as \( D^+ \lambda^1 \). The reduction rules for adjacent subsequences are as follows:

\[
\begin{align*}
\lambda^+ D^+ & \Rightarrow \lambda^1 D^+ \\
\lambda^i \lambda^+ D^+ & \Rightarrow \lambda^{(i+1)} D^+ \\
\lambda^+ D^i D^+ & \Rightarrow \lambda^{1+} D^i D^+ \\
\lambda^i \lambda^+ D^i D^+ & \Rightarrow \lambda^{(i+1)} D^i D^+ \\
D^+ \lambda^+ & \Rightarrow D^+ \lambda^1 \\
D^+ \lambda^i \lambda^+ & \Rightarrow D^+ \lambda^{(i+1)} 
\end{align*}
\]

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

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: $\text{squash}(e_1, e_2) = D^+$. 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 $e_2$, $x = \text{squash}(e_1, e_2)$ 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 subsequent with the best match is squashed using simple rules defined on small subsequences (for example, $\text{squash}(L^1L^+, L^1L^+) \Rightarrow L^{\min(i,j)}L^+$ and $\text{squash}(D^iD^+, L^1L^+) \Rightarrow \text{squash}(D^{\min(i,j)}D^+)$). 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: $\text{squash}(R^+, R^1) = R^+$, $\text{squash}(L^2R^1, L^1R^2) = L^1D^1R^1$, and $\text{squash}(L^1L^+, L^1L^+L^1) = L^1L^+$.

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 $(p, \text{true})$, and a possible path expression $p'$ with the pair $(p', \text{false})$, then we can define the following extensions to the path expression operations.

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

Path expressions $(p_1, b_1)$ and $(p_2, b_2)$ are equal $\Leftrightarrow p_1 = p_2$ and $b_1 = b_2$.

The *and-merge* of path expressions $(p_1, b_1)$ and $(p_2, b_2)$ is defined as:

$$(p_1, b_1) \land^\land (p_2, b_2) = (p_1 \land p_2, b_1 \land b_2).$$

The *or-merge* of path expressions $(p_1, b_1)$ and $(p_2, b_2)$ is defined as:

$$(p_1, b_1) \land^\lor (p_2, b_2) = (p_1 \lor p_2, b_1 \lor b_2).$$

The *concatenation* of path expressions $(p_1, b_1)$ and $(p_2, b_2)$ is defined as:

$$(p_1, b_1) \cdot (p_2, b_2) = (p_1 \cdot p_2, b_1 \land b_2).$$

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 ($p_1 = p_2$)

<table>
<thead>
<tr>
<th>$p_1 \setminus p_1$</th>
<th>$S$</th>
<th>$e_1$</th>
<th>$S + e_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>$e_2$</td>
<td>false</td>
<td>$\text{normal}(e_1) = \text{normal}(e_2)$</td>
<td>false</td>
</tr>
<tr>
<td>$S + e_2$</td>
<td>false</td>
<td>false</td>
<td>$\text{normal}(e_1) = \text{normal}(e_2)$</td>
</tr>
</tbody>
</table>

### Path Expression Merging ($p_1 \bowtie p_2$)

<table>
<thead>
<tr>
<th>$p_2 \setminus p_1$</th>
<th>$S$</th>
<th>$e_1$</th>
<th>$S + e_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S$</td>
<td>$S + e_1$</td>
<td>$S + e_1$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$S + e_2$</td>
<td>$\text{squash}(e_1, e_2)$</td>
<td>$S + \text{squash}(e_1, e_2)$</td>
</tr>
<tr>
<td>$S + e_2$</td>
<td>$S + e_2$</td>
<td>$S + \text{squash}(e_1, e_2)$</td>
<td>$S + \text{squash}(e_1, e_2)$</td>
</tr>
</tbody>
</table>

### Path Expression Concatenation ($p_1 \cdot p_2$)

<table>
<thead>
<tr>
<th>$p_2 \setminus p_1$</th>
<th>$S$</th>
<th>$e_1$</th>
<th>$S + e_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S$</td>
<td>$e_1$</td>
<td>$S + e_1$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$e_2$</td>
<td>$e_1e_2$</td>
<td>$e_2 \bowtie (e_1e_2)$</td>
</tr>
<tr>
<td>$S + e_2$</td>
<td>$S + e_2$</td>
<td>$e_1 \bowtie (e_1e_2)$</td>
<td>$S + (e_1 \bowtie e_2 \bowtie (e_1e_2))$</td>
</tr>
</tbody>
</table>

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: $A := \text{nil}$, $A := \text{new}()$, $A := B$, and $A := B.f$. The first three forms are quite straightforward and their rules are given in figure 3.4. In the case of $A := \text{nil}$ and $A := \text{new}()$, 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: $c := \text{nil}$, $d := \text{new}()$, 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 $A := B.f$. 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 $p[B, h_j]$
3.2 Analysing Simple Statements (TREES)

\[ A := \text{nil} \]
\[
\forall h_i, h_j \in H'.
\]
\[
p'[h_i, h_j] \mapsto
\]
\[
\begin{cases}
\text{if } h_i = A \text{ and } h_j = A & \text{then} \\
\{ S \}
\end{cases}
\]
\[
\text{else if } h_i = A \text{ or } h_j = A & \text{then} \\
\{ \}
\]
\[
\text{else} \\
p[h_i, h_j]
\]

\[ A := \text{new()} \]
\[
\forall h_i, h_j \in H'.
\]
\[
p'[h_i, h_j] \mapsto
\]
\[
\begin{cases}
\text{if } h_i = A \text{ and } h_j = A & \text{then} \\
\{ S \}
\end{cases}
\]
\[
\text{else if } h_i = A \text{ or } h_j = A & \text{then} \\
\{ \}
\]
\[
\text{else} \\
p[h_i, h_j]
\]

\[ A := B \]
\[
\forall h_i, h_j \in H'.
\]
\[
p'[h_i, h_j] \mapsto
\]
\[
\begin{cases}
\text{if } h_i = A & \text{then} \\
\quad \begin{cases}
\text{if } h_j = B \text{ or } h_j = A & \text{then} \\
\{ S \}
\end{cases}
\quad \text{else } p[B,h_j]
\end{cases}
\]
\[
\begin{cases}
\text{else if } h_j = A & \text{then} \\
\quad \begin{cases}
\text{if } h_i = B & \text{then} \\
\{ S \}
\end{cases}
\quad \text{else } p[h_i,B]
\end{cases}
\]
\[
\text{else} \\
p[h_i, h_j]
\]

Figure 3.4: Rules for \( A := \text{nil}, A := \text{new()}, \) and \( A := B. \)
Figure 3.5: Example application of the simple handle assignment rules.
\[ A := B.left \]
\[ \forall h_i, h_j \in H' . \]
\[ p'[h_i, h_j] \mapsto \]
\[ \text{if } h_i = A \text{ then} \]
\[ \text{if } h_j = A \text{ then} \]
\[ \{ S \} \]
\[ \text{else if } h_j = B \text{ then} \]
\[ \{ \} \]
\[ \text{else} \]
\[ \text{map below\_rule } p[B,h_j] \]
\[ \text{else if } h_j = A \text{ then} \]
\[ \text{if } h_i = B \text{ then} \]
\[ \{ L^i \} \]
\[ \text{else} \]
\[ \text{map above\_rule } p[h_i,B] \]
\[ \text{else} \]
\[ p[h_i, h_j] \]

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 \text{above\_rule}(e, d)\) 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 \( x L^1 \) between \( h_i \) and \( A \).

The function \text{below\_rule}(e, d)\) 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)
(  ⟨S, d⟩    ⇒  {⟨L^1, d⟩}
   ⟨x, d⟩    ⇒  {⟨x^L^1, d⟩}
   ⟨S + x, d⟩⇒  {⟨above_rule(S, d) ⊗ (above_rule(x, d))⟩}
)
below_rule(e, d)
(  ⟨S, d⟩    ⇒  {}       
   ⟨x, d⟩    ⇒  shorten(x, d)
   ⟨S + x, d⟩⇒  shorten(x, false)
)
shorten(e, d)
(  ⟨L^1, d⟩    ⇒  {⟨S, d⟩}
   ⟨L^n, d⟩    ⇒  {⟨L^{n-1}, d⟩}
   ⟨L^+, d⟩    ⇒  {⟨S + L^+, d⟩}
   ⟨L^1x, d⟩   ⇒  {⟨x, d⟩}
   ⟨L^nx, d⟩   ⇒  {⟨L^{n-1}x, d⟩}
   ⟨L^+x, d⟩   ⇒  {⟨x ⊕ L^+x, d⟩}
   ⟨D^1, d⟩    ⇒  {⟨S, false⟩}
   ⟨D^n, d⟩    ⇒  {⟨D^{n-1}, false⟩}
   ⟨D^+, d⟩    ⇒  {⟨S + D^+, false⟩}
   ⟨D^1x, d⟩   ⇒  {⟨x, false⟩}
   ⟨D^n, d⟩    ⇒  {⟨D^{n-1}x, false⟩}
   ⟨D^+x, d⟩   ⇒  {⟨x ⊕ D^+x, false⟩}
   ¬     ⇒  []
)

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: \(A.f := \text{nil}\), and \(A.f := B\). Unlike the handle assignment statements, handle update statements can change the relationships between other than \(A\) and \(B\).

We first consider rules of the form \(A.f := \text{nil}\) by examining the analysis rule for \(A.left := \text{nil}\). The effect of the statement \(A.left := \text{nil}\) 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 \(A.left := \text{nil}\) is given in figure 3.9. In this rule we use \(\text{path}(A, \text{path} \_ \text{description}, B, p)\) as a predicate to test for certain paths in the path matrix entry \(p[A, B]\). For example, \(\text{path}(h_i, \_?, A, p)\) is true if there is any path expression (possible or definite) in \(p[h_i, A]\), and \(\text{path}(h_i, \_*, A, p)\) 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 \(\text{path}\), we can now examine how the analysis rule estimates the effect of \(A.left := \text{nil}\). 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 be 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 \(\emptyset\). 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 \(\text{remove} \_ \text{head}\), \(\text{remove} \_ \text{tail}\), and \(\text{make} \_ \text{possible}\). 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 \(\text{DAG}\), or a \(\text{TREE}\). In figure 3.10 we outline the case for \(\text{TREES}\). In section 3.4 we give extensions to the analysis to handle \(\text{DAGS}\). 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
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>S</td>
<td>L^2L^+</td>
<td>R^1D^+</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

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

```
    a
   / \
  b   c
```

```
    a
   / \
  b   d
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>S</td>
<td>L^2L^+</td>
<td>R^1D^+</td>
<td>R^1</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>D^+</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

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

```
    a
   / \
  b   c?
```

```
    a
   / \
  b   d
```

```
    a
   / \
  b   e
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>S</td>
<td>L^2L^+</td>
<td>R^1D^+</td>
<td>R^1</td>
<td>R^1L^1</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>S</td>
<td></td>
<td>S?</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>D^+</td>
<td>S</td>
<td></td>
<td>L^1</td>
</tr>
<tr>
<td>e</td>
<td></td>
<td>(S + D^+)?</td>
<td></td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.8: An example of handle assignments.
3.3 Estimating nil handles.

\[ A.left := \text{nil} \]

\[ \forall h_i, h_j \in H'. \]

\[ p'[h_i, h_j] \mapsto \]

\[ \text{if path}(h_i, *, A, p) \text{ and path}(A, L*, h_j, p) \text{ then} \]

\[ \{ \} \]

\[ \text{else if path}(h_i, ?, A, p) \text{ and path}(A, L?, h_j, p) \text{ then} \]

\[ \begin{array} {l}
\text{if path}(h_i, S, A, p) \text{ then} \\
\text{remove\_head}(L, p[h_i, h_j]) \\
\text{else if path}(A, L, h_j, p) \text{ then} \\
\text{remove\_tail}(L, p[h_i, h_j]) \\
\text{else} \\
\text{make\_possible}(p[h_i, h_j])
\end{array} \]

\[ \text{else} \]

\[ p[h_i, h_j] \]

Figure 3.9: Analysis for \( A.left := \text{nil} \).

detected. The analysis for \( A.left := B \) begins by producing a path matrix \( p' \) that is an estimate of the effect of \( A.left := \text{nil} \). 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.
\( A.\text{left} := B \)

\[
p' \rightarrow f(A.\text{left}:=\text{nil}, p);
\]

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', true \rangle,

p'[B, h_j]

)

)

else

\[
p'[h_i, h_j] \cup
\]

( connect_all

( p'[h_i, A],

\langle L', true \rangle,

make_possible(p'[B, h_j])

)

)

else

\[
p'[h_i, h_j]
\]

Figure 3.10: Analysis for \( A.\text{left} := B \).
3.3 Estimating nil handles.

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 \( \odot \) 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.

\[ \begin{align*}
\odot & \text{ (Either)} \\
\bullet & \text{ (Node)} \\
\odot & \text{ (Nil)} 
\end{align*} \]

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 \( p.IsNode[h_i] = (\bullet, \odot, \odot) \) 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 \( \bullet, \odot, \) and \( \odot \) as specified in figure 3.11, the nil.merge of nil estimates \((x.self, x.left \land x.right) \) and \((y.self, y.left \land y.right) \) is \((x.self \land y.self, (x.left \land y.left) \land (x.right \land y.right)) \).

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 \( A := \text{nil}, A := \text{new()} \), 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 \( A := \text{nil} \), \( A \) is definitely a nil pointer. Executing \( A := \text{new()} \) 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 \( a \) \((\bullet, \bullet) \) indicates that \( a \), \( a.left \), and \( a.right \) are
all handles to nodes, while the estimate for $b (\bullet, \circ \circ)$ 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 $c := \text{nil}$, $d := \text{new}()$, and $e := a$ illustrate the three analysis
rules given in 3.12.

\[
\begin{align*}
A & := \text{nil} \\
\forall h_i \in H' . & \\
& p'.IsNode[h_i] \mapsto \\
& \quad \text{if } h_i = A \text{ then } (\circ, \circ \circ) \text{ else } p.IsNode[h_i]; \\
B & := \text{new}() \\
\forall h_i \in H' . & \\
& p'.IsNode[h_i] \mapsto \\
& \quad \text{if } h_i = A \text{ then } (\bullet, \circ \circ) \text{ else } p.IsNode[h_i]; \\
A & := B \\
\forall h_i \in H' . & \\
& p'.IsNode[h_i] \mapsto \\
& \quad \text{if } h_i = A \text{ then } p.IsNode[B] \text{ else } p.IsNode[h_i];
\end{align*}
\]

Figure 3.12: Nil estimate for $A := \text{nil}$, $A := \text{new}()$, 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 ($\circ$). 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 $A.f := \text{nil}$ 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 $X.left := \text{nil}$ 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 $A.left := \text{nil}$. 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 $A$ ($\text{path}(h_i,S,A,p) = \text{true}$), $h_i.left$ 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 $A.left := \text{nil}$, except that the nilness of the left field of
$A$ and all handles that are the same as $A$ is defined by the value $p.IsNode[B].self$. 
3.3 Estimating nil handles.

(a) Initial Path Matrix

<table>
<thead>
<tr>
<th></th>
<th>(a(\bullet, \bullet))</th>
<th>(b(\bullet, \circ \circ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(S)</td>
<td>(L^+)</td>
</tr>
<tr>
<td>(b)</td>
<td>(S)</td>
<td></td>
</tr>
</tbody>
</table>

(b) After statement: \(c := \text{nil}\)

<table>
<thead>
<tr>
<th></th>
<th>(a(\bullet, \bullet))</th>
<th>(b(\bullet, \circ \circ))</th>
<th>(c(\circ, \circ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(S)</td>
<td>(L^+)</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>(S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td></td>
<td></td>
<td>(S)</td>
</tr>
</tbody>
</table>

(c) After statement: \(d := \text{new}()\)

<table>
<thead>
<tr>
<th></th>
<th>(a(\bullet, \bullet))</th>
<th>(b(\bullet, \circ \circ))</th>
<th>(c(\circ, \circ))</th>
<th>(d(\circ, \circ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(S)</td>
<td>(L^+)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>(S)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td></td>
<td></td>
<td>(S)</td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td></td>
<td></td>
<td></td>
<td>(S)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>(a(\bullet, \bullet))</th>
<th>(b(\bullet, \circ \circ))</th>
<th>(c(\circ, \circ))</th>
<th>(d(\circ, \circ))</th>
<th>(e(\bullet, \bullet))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(S)</td>
<td>(L^+)</td>
<td></td>
<td></td>
<td>(S)</td>
</tr>
<tr>
<td>(b)</td>
<td></td>
<td>(S)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td></td>
<td></td>
<td>(S)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td></td>
<td></td>
<td></td>
<td>(S)</td>
<td></td>
</tr>
<tr>
<td>(e)</td>
<td>(S)</td>
<td>(L^+)</td>
<td></td>
<td></td>
<td>(S)</td>
</tr>
</tbody>
</table>

Figure 3.13: Example application of nil estimate.
\[ \forall h_i \in H'. \]
\[ p'.IsNode[h_i] \leftrightarrow \]
\[ \text{if } h_i = A \text{ then} \]
\[ (p.IsNode[B].left, \odot \odot) \]
\[ \text{else if } h_i = B \text{ then} \]
\[ (\bullet, (p.IsNode[B].left) (p.IsNode[B].right)) \]
\[ \text{else} \]
\[ p.IsNode[h_i]; \]

Figure 3.14: Nil estimate for \( A := B.left \).

### 3.4 Analysing Simple Statements (\textit{DAGS})

The previous section provided an overview of an interference analysis method for \textit{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 \textit{TREE}; thus if a possible \textit{DAG} is detected, the analysis must fail. In this section an extension of the \textit{TREE} analysis to handle \textit{DAGS} is presented.

Simple imperative programs that process \textit{TREES} may also create \textit{DAGS} temporarily. For example, consider the program in figure 3.16. This program operates on nodes with three handle fields, \textit{left}, \textit{middle}, and \textit{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 \textit{TREE}, there are intermediate points (B and C) where the data structure is a \textit{DAG}.

In order to analyse \textit{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 \textit{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 \textit{DAGS} are straightforward extensions to those for \textit{TREES}. Since the number of dag nodes created is not known statically, dag nodes are approximated by associating a \textit{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 \textit{TREE} case. In the case of the example
\[ A.left := \text{nil} \]

\[ \forall h_i \in H'. \]
\[ p'.IsNode[h_i] \mapsto \]
\[ \text{if } h_i = A \text{ then} \]
\[ (\bullet, o \ (p.\text{IsNode}[A].right)) \]
\[ \text{else if path}(h_i, S, A, p) \text{ then} \]
\[ (p.\text{IsNode}[h_i].self, o \ (p.\text{IsNode}[h_i].right)) \]
\[ \text{else if path}(h_i, S?, A, p) \text{ then} \]
\[ (p.\text{IsNode}[h_i].self, \]
\[ (o \cap (p.\text{IsNode}[h_i].left)) \]
\[ (p.\text{IsNode}[h_i].right) \]
\[ ) \]
\[ \text{else} \]
\[ p.\text{IsNode}[h_i]; \]

\[ A.left := B \]

\[ \forall h_i \in H'. \]
\[ p'.IsNode[h_i] \mapsto \]
\[ \text{if } h_i = A \text{ then} \]
\[ (\bullet, (p.\text{IsNode}[B].self) \ (p.\text{IsNode}[A].right)) \]
\[ \text{else if path}(h_i, S, A, p) \text{ then} \]
\[ (p.\text{IsNode}[h_i].self, (p.\text{IsNode}[B].self) \ (p.\text{IsNode}[h_i].right)) \]
\[ \text{else if path}(h_i, S?, A, p) \text{ then} \]
\[ (p.\text{IsNode}[h_i].self, \]
\[ ((p.\text{IsNode}[B].self) \cap (p.\text{IsNode}[h_i].left)) \]
\[ (p.\text{IsNode}[h_i].right)) \]
\[ ) \]
\[ \text{else} \]
\[ p.\text{IsNode}[h_i]; \]

Figure 3.15: Nil estimate for \( A.left := \text{nil} \) 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 PA }
    h.left := m;
    { ⇐ PROGRAM POINT B – path matrix PB }
    h.middle := r;
    { ⇐ PROGRAM POINT C – path matrix PC }
    h.right := l;
    { ⇐ PROGRAM POINT D – path matrix PD }
    { use h,m,l,r .... } 
end;

\[ \begin{array}{|c|c|c|c|c|}
\hline
& h & l & m & r \\
\hline
h & S & L^1 & M^1 & R^1 \\
\hline
l & S & & & \\
\hline
m & & S & & \\
\hline
r & & & S & \\
\hline
\end{array} \]

\[ \begin{array}{|c|c|c|c|c|}
\hline
& h & l & m & r \\
\hline
h & S & L^1, M^1 & R^1 & L^1, M^1 \\
\hline
l & & S & & \\
\hline
m & & S & & \\
\hline
r & & & S & \\
\hline
m:0@1 & & S & & \\
\hline
\end{array} \]

\[ \begin{array}{|c|c|c|c|c|}
\hline
& h & l & m & r \\
\hline
h & S & L^1, M^1 & R^1 & M^1, R^1 \\
\hline
l & & S & & \\
\hline
m & & S & & \\
\hline
r & & & S & \\
\hline
r:1@1 & & S & & \\
\hline
\end{array} \]

\[ \begin{array}{|c|c|c|c|}
\hline
& h & l & m & r \\
\hline
h & S & R^1 & L^1 & M^1 \\
\hline
l & & S & & \\
\hline
m & & S & & \\
\hline
r & & & S & \\
\hline
\end{array} \]

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.

\[
\text{IncrRef}(n) = \\
\quad \text{if } n = ++ \text{ then } ++ \text{ else } n+1
\]

\[
\text{DecrRef}(n) = \\
\quad \text{if } n = ++ \text{ then } ++ \text{ else } n-1
\]

\[
\text{grow_merge}(n,m) = \\
\quad \text{if } n = ++ \text{ or } m = ++ \text{ then } ++ \\
\quad \text{else} \\
\quad \quad \text{if } n = m \text{ then } n \text{ else } ++
\]

\[
\text{max_merge}(n,m) = \\
\quad \text{if } n = ++ \text{ or } m = ++ \text{ then } ++ \text{ else } \text{max}(n,m)
\]

---

**Figure 3.17**: Operations on abstract reference counts.

### 3.4.2 Analysis Rules for **DAG** estimates.

Handle assignment statements of the form: $A := \text{nil}$, $A := \text{new}()$, 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 $A.left := \text{nil}$. 
\[ A := B..left \]
\[
\forall h_i, h_j \in H'. \\
p'[h_i, h_j] \mapsto \\
\text{if } h_i = A \text{ then} \\
\quad \text{if } h_j = A \text{ then} \\
\quad \{ S \} \\
\quad \text{else if } h_j = B \text{ then} \\
\quad \{ \} \\
\quad \text{else} \\
\quad \quad \text{map below_rule } p[B, h_j] \\
\quad \text{else if } h_j = A \text{ then} \\
\quad \quad \text{if } h_i = B \text{ then} \\
\quad \quad \{ L' \} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{map above_rule } p[h_i, B] \\
\quad \quad \text{else} \\
\quad \quad \quad p[h_i, h_j] \\
\forall d_i \in \text{dags}(H) . \\
\text{if path}(B, L, d_i, p) \text{ then} \\
\quad \text{inherit_entries}(p, d_i, A, p') \\
\text{else if path}(B, L^?, d_i, p) \text{ then} \\
\quad \text{inherit_possible_entries}(p, d_i, A, p')
\]

Figure 3.18: Extended rule for \( A := B..left \).
\( A.\text{left} := \text{nil} \)

\[ \forall h_i, h_j \in H'. \quad \text{p}[h_i, h_j] \mapsto \]
\[ \text{if } \text{path}(h_i, *, A, p) \text{ and} \]
\[ \text{path}(A, L*, h_j, p) \text{ and} \]
\[ \text{not (dag.between}(A, L, h_j, p)) \text{ then} \]
\[ \{ \} \]
\[ \text{else if } \text{path}(h_i, ?, A, p) \text{ and} \]
\[ \text{path}(A, L?, h_j, p) \text{ then} \]
\[ \text{if } \text{path}(h_i, S, A, p) \text{ then} \]
\[ \text{remove.head}(L, p[h_i, h_j]) \]
\[ \text{else if } \text{path}(A, L, h_j, p) \text{ then} \]
\[ \text{remove.tail}(L, p[h_i, h_j]) \]
\[ \text{else} \]
\[ \text{make.possible}(p[h_i, h_j]) \]
\[ \text{else} \]
\[ p[h_i, h_j]; \]

\[ \forall d_i \in \text{dags}(H). \]
\[ \text{if } \text{path}(A, L, d_i, p) \text{ or} \]
\[ ( \text{path}(A, L*, d_i, p) \text{ and} \]
\[ \text{not (handle.between}(A, L, d_i, p)) \text{ then} \]
\[ p'.\text{RefCnt}[d_i] \mapsto \text{DecrRef}(p.\text{RefCnt}[d_i]) \]
\[ \text{else} \]
\[ p'.\text{RefCnt}[d_i] \mapsto p.\text{RefCnt}[d_i] \]

Figure 3.19: Extended rule for \( A.\text{left} := \text{nil} \).
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 $A.left := \text{nil}$ 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 $A.left := \text{nil}$, 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, $\text{dagsname}(B)$. 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 $p''[h_i, \text{dagsname}(B)]$ 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 $p'[h_i, \text{dagsname}(B)]$ 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 ($\Rightarrow$) 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 $s_1; s_2; \ldots; s_n$ 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 ($expr$) then ($stmt_1$) else ($stmt_2$), a pair of path matrices ($p_{\text{then}}, p_{\text{else}}$) is produced as follows: the statement analysis function is applied to $p$ and $stmt_1$ to produce $p_{\text{then}}$, and the statement analysis function is applied to $p$ and $stmt_2$ to produce $p_{\text{else}}$. The analysis may continue with the pair of path matrices ($p_{\text{then}}, p_{\text{else}}$), or with one merged path matrix $p_{\text{then}} \Rightarrow p_{\text{else}}$. 
3.5 Analysing Compound Statements

\[ A.left := B \]

\[ p' \mapsto f(A.left:=nil,p); \]
\[ \text{if } \text{path}(B,*,?,A,p') \text{ then} \]
\[ \text{possible_cycle}() \]
\[ \text{else} \]
\[ \forall h_i, h_j \in H'. \]
\[ p''[h_i, h_j] \mapsto \]
\[ \text{if } \text{path}(h_i,*,?,A,p') \]
\[ \text{and } \text{path}(B,*,?,h_j,p') \text{ then} \]
\[ \text{if } \text{is_node}(B) \text{ then} \]
\[ p'[h_i,h_j] \cup \]
\[ (\text{connect_all} \]
\[ (p'[h_i, A], \]
\[ (L', \text{true}), \]
\[ p'[B, h_j] \]
\[ ) \]
\[ ) \]
\[ \text{else} \]
\[ p'[h_i,h_j] \cup \]
\[ (\text{connect_all} \]
\[ (p'[h_i, A], \]
\[ (L', \text{true}), \]
\[ \text{make_possible}(p'[B, h_j]) \]
\[ ) \]
\[ ) \]
\[ \text{else} \]
\[ p'[h_i, h_j]; \]
\[ \text{if } \text{handle_above}(B,p') \text{ then} \]
\[ \text{if } \text{unused_dag}(\text{dag_name}(B),p') \text{ then} \]
\[ p''.\text{RefCnt}[\text{dag_name}(B)] \mapsto \mathbb{E}; \]
\[ \text{init_entries}(p'',B,\text{dag_name}(B)) \]
\[ \text{else} \]
\[ p''.\text{RefCnt}[\text{dag_name}(B)] \mapsto ++; \]
\[ \text{merge_entries}(p'',B,\text{dag_name}(B)); \]
\[ \forall d_i \in \text{dags}(H) . \]
\[ \text{if } \text{path}(B,?,d_i,p') \text{ then} \]
\[ p''.\text{RefCnt}[d_i] \mapsto \text{IncrRef}(p'.\text{RefCnt}[d_i]) \]

Figure 3.20: Extended rule for \( A.left := B \).
Given an input path matrix and a while loop of the form while \(<expr>\) do \(<stmt>\), the analysis produces a pair of path matrices, \(⟨p_0, p_+⟩\). The path matrix \(p_0\) represents zero iterations of the while loop and is equal to 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.

\[
l := h;
\]

\[
\text{while } l.left \neq \text{nil do}
\]

\[
l := l.left
\]

\[
\begin{array}{c|c|c}
\hline
h & l & S \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
\hline
h & l & S \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
\hline
h & l & S \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
\hline
h & l & S \\
\hline
\end{array}
\]

\[
p_0 
\quad 
p_1 
\quad 
p_2 
\quad 
p_3 = p_+
\]

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 (\(\lor^\nu\)) operation. The merge path matrix, \(p_{\text{merged}} = p_1 \bowtie p_2\), is then calculated by the item-wise and-merging given by: \(p_{\text{merged}}[h_i, h_j] = p_1[h_i, h_j] \lor^\nu p_2[h_i, h_j]\). 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-
### 3.5 Analysing Compound Statements

#### Original path matrices

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1 R^+$, $L^1 L^+$</td>
<td>$L^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1 D^+$, $L^1$</td>
<td>$L^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Simplified matrices, each item squashed

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1 D^+$</td>
<td>$L^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>$S$</td>
<td>$S$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1 D^+$, $D^1$</td>
<td>$R^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Result of merging the two simplified matrices

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$S$</td>
<td>$L^1 D^+$</td>
<td>$D^1$</td>
</tr>
<tr>
<td>b</td>
<td>$S$</td>
<td>$S$</td>
<td>$S$</td>
</tr>
<tr>
<td>c</td>
<td>$S$</td>
<td>$S$</td>
<td>$S$</td>
</tr>
</tbody>
</table>

Figure 3.22: An example of the merge operation.
mations for each element \( p_1(h_i, h_j), p_2(h_i, h_j), \ldots, p_n(h_i, h_j) \) 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 \( f(h_1, \ldots, h_n) \), 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 \( \text{SwapChildren}(\text{root}) \) in figure 3.23. In this example, the call to \( \text{SwapChildren} \) changes the relationships between handles \( \text{root}, \text{Lchild} \) and \( \text{r_child} \) by swapping the left and right sub-trees of \( \text{root} \). Figure 3.23 illustrates four stages of the path matrix computation. \( P_A \) represents the path matrix at the point just before the call \( \text{SwapChildren}(\text{root}) \). \( P_B \), the path matrix at the beginning of the body of \( \text{SwapChildren} \), combines the relationships of calling context handles and the parameter handles. In our example, the handles \( \text{Lchild} \) and \( \text{r_child} \) must be included in the calling context since they are below the handle \( \text{root} \). In \( P_B \), \( \text{Lchild}@2 \) and \( \text{r_child}@2 \) capture the context information of the call, \( h*1 \) is used as a symbolic name for the argument handle (\( \text{root} \)), 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 \( \text{TREE} \), 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
3.6 Analysing Procedure Calls

program DemoProcCall

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

procedure SwapChildren(h: handle)
  l, r: handle
begin
  \{ \Leftarrow \textit{PROGRAM POINT B} \rightarrow \textit{path matrix} \ P_B \ \}\n  l := h.left;
  r := h.right;
  h.left := r;
  h.right := l
  \{ \Leftarrow \textit{PROGRAM POINT C} \rightarrow \textit{path matrix} \ P_C \ \}\nend;

procedure main()
  root, l.child, r.child: handle
begin
  \{ .... create a tree at root .... \}
  l.child := root.left;
  r.child := root.right;
  \{ \Leftarrow \textit{PROGRAM POINT A} \rightarrow \textit{path matrix} \ P_A \ \}\n  SwapChildren(root);
  \{ \Leftarrow \textit{PROGRAM POINT D} \rightarrow \textit{path matrix} \ P_D \ \}\n  \{ ... use \ l.child, \ r.child, \ root ... \}
end

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
& $P_A$ & $P_B$ \\
\hline
\textbf{root} & S & L$^1$ \\
\textbf{l.child} & S & \textbf{r.child} \\
\textbf{r.child} & S & \textbf{S} \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|c|c|}
\hline
& L.child@2 & \textbf{r.child@2} & h$^1$ & h \\
\hline
\textbf{L.child@2} & S & & & \\
\textbf{r.child@2} & S & & & \\
\textbf{h$^1$} & L$^1$ & R$^1$ & S & S \\
\textbf{h} & L$^1$ & R$^1$ & S & S \\
\hline
\end{tabular}
\begin{tabular}{|c|c|}
\hline
L.child@2 & S \\
\textbf{r.child@2} & S \\
\textbf{h$^1$} & L$^1$ \\
\hline
\end{tabular}
\begin{tabular}{|c|c|c|}
\hline
\textbf{root} & S & R$^1$ \\
\textbf{L.child} & S & L$^1$ \\
\textbf{r.child} & S & \textbf{S} \\
\hline
\end{tabular}
\caption{Example of a procedure call.}
\end{figure}
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;

\[
\begin{array}{c|c}
\hline
P_A & \text{root} \\
\hline
\text{root} & S \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
\hline
P_B & h+3 & \text{m} \\
\hline
h+3 & S & S \\
m & S & S \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c}
\hline
P_C & h+3 & m \\
\hline
h+3 & S & (S + L^+) \\
m & S? & S \\
\hline
\end{array}
\quad
\begin{array}{c|c}
\hline
P_D & \text{min} \\
\hline
\text{root} & S & (S + L^+) \\
\text{min} & S? & S \\
\hline
\end{array}
\]

Figure 3.24: Example of a function call.
3.6 Analysing Procedure Calls

relationships are encoded in the path matrix valid at end of the body of $f$. As an example, consider the call $\text{min} := \text{FindMin(root)}$ in figure 3.24. At program point $P_C$, $h * 3$ is a symbolic name corresponding to $\text{root}$ 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 $\text{root}$, and $m$ with $\text{min}$.

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 $p_{\text{in}_i}, p_{\text{out}_j}$. The path matrix $p_{\text{in}_i}$ represents the merge of all possible path matrices at the beginning of the body of $f$, while $p_{\text{out}_j}$ represents the merge of all possible path matrices at the end of the body of $f$. The iterative approximation terminates when $p_{\text{in}_i} = p_{\text{in}_{i+1}}$ and $p_{\text{out}_j} = p_{\text{out}_{j+1}}$.

As an example of the iterative approximation for the recursive procedures, consider the approximation for $\text{reverse}$ given in figure 3.25. In this case, the first input approximation $p_{\text{in}_1}$ is due to the call $\text{reverse(root)}$ and the initial output approximation $p_{\text{out}_0}$ is UNDEFINED. The first defined output approximation $p_{\text{out}_1}$ is computed by analysing all non-recursive paths in the body of $\text{reverse}$. In this case the only non-recursive path is the empty statement, and thus $p_{\text{out}_1}$ is trivial to compute from $p_{\text{in}_1}$. The second input approximation $p_{\text{in}_2}$ is due to the recursive call $\text{reverse(l)}$. 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 $\text{reverse(l)}$ gives $p_{\text{in}_2}$. Since $p_{\text{in}_2}$ is equal to $p_{\text{in}_1}$, $p_{\text{out}_1}$ can be used as an estimate of the effect of the call to $\text{reverse(l)}$, and similarly for the call to $\text{reverse(r)}$. The resulting path matrix at the end of the body of $\text{reverse}$ is $p_{\text{out}_2}$. Since both $p_{\text{in}_1} = p_{\text{out}_1}$, and $p_{\text{out}_1} = p_{\text{out}_2}$, the iterative approximation terminates and $p_{\text{out}_2}$ is used to build the result of the call.

4.1 A Formal Description of SIL

\[ E : \text{Expression} \rightarrow (\text{Env} \rightarrow (\text{Mem} \rightarrow \text{Value})) \]
\[ M : \text{Statement} \rightarrow (\text{Env} \rightarrow (\text{Mem} \rightarrow \text{Mem})) \]
\[ D : \text{Definition} \rightarrow (\text{Env} \rightarrow \text{Env}) \]
\[ P : \text{Program} \rightarrow (\text{File} \rightarrow \text{File} + \text{error}) \]
nodedef left, right: handle,

procedure reverse(h:handle)
  l, r: handle
begin
  { ≡ p.in₁ (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  
  end  
  { ≡ p.out₁ (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;

\[
[p.\text{in}_1, p.\text{out}_0] = \begin{bmatrix}
  \begin{array}{ccc}
    h^{*1} & h \\
    h^{*1} & S & S \\
    h & S & S \\
  \end{array} & , & \text{UNDEFINED}
\end{bmatrix}
\]

\[
[p.\text{in}_1, p.\text{out}_1] = \begin{bmatrix}
  \begin{array}{ccc}
    h^{*1} & h \\
    h^{*1} & S & S \\
    h & S & S \\
  \end{array} & , & \begin{array}{c}
    h^{*1} \\
    h^{*1} & S \\
  \end{array}
\end{bmatrix}
\]

\[
[p.\text{in}_2, p.\text{out}_2] = \begin{bmatrix}
  \begin{array}{ccc}
    h^{*1} & h \\
    h^{*1} & S & S \\
    h & S & S \\
  \end{array} & , & \begin{array}{c}
    h^{*1} \\
    h^{*1} & S \\
  \end{array}
\end{bmatrix}
\]

Figure 3.25: Estimating the path matrices for reverse.
program RecursiveFunction

nodeleft, right: handle; value: int end;

function FindMin(h: handle) handle
  t, m: handle
begin
{ ⇐ p.in₁ (a merge of all input path matrices) }
if h.left = nil then
  m := h
else
  begin
    t := h.left;
    { ⇐ pₐ (before the recursive call) }
    m := FindMin(t)
  end
end
{ ⇐ p.out₁ (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.in₁, p.out₀] =
\[
\begin{bmatrix}
  \ h⁺ S & h \\
  \ h⁺ S & S \\
  \ h & S & S \\
\end{bmatrix}, UNDEFINED
\]

[p.in₁, p.out₁] =
\[
\begin{bmatrix}
  \ h⁺ S & h \\
  \ h⁺ S & S \\
  \ h & S & S \\
\end{bmatrix},
\begin{bmatrix}
  \ h⁺ S & m \\
  \ h⁺ S & S \\
  \ h & S & S \\
\end{bmatrix},
\begin{bmatrix}
  \ h⁺ S & m \\
  \ h⁺ S & S \\
  \ h & S & S \\
\end{bmatrix}
\]
Chapter 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: $E$ for expressions, $M$ for statements, $D$ for definitions, and $P$ 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
\[ E : \text{Expression} \rightarrow (\text{Env} \rightarrow (\text{Mem} \rightarrow \text{Value})) \]
\[ M : \text{Statement} \rightarrow (\text{Env} \rightarrow (\text{Mem} \rightarrow \text{Mem})) \]
\[ D : \text{Definition} \rightarrow (\text{Env} \rightarrow \text{Env}) \]
\[ P : \text{Program} \rightarrow (\text{File} \rightarrow \text{File} + \text{error}) \]

where

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

Figure 4.1: Semantic domains for SIL.

Figure 4.2: The model for the
Description of Interference Analysis

The tree in figure 4.2 gives a more abstract view of the environment and memory. In chapter 3 we discussed the analysis as estimating the paths between each pair of live handles $h_i$ and $h_j$. In a 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 $\beta$ from handle $A$ to handle $B$ if $\beta$ 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).

![Figure 4.3: A tree and the corresponding path matrix](image)

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 $\oplus$ superscripts (solid lines) represents length approximation.

4.2 Path Matrices Revisited

Given a domain for path expressions with length $n$ 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, $p_1 \preceq p_2$, must obey the following condition:

**Condition 1.** $(p_1 \preceq p_2) \subseteq (p_1 \cup p_2)$

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 $p_1 \preceq p_2$ then the relation $\preceq$ is given by: (1) $p_1 \preceq p_2$, (2) $p_1 < p_2$, or $p_1 \preceq p_2$

The ordering can be extended
Figure 4.5: Partial ordering of path expressions of length 1 or 2
\[ M[ A := \text{nil} ] aenv \ p = p', \text{ where} \]
\[ \forall h_i, h_j \in H'. \]
\[ p'[h_i, h_j] \mapsto \]
\[ \text{if } h_i = A \text{ and } h_j = A \text{ then} \]
\[ \{ S \} \]
\[ \text{else if } h_i = A \text{ or } h_j = A \text{ then} \]
\[ \{ \} \]
\[ \text{else} \]
\[ p[h_i, h_j] \]
\[ \forall h_i \in H'. \]
\[ p'.IsNode[h_i] \mapsto \]
\[ \text{if } h_i = A \text{ then } (0, \infty) \text{ else } p.IsNode[h_i]; \]

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

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

Figure 4.9: Abstract semantic functions for handle assignment statements.
\[ A[ \text{begin end }] \text{ aenv } p = p \]

\[ A[ \text{begin } s_1 ; s_2 ; \ldots ; s_n \text{ end }] \text{ aenv } p =\]
\[ A[ \text{begin } s_2 ; \ldots ; s_n \text{ end }] \text{ aenv } (A[ s_1 ] \text{ aenv } p) \]

\[ A[ \text{if } \text{ exp then } s_1 \text{ else } s_2 ] \text{ aenv } p =\]
\[ (A[ s_1 ] \text{ aenv } p) \triangleright (A[ s_2 ] \text{ aenv } p) \]

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

\[ A[ \text{repeat } s \text{ until } e ] \text{ aenv } p = A (A[ s ] ] \text{ aenv } p) \]
where
\[ A = \lambda p . \]
\[ \text{let} \]
\[ \text{new}_p = p \triangleright (A[ s ] \text{ aenv } p) \]
\[ \text{in} \]
\[ \text{if } \text{ new}_p = p \text{ then } p \text{ else } A \text{ new}_p \]

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.

\[
\mathcal{M}[\text{while } e \text{ do } s]\ env\ mem = fIx(W)\ mem
\]

where

\[
W = \lambda T . \lambda m .
\]

\[
\text{if } \mathcal{E}[e] env\ m \text{ then } T(\mathcal{M}[s]\ env\ m) \text{ else } m
\]

\[
\mathcal{A}[\text{while } e \text{ do } s]\ aenv\ p = A\ p
\]

where

\[
A = \lambda p .
\]

\[
\text{let}
\]

\[
\text{new}\_p = (\mathcal{A}[s]\ aenv\ p) \cdot p
\]

\[
\text{in}
\]

\[
\text{if } new\_p = p \text{ then } p \text{ else } A\ new\_p
\]

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.

\[
\mathcal{A}[q(a_1, a_2, \ldots, a_n)] aenv\ P =
\]

\[
\text{let } h_1 \ldots h_m = \text{handles}(a_1 \ldots a_n) \text{ in}
\]

\[
aenv(q.D) h_1 \ldots h_m aenv\ P
\]

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.
\[ D[ \text{proc } q \ ( x_1:t ; \ldots ; x_n:t ) \ l_1:t ; \ldots ; l_m:t ; \ s ] \ env = \ env\ [ q \mapsto Q ] \]

where \( Q = \lambda v_1 . \lambda v_2 . \ldots . \lambda v_n . \lambda (\text{store, heap, sp, fl, in, out}) . \)

\[ \text{let} \]
\[ \text{local.env} = \]
\[ \ env[ x_1 \mapsto sp + 1, \ldots , x_n \mapsto sp + n, \]
\[ \ l_1 \mapsto sp + n + 1, \ldots , l_n \mapsto sp + n + m ] \]
\[ \text{local.store} = \]
\[ \text{store[local.env}(x_1) \mapsto v_1, \ldots, \text{local.env}(x_n) \mapsto v_n, \]
\[ \text{local.env}(l_1) \mapsto \text{undef}, \ldots, \text{local.env}(l_m) \mapsto \text{undef} ] \]
\[ \text{new.mem} = \mathcal{M}[ s ] \text{local.env}(\text{local.store, heap, sp+m+n, fl, in, out}) \]

\[ \text{in} \]
\[ \text{if new.mem = error then} \]
\[ \text{error} \]
\[ \text{else} \]
\[ \text{let} \]
\[ (\text{local.store}', \text{heap}', sp', fl', in', out') = \text{new.mem} \]
\[ \text{in} \]
\[ (\text{store, heap, sp, fl, in, out}) \]

\[ F[ \text{proc } q \ ( x_1:\text{handle} ; \ldots ; x_n:\text{handle} ) \]
\[ l_1:\text{handle} ; \ldots ; l_m:\text{handle} ; \ s ] \ aenv = \ aenv\ [ q.D \mapsto D ] \]

where
\[ D = \lambda h_1 . \ldots . \lambda h_n . \lambda e . \lambda P . \]

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

\[ F = \lambda e . \lambda P . \]

\[ \text{let} \]
\[ P'_1 = \text{insert}([ x_1 \mapsto x_1, \ldots, x_n \mapsto x_n, l_1 \mapsto \text{nil}, \ldots, l_m \mapsto \text{nil} ], P) \]
\[ P'_2 = \mathcal{A}[ s ] \ aenv \ P'_1 \]
\[ \text{in} \]
\[ \text{remove}([ x_1, \ldots, x_n, l_1, \ldots, l_m ], P'_2) \]

Figure 4.13: Semantic functions for non-recursive procedures.
The first step in $D$ splits the input path matrix $P$ into two path matrices $P_{\text{inv}}$ and $P_{\text{var}}$. $P_{\text{inv}}$ represents the relationships between all handles that cannot reached through paths starting from any of the argument handles, while $P_{\text{var}}$ 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 $P_{\text{var}}$.

The second step in $D$ is to create $P_1$ from $P_{\text{var}}$ 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 $P_{\text{inv}}$ and $P_3$. The joining of $P_{\text{inv}}$ 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 $P_{\text{inv}}$, 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 $P_{\text{inv}}$, 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 $\text{insert}$ 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 $\text{base}$.

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 $r.\text{est}$ and a boolean flag $r.\text{new.}\text{est}$

\(^2\)We assume that all parameter names and local variables are uniquely named (implemented by tagging them with their procedure number).
\[ F = \lambda e . \lambda P . \]
\[
\text{let } (P_{\text{in}}, P_{\text{out}}) = e(r.\text{est}) \text{ in }
\]
\[
\text{if } e(r.\text{new.est}) \text{ then }
\{ \text{ ----- CASE 1 (generalize output?) ----- } \}
\]
\[
\text{let }
\]
\[
P_1 = \text{insert}([x_1 \mapsto x_1, \ldots, x_n \mapsto x_n, l_1 \mapsto \text{nil}, \ldots, l_m \mapsto \text{nil}], P)
\]
\[
P_2 = A_{\left[ s \right]} e(r.\text{new.est} \mapsto \text{false}) P_1
\]
\[
NP_{\text{out}} = \text{remove}([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2)
\]
\[
\text{in }
\]
\[
\text{if } NP_{\text{out}} \subseteq P_{\text{out}} \text{ then }
P_{\text{out}} \{ \text{ -- 1(a) -- } \}
\]
\[
\text{else }
\]
\[
F e(r.\text{est} \mapsto (P, (NP_{\text{out}} \bowtie P_{\text{out}}))) P \{ \text{ -- 1(b) -- } \}
\]
\[
\text{else }
\]
\[
\{ \text{ ----- CASE 2 (generalize input?) ----- } \}
\]
\[
\text{if } P \subseteq P_{\text{in}} \text{ then }
P_{\text{out}} \{ \text{ -- 2(a) -- } \}
\]
\[
\text{else }
\]
\[
\text{let }
\]
\[
NP_{\text{in}} = (P \bowtie P_{\text{in}})
\]
\[
in \{ \text{ -- 2(b) -- } \}
\]
\[
F e(r.\text{new.est} \mapsto \text{true}, r.\text{est} \mapsto (NP_{\text{in}}, \text{base}(NP_{\text{in}}))) NP_{\text{in}}
\]
\[
\]
\[
base = \lambda P .
\]
\[
\text{let }
\]
\[
P_1 = \text{insert}([x_1 \mapsto x_1, \ldots, x_n \mapsto x_n, l_1 \mapsto \text{nil}, \ldots, l_m \mapsto \text{nil}], P)
\]
\[
P_2 = A_{\left[ s \odot r \right]} aenv P_1
\]
\[
\text{in }
\]
\[
\text{remove}([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2)
\]

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 \((\Omega, \Omega)\), and the flag \(r_{.newest}\) 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 \((r_{.newest} = true)\). 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 $h_j$, $H[h_i \sim h_j]$ denotes the set of paths connecting $h_i$ to $h_j$ in $H$. That is, $\forall r \in (L + R)^* . \text{if } r \text{ connects } h_i \text{ and } h_j \text{ in } H$, then $r \in H[h_i \sim h_j]$.

**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 $X \preceq H[h_i \sim h_j]$, if:

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

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 $P \preceq (M, env)$ if:
4.4 Soundness Theorem

(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. \( \forall l_i, l_j \in L \cdot P[l_i, l_j] \leq H[\text{store}(\text{env}(l_i)) \rightarrow \text{store}(\text{env}(l_j))] \)
2. \( \forall l_i \in L, d_j \in D \cdot P[l_i, x_j] \leq H[\text{store}(\text{env}(l_i)) \rightarrow d_j] \)
3. \( \forall d_i \in D, l_j \in L \cdot P[x_i, l_j] \leq H[d_i \rightarrow \text{store}(\text{env}(l_j))] \)
4. \( \forall d_i, d_j \in D \cdot P[x_i, x_j] \leq H[d_i \rightarrow d_j] \)
5. \( \forall d_i \in D \cdot \text{RefCnt}(d_i) \leq P.\text{RefCnt}[x_i] \)
6. \( \forall l_i \in L \cdot \text{Nilness}(\text{env}(l_i)) \leq P.\text{Nilness}[l_i] \)

Condition \( I \) states that \( P \) must contain the relationships for all live handles in \( \text{env} \). 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 \( (M, \text{env}) \).

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 \( X \preceq H[h_i \leadsto h_j] \) and \( X \subseteq X' \), then \( X' \preceq H[h_i \leadsto h_j] \).

**Proof:** Suppose that \( r \) is a path in \( H[h_i \leadsto h_j] \). Suppose also that \( X = \left< (d_1, \ldots, d_k), (e_1?, \ldots, e_l?) \right> \), and \( X' = \left< (f_1, \ldots, f_m), (g_1?, \ldots, g_n?) \right> \). Because \( X \preceq H[h_i \leadsto h_j] \) holds we know that \( r \in (\cup_i d_i) \cup (\cup_j e_j) \). The first condition in definition 11 state that \( (\cup_i d_i) \cup (\cup_j e_j) \subseteq (\cup_i f_i) \cup (\cup_j g_j) \), thus \( r \in ((\cup_i f_i) \cup (\cup_j g_j)) \).

Consider any \( f_i \), by the second condition in the definition of \( \preceq \) there is some \( d_j \subseteq f_i \). By the second condition in the definition of \( \preceq \), 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 \( X' \preceq H[h_i, h_j] \) to hold. \( \blacksquare \)

**Lemma 2.** If \( P \preceq (M, \text{env}) \) and \( P \subseteq P' \) then \( P' \preceq (M, \text{env}) \).

**Proof:** Suppose \( P, P', M \) and \( \text{env} \) are as in the statement. Since \( P \) and \( P' \) must be defined on the same handles, condition \( I \) holds for \( P' \preceq (M, \text{env}) \). 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 \( \forall h_i, h_j \in (L \cup X), P[h_i, h_j] \subseteq P'[h_i, h_j] \). Thus, using lemma 1 we know that the \( \preceq \) 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 $\prec$ 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 $P \prec (M, env)$ we have $RP \prec (TM, env)$ 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 $aenv \prec env$ 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 $P \prec (M, env)$ then $[aenv(q)h_1\ldots h_kP] \prec ([env(q)(env(h_1))\ldots(env(h_k)))M, env)$.

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 $\forall i. R \prec T_i$ then $R \prec \sqcup T_i$. If $R_i$ is an increasing sequence of path matrix transformers and $T$ is a memory transformer such that $\forall i. R_i \prec T$ then $\sqcup R_i \prec T$.

**Proof:** Suppose that $P, M$ and $env$ satisfy $P \prec (M, env)$. 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, $(\sqcup T_i)M = T_kM$. Since $R$ is safe for $T_k$ we immediately have $RP \prec (T_kM, env)$. 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.
4.4 Soundness Theorem

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

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

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

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

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 $k$th 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, $\triangleright$, to determine an upper bound. We call the successive stages in the computation of $\mathcal{A}[\text{while } e \text{ do } s], A^{(k)}$ in analogy with the approximants to the while loop.
\[ W^{(0)} = \lambda m . \perp \]
\[ W^{(1)} = \lambda m . \text{if } \epsilon[e] env m \text{ then } \perp \text{ else } m \]
\[ W^{(k+2)} = \lambda m . \text{if } \epsilon[e] env m \text{ then } W^{(k+1)} (\mathcal{M}[s] env m) \text{ else } m \]

\[ A^{(0)} = \lambda p . \ p \]
\[ A^{(k+1)} = \lambda p . \]
\[ \text{let} \]
\[ \text{new}_p = (A[s] aenv p) \Rightarrow p \]
\[ \text{in} \]
\[ \text{if } \text{new}_p = p \text{ then } p \text{ else } A^{(k)} \text{ new}_p \]

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 \( k \), \( A^{(k)} \triangleleft W^{(k+1)} \). This proceeds by induction on \( k \). The base case, \( k = 0 \), is immediate from the definitions. For the inductive case we assume that

\[ A^{(k)} P \triangleleft (W^{(k+1)} M, env) \]

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

\[ A^{(k+1)} P \triangleleft (W^{(k+2)} M, env) \]

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

\[ (FIH) : A^{(k)} P \triangleleft (W^{(k+1)} M, env) \]
\[ (SIH) : A[s] aenv P \triangleleft (\mathcal{M}[s] env M), env) \]

We will consider two cases, \( \epsilon[e] env M = \text{false} \), and \( \epsilon[e] env M = \text{true} \).

Suppose that \( \epsilon[e] env M \) 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 \( P \subseteq A^{(k+1)} P \). Therefore, \( A^{(k+1)} P \triangleleft (M, env) \), which can be restated as \( A^{(k+1)} P \triangleleft (W^{(k+2)} M, env) \).

Now suppose that \( \epsilon[e] env m \) is true.

By SIH we have \( A[s] aenv P \triangleleft (\mathcal{M}[s] env M, env) \).

Consider new\(_p\) in the body of \( A^{(k+1)} \). Since new\(_p\) is the result of \( P \Rightarrow (A[s] aenv P) \), we know that \( A[s] aenv P \subseteq \text{new}_p \), and therefore \( \text{new}_p \triangleleft (\mathcal{M}[s] env M, env) \).
4.4 Soundness Theorem

Now let us consider the result of $A^{(k+1)}P$. If $new \cdot p = P$, then the result of $A^{(k+1)}P$ is $P = A^kP = A^k(new \cdot p)$.

If $new \cdot p \neq P$, then the result of $A^{(k+1)}P$ is also $A^k(new \cdot p)$.

Applying $FIH$ to $new \cdot p \triangleleft (M[s] \ env \ M, env)$ gives

$$A^k(new \cdot p) \triangleleft (W^{(k+1)}(M[s] \ env \ M, env),$$

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

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 $P[X, Y] = \{\}$ 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 $P \triangleleft (M, env)$. Suppose that $T_A$ is a path matrix transformer and $T_M$ is a memory transformer with $T_A \triangleleft T_M$.

Suppose that all the handles accessed by $T_A$ are included in a set of handles $\chi$. 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 $T'_A \triangleleft T_M$.

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 $\chi$ since $P \triangleleft (M, env)$. 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 $\chi$. This would mean that $Y$ is below some node in $\chi$ which is impossible. Thus $T'_A$ gives the right relationships between pairs of handles both of which are in $\chi$ or neither of which are in $\chi$. If we have a handle, $X$, above $\chi$ and a handle $Y$ below $\chi$ then, because we are looking at tree structures, we can see that a path from $X$ to $Y$ must go through $\chi$. Thus $\mathtt{join}$ 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 $\mathtt{split}$ and $\mathtt{join}$ in the rest of the proof.

In demonstrating soundness for definitions we need to compare the effects of $F$ and $D$ rather than $A$ and $M$. 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 $\mathtt{local.env}$ and $\mathtt{local.store}$ in $Q$. Note that the role of $\mathtt{rename}$ 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 $P'_1 < (\mathtt{local.store}, \ldots), \mathtt{local.env})$. Now the structural induction hypothesis applied to $s$ gives us that $P'_2 < ((\mathtt{local.store}', \mathtt{heap'}, \ldots), \mathtt{local.env})$. Finally the effect of returning $\mathtt{store}$ 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 $\mathtt{rename}$, $\mathtt{split}$ 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
4.4 Soundness Theorem

in the environment, to the standard function $Q$. The proof relates $D$ to $Q$ by fixed-point induction. Since safety is $\omega$-inclusive this amounts to relating $D^{(k)}$ to $Q^{(k)}$, where each of these denote the $k$th approximant in the unwinding of the recursive definitions.

We begin with a discussion of the unwinding 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 $\text{fix}(R)$. Using the fixed-point rule on $\mathcal{R}(\text{env})$ gives

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

and carrying out the indicated $\beta$-reductions gives

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

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

\[
\begin{align*}
D[r (x_1:t ; \ldots ; x_n:t ) l_1:t ; \ldots ; l_m:t ; s ] & \text{ env } = \text{fix}(R) \text{ env} \\
\text{where} & \\
R & = \lambda T. \lambda e. e[r \mapsto Q] \\
Q & = \lambda v_1. \lambda v_2. \ldots. \lambda v_n. \lambda (\text{store,heap,sp,fl,in,out}) \\
\text{let} & \\
\text{local.env} & = \\
(T e)[x_1\mapsto sp+1, \ldots , x_n\mapsto sp+n, \\
& l_1\mapsto sp+n+1, \ldots , l_m\mapsto sp+n+m] \\
\text{local.store} & = \\
\text{store[local.env(x_1)\mapsto v_1, \ldots , local.env(x_n)\mapsto v_n, \\
& local.env(l_1)\mapsto \text{undef}, \ldots , local.env(l_m)\mapsto \text{undef}] \\
\text{new.mem} & = \\
\mathcal{M}[s ] \text{ local.env (local.store, heap, sp+n+m, fl, in, out) } \\
\text{in} & \\
\text{if new.mem } = \text{error then} & \text{error} \\
\text{else} & \\
\text{let} & \\
& (\text{local.store',heap',sp',fl',in',out'}) = \text{new.mem} \\
\text{in} & \\
& (\text{store,heap',sp',fl',in',out'})
\end{align*}
\]

Figure 4.16: Recursive procedures in the Standard Semantics.

Similarly, we can define the unwinding of the functions in the abstract semantics. We define a sequence of $F^{(k)}$s and $D^{(k)}$s as in figure 4.18.
\[ R^{k+1}(env) = env[r \mapsto Q^{(k)}] \]

\[ Q^{(k)} = \lambda v_1 . \lambda v_2 . . . . . \lambda v_n . \lambda (store, heap, sp, fl, in, out) \]

\textbf{let}

\texttt{local_env =}

\[ (R^{(k)} env)[ x_1 \mapsto sp+1, \ldots, x_n \mapsto sp+n, \]

\[ l_1 \mapsto sp+n+1, \ldots, l_m \mapsto sp+n+m ] \]

\texttt{local_store =}

\[ store[local_env(x_1) \mapsto v_1, \ldots, local_env(x_n) \mapsto v_n, \]

\[ local_env(l_1) \mapsto \text{undef}, \ldots, local_env(l_m) \mapsto \text{undef} ] \]

\texttt{new_mem =}

\[ M[ s ] \text{ local_env (local_store, heap, sp+n+m, fl, in, out) } \]

\textbf{in}

\textbf{if new_mem = error then error else}

\textbf{let}

\[ (local\_store', heap', sp', fl', in', out') = new\_mem \]

\textbf{in}

\[ (store, heap', sp, fl', in', out') \]

---

Figure 4.17: The k'th unwinding of \( R \) and \( Q \).

The base 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 unw windings 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 \( \Omega \) 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 \( \Omega \) so it will not appear in any estimate, it merely serves as a convenient way to denote the partial unw windings.
\[ D^{(k)} = \lambda h_1 \ldots \ldots \lambda h_n . \lambda e . \lambda P . \]

\begin{verbatim}
let
  (P_var, P_inv) = split P [h_1, ..., h_n]
  P_1 = rename([x_1 \mapsto x_1, ..., x_n \mapsto x_n], P_var)
  P_2 = if e(r.new_est) then
         F^{(k)} e[r.est \mapsto (P_1, base(P_1))] P_1
      else F^{(k)} e P_1
  P_3 = rename([x_1 \mapsto h_1, ..., x_n \mapsto h_n], P_2)

in
  join(P_inv, P_3) [h_1, ..., h_n]
\end{verbatim}

\[ F^{(k+1)} = \lambda e . \lambda P . \]

\begin{verbatim}
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 x_1, ..., x_n \mapsto x_n,
                     l_1 \mapsto \text{nil}, ..., l_m \mapsto \text{nil}], P)
      P_2 = A[s] e[r.new_est \mapsto \text{false}, r.D \mapsto D^{(k)}] P_1
      NP_out = remove([x_1, ..., x_n, l_1, ..., l_m], P_2)
    in
    if NP_out \subseteq P_out then
      P_out { ----- 1(a) ----- }
    else
      F^{(k)} e[r.est \mapsto (P, (NP_out \bowtie P_out))] P { ----- 1(b) ----- }
  else
    { ----- CASE 2 (generalize input?) ----- }
    if P \subseteq P_in then
      P_out { ----- 2(a) ----- }
    else
      let
        NP_in = (P \bowtie P_in)
      in
      F^{(k)} e[r.new_est \mapsto \text{true}, r.est \mapsto (NP_in, base(NP_in))] NP_in
\end{verbatim}

Figure 4.18: The k'th unwinding of F.
\[ D^{(1)} = \lambda h_1 \ldots \lambda h_n \cdot \lambda e \cdot \lambda P . \]

let

\((P_{\text{var}}, P_{\text{inv}}) = \text{split } P \ [h_1, \ldots, h_n]\)

\[ P_1 = \text{rename}([h_1 \mapsto x_1, \ldots, h_n \mapsto x_n], P_{\text{var}}) \]

\[ P_2 = \text{if } e(r.\text{new}._\text{est}) \text{ then } \]

\[ F^{(1)} e[r.\text{est} \mapsto (P_1, \text{base}(P_1))] P_1 \]

\text{else } F^{(1)} e P_1

\[ P_3 = \text{rename}([x_1 \mapsto h_1, \ldots, x_n \mapsto h_n], P_2) \]

\text{in}

\[ \text{join}(P_{\text{inv}}, P_3) [h_1, \ldots, h_n] \]

\[ F^{(1)} = \lambda e \cdot \lambda P . \]

let\ ((P_{\text{in}}, P_{\text{out}}) = e(r.\text{est}) \text{ in})

\text{if } e(r.\text{new}._\text{est}) \text{ then }

\{ ----- CASE 1 (generalize output?) ----- \}

let

\[ P_1 = \text{insert}([x_1 \mapsto x_1, \ldots, x_n \mapsto x_n, \]

\[ l_1 \mapsto \text{nil}, \ldots, l_m \mapsto \text{nil}], P) \]

\[ P_2 = A[[s]] e[r.\text{new}._\text{est} \mapsto \text{false}, r.D \mapsto D^{(0)}] P_1 \]

\[ N_{P_{\text{out}}} = \text{remove}([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2) \]

\text{in}

\text{if } N_{P_{\text{out}}} \subseteq P_{\text{out}} \text{ then }

\[ P_{\text{out}} \ {--- 1(a) ---} \]

\text{else}

\[ \Omega \ {--- 1(b) ---} \]

\text{else}

\{ ----- CASE 2 (generalize input?) ----- \}

\text{if } P \subseteq P_{\text{in}} \text{ then }

\[ P_{\text{out}} \ {--- 2(a) ---} \]

\text{else}

\[ \Omega \ {--- 2(b) ---} \]

Figure 4.19: The first unwinding of \( F \), \( F^{(1)} \).
of the recursive definition. The termination of $A$ 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 $\omega$, a path matrix transformer that maps any path matrix to $\Omega$. 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 $\omega$ else $\omega = \omega$,

4. while $b$ do $\omega = skip$.

5. repeat $\omega$ until $e = \omega$.

Note that the while loop yields $skip$, since the body may never be executed, whereas the repeat loop yields $\omega$.

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 $\text{aenv}$ 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-$\Omega$ 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 $\Omega$, results from applying $A[s \ominus r]$ 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 $\lambda$-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 $\Omega$, 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 $F_{n-1}^{p-1,m}$ and the second by $F_{n-1}^{m-1}$. In the computation of $P_{2}$ we use an environment that has $r.D$ bound to $D_{n-1}^{m-1}$. We define $D_{n}^{m}$ analogously to $D$ using $F_{n}^{m-1,m}$ 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 unwinds of the recursive definition of $H$ are typically denoted by $H^{(n)}$ as we have been doing with $Q$ and $D$. These partial unwinds 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 $H^{(n-1)}$, $H^{(n-2)}$, etc., and 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 $P \preceq (M, env)$. Suppose that $aenv$ is an abstract environment such that $aenv \preceq env$ and that $aenv$ has $\Omega$ as the input estimate for calculating the effects of $r$. Consider the call tree of the expression $F_{0}^{i,m}aenvP$ and suppose that it contains a term of the form $F_{0}^{i,m}aenvP'$. Let the input estimate stored in $aenv'$ be $P_{1}$. Now consider the corresponding call tree of $R^{(n)}env$ arising as the $n$th approximant to $M[r(a_{1}, \ldots, a_{n})]envM$. Consider any memory $M'$ that occurs in the call tree in a term of the form $Q^{(k)}v_{1} \ldots v_{n}M'$ where $k \leq n$. Let $env'$ be the environment at this point in the call tree. For all such $(M', env')$ we have $P_{1} \preceq (M', env')$.

**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 $F_{n-1}^{m-1,m}$ 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 $P \preceq P_{n}$ we have incorporated the effect of one more level of nesting of recursive calls to $r$ than is captured by $P_{n}$. The inductive hypothesis states that when the term $F_{0}^{i,m}$ appears in the call tree generated by $F_{n-1}^{m-1}$, the abstract environment has an input estimator, in this case $P_{n}$, that incorporates the effect of $n - 1$ levels of
\[ D_n^m = \lambda h_1 \ldots \ldots \lambda h_n \cdot \lambda e \cdot \lambda P. \]

let
\[ (P_{var}, P_{inv}) = \text{split } P \left[ h_1, \ldots, h_n \right] \]
\[ P_1 = \text{rename}([h_1 \mapsto x_1, \ldots, h_n \mapsto x_n], P_{var}) \]
\[ P_2 = \begin{cases} \text{if } e(r.new.est) \text{ then} & F_n^{m,m} e[r.est \mapsto (P_1, \text{base}(P_1))] P_1 \\ \text{else } F_n^{m,m} \text{ e } P_1 \end{cases} \]
\[ P_3 = \text{rename}([x_1 \mapsto h_1, \ldots, x_n \mapsto h_n], P_2) \]

in
\[ \text{join}(P_{inv}, P_3) \left[ h_1, \ldots, h_n \right] \]
\[ F_n^p = \lambda e \cdot \lambda P. \]

let \( P_{in}, P_{out} = e(r.est) \) in
\[ \begin{cases} \text{if } e(r.new.est) \text{ then} & \{ \text{----- CASE 1 (generalize output?) ----- } \} \\
\text{let} & P_1 = \text{insert}([x_1 \mapsto x_1, \ldots, x_n \mapsto x_n, \\
& l_1 \mapsto \text{nil}, \ldots, l_m \mapsto \text{nil}], P) \\
& P_2 = A[s] e[r.new.est \mapsto \text{false}, r.D \mapsto D_n^{m-1}] P_1 \\
& NP_{out} = \text{remove}([x_1, \ldots, x_n, l_1, \ldots, l_m], P_2) \text{ in} \\
& \text{if } NP_{out} \subseteq P_{out} \text{ then} & P_{out} \{ \text{-- 1(a) -- } \} \\
& \text{else} & F_n^{p-1,m} e[r.est \mapsto (P, (NP_{out} \bowtie P_{out}))] P \{ \text{-- 1(b) -- } \} \end{cases} \]

else
\[ \begin{cases} \text{----- CASE 2 (generalize input?) ----- } \} \\
\text{if } P \subseteq P_{in} \text{ then} & P_{out} \{ \text{-- 2(a) -- } \} \end{cases} \]

else
\[ \begin{cases} \text{let} & NP_{in} = (P \bowtie P_{in}) \\
in \{ \text{-- 2(b) -- } \} & F_n^{m-1} e[r.new.est \mapsto \text{true}, r.est \mapsto (NP_{in}, \text{base}(NP_{in}))] NP_{in} \end{cases} \]

Figure 4.20: \( F_n^p \) approximants.
nesting of recursive calls of $r$. Thus, $F^{i,m}_n$ 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, $F^{p,m}_n$, 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 $F^{p,m}_n$ 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 $P \subseteq P_{in}$ 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 $(\Omega, \Omega)$ as the estimator an initialized environment. The key fact that we need to prove is the following.

**Lemma 6.** Suppose that $P \triangleleft (M, env)$. Suppose that $aenv \triangleleft env$ and that $aenv$ is an initialized environment. Suppose $n$ is chosen large enough that $F^{p,m}_n$ 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. $F^{p,m}_n aenv P \triangleleft (Q^{(m)} v_1 \ldots v_l M, env)$.

2. Any leaf in the call tree of $F^{p,m}_n aenv P$ 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 $F^{p,m}_n aenv P$. 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 insufficieently 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 $NP_{out} \subseteq P_{out}$, $P_{out}$ 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 have 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 $F_{n_2}^{m_2,m_2} \subseteq F_{n_3}^{m_3,m_3}$. 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 $F^{(k)} \subseteq F_{n}^{m,m}$. 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 $\omega$-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.

![Diagram](image)

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 \((\text{name}, \text{kind})\), where \text{name} is the name of a variable and \text{kind} is
5.1 Interference between Basic Statements

either the special designation \texttt{var} (variable), or a field name. For binary trees the possible kinds are: \texttt{var} - a variable, \texttt{left} - the left field of a node, \texttt{right} - the right field of a node, or \texttt{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 \textit{definite aliases}, while the other three possibilities indicate that locations \((x, f)\) and \((a, f)\) are \textit{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.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Read Set ( \mathcal{R}(s, p) )</th>
<th>Write Set ( \mathcal{W}(s, p) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{a := nil}</td>
<td>{}</td>
<td>{\texttt{(a, var)}}</td>
</tr>
<tr>
<td>\texttt{a := new()}</td>
<td>{}</td>
<td>{\texttt{(a, var)}}</td>
</tr>
<tr>
<td>\texttt{a := b}</td>
<td>{\texttt{(b, var)}}</td>
<td>{\texttt{(a, var)}}</td>
</tr>
<tr>
<td>\texttt{a := b.f}</td>
<td>{\texttt{(b, var)}} \cup \mathcal{A}(b, f, p)</td>
<td>{\texttt{(a, var)}}</td>
</tr>
<tr>
<td>\texttt{a.f := b}</td>
<td>{\texttt{(a, var)}, \texttt{(b, var)}}</td>
<td>\mathcal{A}(a, f, p)</td>
</tr>
</tbody>
</table>

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

The \textit{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.

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

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

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

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

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

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

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

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 \( p[x_i, y_j] = p[y_j, x_i] = \emptyset \). 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 \( l\text{-side} \) and \( r\text{-side} \) are not related \( (p_A[l\text{-side}, r\text{-side}] = p_A[r\text{-side}, l\text{-side}] = \emptyset) \). Therefore, the procedure calls \( add_n(l\text{-side}, 1) \) and \( add_n(r\text{-side}, -1) \) 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-
5.2 Interference between Procedure Calls

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 – pa }
    add_n(l_side,i);
    add_n(r_side,-i);
    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 – pb }
            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 – pc }
            reverse(l);
            reverse(r);
            h.left := r;
            h.right := l
        end
    end;

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 \( \mathcal{L} \). Therefore, we will refer to locations by their access path from handles in \( \mathcal{L} \). A relative location is a triple \( (\text{name}, \text{field.type}, \text{access.path}) \) where \text{name} is the name of a handle in \( \mathcal{L} \), \text{field.type} is one of \text{var}, \text{left}, \text{right}, or \text{value}, and \text{access.path} is a set of path expressions describing the path from \text{name} to the node which is being read or updated.

The relative read and write functions \( \mathcal{W}_r^r(s, p, \mathcal{L}) \) and \( \mathcal{R}_r^r(s, p, \mathcal{L}) \) are defined as in figure 5.7. In these rules we use the relative alias function \( \mathcal{A}_r^r \). Given a handle \( h \), a field name \( f \), a set of live handles \( \mathcal{L} \), and a path matrix \( p \), \( \mathcal{A}_r^r(h, f, \mathcal{L}, p) \) returns the set of relative locations which are possibly aliased to the location referred to by \( h.f \). \( \mathcal{A}_r^r(h, f, \mathcal{L}, p) \) 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 \( \mathcal{R}_n^r([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L}) \) to be the relative read set for statement sequence \([s_1, \ldots, s_n]\) and \( \mathcal{W}_n^r([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L}) \) to be the relative write set for statement sequence \([s_1, \ldots, s_n]\).

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

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

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

\[\text{\footnote{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 $V$.

<table>
<thead>
<tr>
<th>Statement</th>
<th>$\mathcal{R}^r(s, p, \mathcal{L})$</th>
<th>$\mathcal{W}^r(s, p, \mathcal{L})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a := \text{nil}$</td>
<td>${}$</td>
<td>${(a, \text{var}, S)}$</td>
</tr>
<tr>
<td>$a := \text{new()}$</td>
<td>${}$</td>
<td>${(a, \text{var}, S)}$</td>
</tr>
<tr>
<td>$a := b$</td>
<td>${(b, \text{var}, S)}$</td>
<td>${(a, \text{var}, S)}$</td>
</tr>
<tr>
<td>$a := b.f$</td>
<td>${(b, \text{var}, S)} \cup \mathcal{A}^r(b, f, \mathcal{L}, p)$</td>
<td>${(a, \text{var}, S)}$</td>
</tr>
<tr>
<td>$a.f := b$</td>
<td>${(a, \text{var}, S), (b, \text{var}, S)}$</td>
<td>$\mathcal{A}^r(a, f, \mathcal{L}, p)$</td>
</tr>
</tbody>
</table>

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, $I'(U, P, V, Q, L)$, is defined as follows.

$$I'(U, P, V, Q, L) =$$

$$\left[ W'(U, P, L) \cap \left( R'(V, Q, L) \cup W'(V, Q, L) \right) \right] \cup$$

$$\left[ W'(V, Q, L) \cap \left( R'(U, P, L) \cup W'(U, P, L) \right) \right]$$

If the data structure is a TREE at the program point just before $U \parallel V$, then $U$ and $V$ do not interfere if $I'(U, P, V, Q, L) = \emptyset$. 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 $L$. 

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\textsuperscript{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.

\textsuperscript{1}Butterfly is a trademark of Bolt Beranek and Newman Inc.
Figure 6.1: Current System

<table>
<thead>
<tr>
<th></th>
<th>Global (microseconds)</th>
<th>Local (microseconds)</th>
<th>Global/Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>8.12</td>
<td>1.22</td>
<td>6.7</td>
</tr>
<tr>
<td>Write</td>
<td>3.52</td>
<td>.81</td>
<td>4.3</td>
</tr>
</tbody>
</table>

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 $\ast$). We refer to the recursive unfolding in reverse as $(\ast, \ast)$-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 $\text{foo}(h) || \text{process\_next}(t)$ as $(1, \ast)$-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 $(\ast, \ast)$-unfoldings we use an interval-based processor allocation scheme which is similar to the team-split mechanism in PCP [Bro88]. For the $(1, \ast)$-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 $(1, \ast)$-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 ≠ 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 ≠ 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.
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 Bmerge. The Bisort procedure (figure 6.5) has three arguments: a TREE with \( n-1 \) nodes (\( \text{root} \)), a TREE with 1 node (\( \text{sp}.r \)), and an integer direction (\( \text{dir} \)). Bisort recursively sorts two smaller problems, and then merges the result using Bmerge. The procedure Bmerge (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 Bmerge 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 Bmerge can be safely executed in parallel, and that the recursive calls in Bisort can be safely executed in parallel. Consider the recursive calls Bmerge(\( rl, sp.l, dir \)) and Bmerge(\( rr, sp.r, dir \)) in the body of the procedure Bmerge. 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 \( rl, rr, sp.l \text{ and } sp.r \) 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 Bmerge 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,l,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.
### Generating Parallel Programs

<table>
<thead>
<tr>
<th></th>
<th>root*9</th>
<th>sp_r*9</th>
<th>root</th>
<th>sp_r</th>
<th>sp_l</th>
<th>rl</th>
<th>rr</th>
</tr>
</thead>
<tbody>
<tr>
<td>root*9</td>
<td>S</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td>L1</td>
<td>R1</td>
</tr>
<tr>
<td>sp_r*9</td>
<td></td>
<td>S</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>root</td>
<td></td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td>L1</td>
<td>R1</td>
</tr>
<tr>
<td>sp_r</td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sp_l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>rr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

Figure 6.7: Path Matrix before recursive calls in Bmerge.

*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 *Bmerge* 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 than one processor, the interval is divided between a parallel call to *Bisort* (1, 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 are 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 (execute-
6.2 An example: Adaptive Bitonic Sort

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);
}
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.
6.3 Further Issues

\[ \text{Speedup}(S) \text{ vs. Number of Processors } (P) \]
\[ (o) \text{ } n=128, \text{ } (\times) \text{ } n=512, \text{ } \text{and } (\bullet) \text{ } n=8192 \]

Figure 6.10: Speedup for 1 \ldots 32 processors.
$\text{Speedup}(S) \text{ vs. Number of Processors (P)}$

\((\circ) n=128, (\star) n=512, \text{ and } (\bullet) n=8192\)

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

---

*Speedup*$(S)$ vs. *Number of Processors*$(P)$

- nodes allocated on all 32 processors
- nodes allocated only on $P$ processors

$n=8192$

---

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 \textit{TREE} or \textit{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 \textit{TREES} and \textit{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 in-
volves 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
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 Semantics for SIL

A.1 Semantic Domains

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

where

\[ \text{Env} = \text{Id} \rightarrow \text{Loc} + \text{Procedure} \]
\[ \text{Store} = \text{Loc} \rightarrow \text{Value} \]
\[ \text{Heap} = \text{NodePtr} \rightarrow ((\text{Int} + \text{undef}) \times (\text{NodePtr} + \text{nil}) \times (\text{NodePtr} + \text{nil})) \]
\[ \text{Value} = \text{Int} + \text{NodePtr} + \text{nil} + \text{undef} \]
\[ \text{Procedure} = \text{Value}^n \rightarrow (\text{Mem} \rightarrow \overline{\text{Mem}}) \]
\[ \text{File} = \text{Int list} \]
\[ \text{NodePtr} = \text{Int} \]
\[ \text{Loc} = \text{Int} \]
\[ \text{Mem} = (\text{Store} \times \text{Heap} \times \text{Loc} \times \text{NodePtr} \times \text{File} \times \text{File}) + \text{error} \]

A.2 Expressions

\[ \mathcal{E}[\text{constant}] \text{ env mem} = \text{constant} \]

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

\[ \mathcal{E}[X.f] \text{ env (store, heap, sp, fl, in, out)} = \]
\[ \hspace{1em} \text{let} \]
\[ \hspace{2em} h = \text{store(env}(X)) \]
\[ \hspace{2em} \text{in} \]
\[ \hspace{3em} \text{heap}(h).f \]
\[ \mathcal{E}[\ e_1 \ op \ e_2 \ ] \ env \ mem = \]
\[
\begin{align*}
\text{let} & \quad v_1 = (\ \mathcal{E}[\ e_1 \ ] \ env \ mem) \\
\text{let} & \quad v_2 = (\ \mathcal{E}[\ e_2 \ ] \ env \ mem) \\
\text{in} & \quad \begin{cases} 
\text{if } v_1 = \text{undef or } v_2 = \text{undef then} & \text{undef} \\
\text{else} & v_1 \ op \ v_2 
\end{cases}
\end{align*}
\]

### A.3 Scalar Statements

\[ \mathcal{M}[\ s \ ] \ env \ \text{error} = \text{error} \]

\[ \mathcal{M}[\ X := e \ ] \ env \ (\text{store, heap, sp, fl, in, out}) = \]
\[
\begin{align*}
\text{let} & \quad v = \mathcal{E}[\ e] \ env \ (\text{store, heap, sp, fl, in, out}) \\
\text{in} & \quad (\text{store}[\text{env}(X) \mapsto v], \text{heap}, \text{sp}, \text{fl}, \text{in}, \text{out})
\end{align*}
\]

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

\[ \mathcal{M}[\ \text{put}(e) \ ] \ env \ (\text{store, heap, sp, fl, in, out}) = \]
\[
\begin{align*}
\text{let} & \quad v = \mathcal{E}[\ e] \ env \ mem \\
\text{in} & \quad (\text{store, heap, sp, fl, in, out}::v)
\end{align*}
\]

### A.4 Handle Statements

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

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

\[ \mathcal{M}[\ A := B \ ] \ env \ (\text{store, heap, sp, fl, in, out}) = \]
\[
(\text{store}[\text{env}(A) \mapsto \text{store}(\text{env}(B))], \text{heap}, \text{sp, fl, in, out})
\]
\[ M[ A := B.f ] env (store, heap, sp, fl, in, out) = \]
\[
\begin{align*}
& \textbf{let} \\
& \quad h = \text{store}(env(B)) \\
& \textbf{in} \\
& \quad \textbf{if } h = \textit{nil} \textbf{ then error else } (\text{store}[env(A) \mapsto h.f], heap, sp, fl, in, out)
\end{align*}
\]
\[
M[ A.f := e ] env (store, heap, sp, fl, in, out) = \\
\begin{align*}
& \textbf{let} \\
& \quad h = \text{store}(env(A)) \\
& \quad v = E[ exp ] env \text{ mem} \\
& \textbf{in} \\
& \quad \textbf{if } h = \textit{nil} \textbf{ or } v = \textit{error then} \\
& \quad \quad \textit{error} \\
& \quad \textbf{else} \\
& \quad \quad (\text{store}, \text{heap}[h.f \mapsto v], sp, fl, in, out)
\end{align*}
\]

A.5 Compound Statements

\[ M[ \text{begin end } ] env \text{ mem } = \text{mem} \]

\[
M[ \text{begin } s_1 ; s_2 ; \ldots ; s_n \text{ end } ] env \text{ mem } = \\
M[ \text{begin } s_2 ; \ldots ; s_n \text{ end } ] env \left( M[ s_1 ] env \text{ mem } \right)
\]

\[
M[ \text{if } e \text{ then } s_1 \text{ else } s_2 ] env \text{ mem } = \\
\begin{align*}
& \textbf{if } E[ e ] env \text{ mem then} \\
& \quad M[ s_1 ] env \text{ mem} \\
& \textbf{else} \\
& \quad M[ s_2 ] env \text{ mem}
\end{align*}
\]

\[
M[ \text{while } e \text{ do } s ] env \text{ mem } = \text{fix}(W) \text{ mem} \\
\textbf{where}
\begin{align*}
W &= \lambda T. \lambda m. \\
& \quad \textbf{if } E[ e ] env \text{ m then } T(M[ s ] env m) \text{ else } m
\end{align*}
\]

\[
M[ \text{repeat } s \text{ until } e ] env \text{ mem } = \text{fix}(W) \text{ mem} \\
\textbf{where}
\begin{align*}
W &= \lambda T. \lambda m. \\
& \quad \textbf{let } m_2 = M[ s ] env m \textbf{ in} \\
& \quad \textbf{if } E[ e ] env m_2 \textbf{ then } m_2 \textbf{ else } T(M[ s ] env m_2)
\end{align*}
\]
A.6 Procedures

\[ \mathcal{M}[ p( a_1, a_2, \ldots, a_n ) ] \text{ env mem} = \]
\[ \text{let} \]
\[ v_1 = \mathcal{E}[ a_1 ] \text{ env mem} \]
\[ \ldots \]
\[ v_n = \mathcal{E}[ a_n ] \text{ env mem} \]
\[ \text{in} \]
\[ \text{env}(p)(v_1 \ldots v_n) \text{ mem} \]
\[ \mathcal{D}[ \text{proc } q( x_1:t; \ldots; x_n:t ) \ l_1:t \ldots l_m:t; \ s ] \text{ env} = \text{env}[q \mapsto Q] \]
where \( Q = \lambda v_1. \lambda v_2. \ldots. \lambda v_n. \lambda (\text{store, heap, sp, fl, in, out}) \).
\[ \text{let} \]
\[ \text{local.env} = \]
\[ \text{env}[ x_1 \mapsto \text{sp}+1, \ldots, x_n \mapsto \text{sp}+n, \]
\[ l_1 \mapsto \text{sp}+n+1, \ldots, l_m \mapsto \text{sp}+n+m ] \]
\[ \text{local.store} = \]
\[ \text{store}[\text{local.env}(x_1) \mapsto v_1, \ldots, \text{local.env}(x_n) \mapsto v_n, \]
\[ \text{local.env}(l_1) \mapsto \text{undef}, \ldots, \text{local.env}(l_m) \mapsto \text{undef} ] \]
\[ \text{new.mem} = \mathcal{M}[ s ] \text{ local.env (local.store, heap, sp+n+m, fl, in, out) } \]
\[ \text{in} \]
\[ \text{if new.mem = error then} \]
\[ \text{error} \]
\[ \text{else} \]
\[ \text{let} \]
\[ (\text{local.store}', \text{heap}', \text{sp}', \text{fl}', \text{in}', \text{out}') = \text{new.mem} \]
\[ \text{in} \]
\[ (\text{store, heap', sp', fl', in', out'}) \]
\[ \mathcal{D}[ \text{recproc } r( x_1:t; \ldots; x_n:t ) \ l_1:t \ldots l_m:t; \ s ] \text{ env} = \text{fix}(R) \text{ env} \]
where \( R = \lambda T. \lambda e. e[r \mapsto Q] \)
\[ Q = \lambda v_1. \lambda v_2. \ldots. \lambda v_n. \lambda (\text{store, heap, sp, fl, in, out}) \).
\[ \text{let} \]
\[ \text{local.env} = \]
\[ (T e)[ x_1 \mapsto \text{sp}+1, \ldots, x_n \mapsto \text{sp}+n, \]
\[ l_1 \mapsto \text{sp}+n+1, \ldots, l_m \mapsto \text{sp}+n+m ] \]
\[ \text{local.store} = \]
\[ \text{store}[\text{local.env}(x_1) \mapsto v_1, \ldots, \text{local.env}(x_n) \mapsto v_n, \]
\[ \text{local.env}(l_1) \mapsto \text{undef}, \ldots, \text{local.env}(l_m) \mapsto \text{undef} ] \]
\[ \text{new.mem} = \]
\[ \mathcal{M}[ s ] \text{ local.env (local.store, heap, sp+n+m, fl, in, out) } \]
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')

D[[ d_1; d_2; \ldots; d_n ]] env =
D[[ d_2; \ldots; d_n ]] newenv
where newenv = D[[ d_1 ]] env

A.7 Programs

P[[ program id def_list ]] in_file =
let
  final_mem =
  M[[ main() ]]
  ( 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 s_1;
  ...
  int s_n;
  struct node *h_1;
  ...
  struct node *h_m;
};

typedef struct node HANDLE;

#define NIL ((HANDLE *) 0)
```
B.2 Statements

<table>
<thead>
<tr>
<th>$a := b$</th>
<th>$a = b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a := b.f$</td>
<td>$a = b-&gt;f$</td>
</tr>
<tr>
<td>$a.f := b$</td>
<td>$a-&gt;f = b$</td>
</tr>
<tr>
<td>begin</td>
<td>{ s1; s2; ... sn; }</td>
</tr>
<tr>
<td>$s_1; s_2; \ldots s_n$ end</td>
<td></td>
</tr>
<tr>
<td>if $exp$ then</td>
<td>if (exp)</td>
</tr>
<tr>
<td>$s_1$</td>
<td>s1</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>$s_2$</td>
<td>s2</td>
</tr>
<tr>
<td>while $exp$ do</td>
<td>while (exp)</td>
</tr>
<tr>
<td>$s_1$</td>
<td>s1</td>
</tr>
<tr>
<td>repeat</td>
<td>do</td>
</tr>
<tr>
<td>$s_1$</td>
<td>s1</td>
</tr>
<tr>
<td>until $exp$</td>
<td>while (! exp)</td>
</tr>
</tbody>
</table>

B.3 Procedures and Functions

The following rules give the translations for procedures, functions returning one value, and functions return more than 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.

<table>
<thead>
<tr>
<th>$p(x_1,x_2,\ldots,x_n)$</th>
<th>$p(x_1,x_2,\ldots,x_n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1 := f(x_1,x_2,\ldots,x_n)$</td>
<td>$r_1 = f(x_1,x_2,\ldots,x_n)$</td>
</tr>
<tr>
<td>$r_1, \ldots, r_m := f(x_1,x_2,\ldots,x_n)$</td>
<td>$f(x_1,x_2,\ldots,x_n,&amp;r_1,\ldots,&amp;r_m)$</td>
</tr>
<tr>
<td>Procedure</td>
<td>Void Function</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
</tr>
</tbody>
</table>
| \texttt{procedure } p(x_1 : t_1, \ldots, x_n : t_n) \begin{align*} l_1 & : lt_1; \\ \vdots & \\ l_m & : lt_m \end{align*} \texttt{begin} s_1; \\vdots \\texttt{end} & \texttt{void p(x_1, x_2, \ldots, x_n)} \\
| & \begin{align*} t_1 & : x_1; \\vdots \\texttt{tn : x_n} \\ \{ & \texttt{lt1 li;} \\ \vdots & \\ \	exttt{ltm lm;} \\ s_1 & ; \\ \vdots & \\ \	exttt{st;} \end{align*} \texttt{)} \\
| Function | Void Function |
| \texttt{function } f(x_1 : t_1, \ldots, x_n : t_n) \texttt{rt} \begin{align*} l_1 & : lt_1; \\ \vdots & \\ l_m & : lt_m \end{align*} \texttt{begin} s_1; \\vdots \\texttt{st} \\texttt{end } \Rightarrow \texttt{return(r)} & \texttt{rt f(x_1, x_2, \ldots, x_n)} \\
| & \begin{align*} t_1 & : x_1; \\vdots \\text{tn : x_n} \\ \{ & \text{lt1 li;} \\ \vdots & \\ \	exttt{ltm lm;} \\ s_1 & ; \\ \vdots & \\ \	exttt{st;} \texttt{)} \\
| Function | Void Function |
| \texttt{function } f(x_1 : t_1, \ldots, x_n : t_n) \texttt{rt}_1, \ldots, \texttt{rt}_q \begin{align*} l_1 & : lt_1; \\ \vdots & \\ l_m & : lt_m \end{align*} \texttt{begin} s_1; \\vdots \\texttt{st} \\texttt{end } \Rightarrow \texttt{return(r}_1, \ldots, r_q) & \texttt{void f(x_1, x_2, \ldots, x_n,} \\
| & \texttt{r}_1, \ldots, \texttt{r}_q) \\
| & \begin{align*} t_1 & : x_1; \\vdots \\text{tn : x_n} \\ \texttt{tr}_1 \ast \texttt{r}_1; \\vdots \\texttt{tr}_q \ast \texttt{r}_q; \\ \{ & \texttt{lt1 : li;} \\ \vdots & \\ \	exttt{ltm : lm;} \\ s_1 & ; \\ \vdots & \\ \	exttt{st;} \texttt{)} \\
| & \begin{align*} \ast \texttt{r}_1 & = \texttt{ri;} \\ \vdots & \\ \ast \texttt{r}_q & = \texttt{rq;} \end{align*} \texttt{)}
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.

\begin{verbatim}
program quicksort
constant
  RANDOMDIV = 100000
end;

node
  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
\end{verbatim}

\(^1\)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,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
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)
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 `(O, O)` 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 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.

```plaintext
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)  

<table>
<thead>
<tr>
<th>root(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
</tr>
</tbody>
</table>

last_node := root  

<table>
<thead>
<tr>
<th>root(*)</th>
<th>last_node(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

while (n>1) do ....  

============ BEGIN WHILE/REPEAT ===========  

begin ... end  
next_val:=random()  
next_val := (next_val/100000)  
next_node:=new_node(next_val)  

<table>
<thead>
<tr>
<th>root(*)</th>
<th>last_node(*)</th>
<th>next_node(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

last_node.next := next_node  

<table>
<thead>
<tr>
<th>root(*)</th>
<th>next_node(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>N^1</td>
</tr>
</tbody>
</table>

last_node := next_node  

<table>
<thead>
<tr>
<th>root(*)</th>
<th>last_node(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>N^1</td>
</tr>
</tbody>
</table>

n := (n-1)  

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

IN:  

<table>
<thead>
<tr>
<th>root(*)</th>
<th>last_node(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

OUT:
### C.3 Analysis of build_list

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*$</td>
<td>$*$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$N^1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S$</td>
</tr>
</tbody>
</table>

---

ITERATION # 2

begin ... end
next_val:=random()
next_val := (next_val/100000)
next_node:=new_node(next_val)

| root | last_node | next_node |  
|------|-----------|
| $*$  | $*$       | $*$       |

| root | last_node | next_node |  
|------|-----------|
| $S$  | $S$       | $S$       |

last_node.next := next_node

<table>
<thead>
<tr>
<th>root</th>
<th>next_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*$</td>
<td>$*$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>root</th>
<th>next_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$N^2$</td>
</tr>
</tbody>
</table>

last_node := next_node

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*$</td>
<td>$*$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$N^2$</td>
</tr>
</tbody>
</table>

n := (n-1)

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

**IN:**

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*$</td>
<td>$*$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$N^1$</td>
</tr>
</tbody>
</table>

**OUT:**

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*$</td>
<td>$*$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$N^2$</td>
</tr>
</tbody>
</table>

IN merge OUT => next(IN):

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*$</td>
<td>$*$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>root</th>
<th>last_node</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$N^+$</td>
</tr>
</tbody>
</table>

---
ITERATION # 3

begin ... end
next_val := random()
next_val := (next_val/100000)
next_node := new_node(next_val)

<table>
<thead>
<tr>
<th></th>
<th>root(0,0)</th>
<th>last_node(0,0)</th>
<th>next_node(0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td>N^+</td>
<td></td>
</tr>
<tr>
<td>last_node</td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>next_node</td>
<td></td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

last_node.next := next_node

<table>
<thead>
<tr>
<th></th>
<th>root(0,0)</th>
<th>next_node(0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td>N^1N^+</td>
</tr>
<tr>
<td>next_node</td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

last_node := next_node

<table>
<thead>
<tr>
<th></th>
<th>root(0,0)</th>
<th>last_node(0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td>N^1N^+</td>
</tr>
<tr>
<td>last_node</td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

n := (n-1)

results of iteration #3

IN:

<table>
<thead>
<tr>
<th></th>
<th>root(0,0)</th>
<th>last_node(0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td>N^+</td>
</tr>
<tr>
<td>last_node</td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

OUT:

<table>
<thead>
<tr>
<th></th>
<th>root(0,0)</th>
<th>last_node(0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td>N^1N^+</td>
</tr>
<tr>
<td>last_node</td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

IN merge OUT => next(IN):

<table>
<thead>
<tr>
<th></th>
<th>root(0,0)</th>
<th>last_node(0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td></td>
<td>N^+</td>
</tr>
<tr>
<td>last_node</td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

END WHILE/REPEAT
C.4 Analysis of qsort

<table>
<thead>
<tr>
<th>root((\bullet, \odot))</th>
<th>last_node((\bullet, \odot))</th>
</tr>
</thead>
<tbody>
<tr>
<td>root (S)</td>
<td>((S + N^+))</td>
</tr>
<tr>
<td>last_node (S?)</td>
<td>(S)</td>
</tr>
</tbody>
</table>

================ END build_nonnil_list ===============

<table>
<thead>
<tr>
<th>list((\bullet, \odot))</th>
</tr>
</thead>
<tbody>
<tr>
<td>list (S)</td>
</tr>
</tbody>
</table>

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

list := nil

<table>
<thead>
<tr>
<th>list((\bullet, \odot))</th>
</tr>
</thead>
<tbody>
<tr>
<td>list (S)</td>
</tr>
</tbody>
</table>

================ END cond ===============

<table>
<thead>
<tr>
<th>list((\odot, \odot))</th>
</tr>
</thead>
<tbody>
<tr>
<td>list (S)</td>
</tr>
</tbody>
</table>

================ 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.

```plaintext
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, l.t, r.r, r.t: handle;
p.l, p.t: handle
begin
  if h = nil then
    begin
      q.h := nil;
      q.t := nil
    end
  else
```
unsorted list

\[ l, p, h, p_t, r := \text{partition}(h); \]

\begin{align*}
\text{sorted} < p & : & \text{l.h} & \text{l.t} \\
\text{sorted} = p & : & \text{p.h} & \text{p.t} \\
\text{sorted} > p & : & \text{r.h} & \text{r.t}
\end{align*}

\[ \text{l.h}, \text{l.t} := \text{qsort(l)}; \quad \text{r.h}, \text{r.t} := \text{qsort(r)}; \]

\[ q, h, q, t := \text{link.up(l.h, l.t, p.h, p.t)}; \]

\begin{align*}
\text{sorted} \leq p & : & \text{q.h} & \text{q.t} \\
\text{sorted} > p & : & \text{r.h} & \text{r.t}
\end{align*}

\[ q, h, q, t := \text{link.up(q.h, q.t, r.h, r.t)}; \]

sorted list

\[ \text{q.h} \quad \text{q.t} \]

Figure C.1: Diagrammatic outline for *qsort*. 
************* [ 1 ] qsort - LEVEL 0 *************

INPUT APPROX:

<table>
<thead>
<tr>
<th>h (\in) (\emptyset, \emptyset)</th>
<th>h((\emptyset, \emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>h (\in) (\emptyset)</td>
<td>S</td>
</tr>
<tr>
<td>h</td>
<td>S</td>
</tr>
</tbody>
</table>

OUTPUT APPROX:

<table>
<thead>
<tr>
<th>h (\in) (\emptyset, \emptyset)</th>
<th>q.h((\emptyset),(\emptyset))</th>
<th>q.t((\emptyset),(\emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>h (\in) (\emptyset)</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>q.h</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>q.t</td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

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

INPUT APPROX:

<table>
<thead>
<tr>
<th>h (\in) (\emptyset, \emptyset)</th>
<th>h((\emptyset, \emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>h (\in) (\emptyset)</td>
<td>S</td>
</tr>
<tr>
<td>h</td>
<td>S</td>
</tr>
</tbody>
</table>

OUTPUT APPROX:

<table>
<thead>
<tr>
<th>h (\in) (\emptyset, \emptyset)</th>
<th>q.h((\emptyset),(\emptyset))</th>
<th>q.t((\emptyset),(\emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>h (\in) (\emptyset)</td>
<td>S</td>
<td>(S + N^+)?</td>
</tr>
<tr>
<td>q.h</td>
<td>(S^?)</td>
<td>(S + N^+)</td>
</tr>
<tr>
<td>q.t</td>
<td>(S^?)</td>
<td>(S^?)</td>
</tr>
</tbody>
</table>

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

INPUT APPROX:

<table>
<thead>
<tr>
<th>h (\in) (\emptyset, \emptyset)</th>
<th>h((\emptyset, \emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>h (\in) (\emptyset)</td>
<td>S</td>
</tr>
<tr>
<td>h</td>
<td>S</td>
</tr>
</tbody>
</table>

OUTPUT APPROX:

<table>
<thead>
<tr>
<th>h (\in) (\emptyset, \emptyset)</th>
<th>q.h((\emptyset),(\emptyset))</th>
<th>q.t((\emptyset),(\emptyset))</th>
</tr>
</thead>
<tbody>
<tr>
<td>h (\in) (\emptyset)</td>
<td>S</td>
<td>(S + N^+)?</td>
</tr>
<tr>
<td>q.h</td>
<td>(S + N^+)?</td>
<td>(S + N^+)?</td>
</tr>
<tr>
<td>q.t</td>
<td>(S^?)</td>
<td>(S^?)</td>
</tr>
</tbody>
</table>

Figure C.2: Iterative Approximation for qsort.
begin
1.p.h.p.t.r := partition(h);
1.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 \textit{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 \textit{qsort} indicates that \textit{l} and \textit{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{center}
\begin{tabular}{|c|c|}
\hline
\textit{h} & \textit{q.h} \tabularnewline
\hline
6(0,0) & 6(0,0) \tabularnewline
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textit{h} & \textit{q.h} \tabularnewline
\hline
6 & S \tabularnewline
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\textit{h} & \textit{q.h} & \textit{q.t} \tabularnewline
\hline
6(0,0) & 6(0,0) & 6(0,0) \tabularnewline
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textit{h} & \textit{q.t} \tabularnewline
\hline
6 & S \tabularnewline
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\textit{h} & \textit{q.t} & \textit{p.t} \tabularnewline
\hline
6(0,0) & 6(0,0) & 6(0,0) \tabularnewline
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\textit{h} & \textit{l} & \textit{r} & \textit{p.h} & \textit{p.t} \tabularnewline
\hline
6 & S & S & S? & (S + N^+)? \tabularnewline
\hline
l & S & & & \tabularnewline
\hline
r & & S & & \tabularnewline
\hline
p.h & S? & & S & (S + N^+) \tabularnewline
\hline
p.t & S? & & S? & S \tabularnewline
\hline
\end{tabular}
\end{center}
### Analysis of qsort

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

<table>
<thead>
<tr>
<th></th>
<th>h*6</th>
<th>Lh</th>
<th>Lt</th>
<th>r</th>
<th>p.h</th>
<th>p.t</th>
</tr>
</thead>
<tbody>
<tr>
<td>h*6</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l_h</td>
<td>S</td>
<td></td>
<td>(S + N^+)?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lt</td>
<td>S?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p.h</td>
<td>S?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p.t</td>
<td>S?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>h*6</th>
<th>Lh</th>
<th>Lt</th>
<th>r_h</th>
<th>r_t</th>
<th>p.h</th>
<th>p.t</th>
</tr>
</thead>
<tbody>
<tr>
<td>h*6</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lh</td>
<td>S</td>
<td></td>
<td>(S + N^+)?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lt</td>
<td>S?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_h</td>
<td>S</td>
<td></td>
<td>(S + N^+)?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_t</td>
<td>S?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p.h</td>
<td>S?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p.t</td>
<td>S?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>h*6</th>
<th>q_h</th>
<th>q_t</th>
<th>r_h</th>
<th>r_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>h*6</td>
<td>S</td>
<td>S?</td>
<td>(S + N^+)?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q_h</td>
<td>S?,(S + N^+)?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q_t</td>
<td>S?</td>
<td>S?</td>
<td>(S + N^+)?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_h</td>
<td>S</td>
<td>S?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r_t</td>
<td>S?</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>h*6</th>
<th>q_h</th>
<th>q_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>h*6</td>
<td>S</td>
<td>S?</td>
<td>(S + N^+)?</td>
</tr>
<tr>
<td>q_h</td>
<td>S?,(S + N^+)?</td>
<td>S</td>
<td>(S + N^+)?</td>
</tr>
<tr>
<td>q_t</td>
<td>S?</td>
<td>S?</td>
<td>S</td>
</tr>
</tbody>
</table>

------------------- END COND -------------------

<table>
<thead>
<tr>
<th></th>
<th>h*6</th>
<th>q_h</th>
<th>q_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>h*6</td>
<td>S</td>
<td>S?</td>
<td>(S + N^+)?</td>
</tr>
<tr>
<td>q_h</td>
<td>(S + N^+)?</td>
<td>S</td>
<td>(S + N^+)?</td>
</tr>
<tr>
<td>q_t</td>
<td>S?</td>
<td>S?</td>
<td>S</td>
</tr>
</tbody>
</table>

------------------- 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;

{ --- AUXILIARY 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;
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
\[ \text{ll := l.left; } \\
\text{rl := r.left; } \\
\text{l.left := rl; } \\
\text{r.left := ll } \\
\text{end; } \{ \text{SwapLeft} \} \]

\textbf{procedure} SwapRight(l, r: handle)
\textbf{begin}
\text{l.r, rr: handle}
\text{begin}
\text{l.r := l.right; } \\
\text{rr := r.right; } \\
\text{l.right := rr; } \\
\text{r.right := l.r } \\
\text{end; } \{ \text{SwapRight} \} \]

\{ --- BITONIC MERGE --- \}

\textbf{procedure} Bmerge(root: handle; sp.r: handle; dir: int)
\textbf{begin}
\text{rightexchange, elementexchange, temp : int; } \\
\text{sp.l, pl, pr, rl, rr: handle}
\text{begin}
\text{rightexchange := (root.value > sp.r.value) xor dir; } \\
\text{if rightexchange then SwapValue(root, sp.r); } \\
\text{pl := root.left; } \\
\text{pr := root.right; } \\
\text{while (pl \neq nil) do}
\text{begin}
\text{elementexchange := (pl.value > pr.value) xor dir; } \\
\text{if elementexchange then}
\text{if elementexchange then}
\text{begin}
\text{swap values and right subtrees, search path goes left}
\text{begin}
\text{SwapValue(pl, pr); } \\
\text{SwapRight(pl, pr); } \\
\text{pl := pl.left; } \\
\text{pr := pr.left } \\
\text{end}
\text{else}
\text{search path goes right}
\text{begin}
\text{pl := pl.right; } \\
\text{pr := pr.right } \\
\text{end}
\text{else}
\text{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
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;
};

{ --- 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;
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 Bmerge();
void SSBmerge();
void Bisort();
void SSBSort();
void sil_main();
void sil_child();
#define DD_Bmerge 0
#define DD_Bsort 1
#define DD_EXIT 2

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

#ifdef EXTERN
extern char *procnames[];
#else
static char *procnames[] =
{
    "Bmerge",
    "Bsort",
    "EXIT"
};
#endif

D.2.3 File: bitonicp.c

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

#include <sys.h>    /* Uniform System */
#include <stdio.h>   /* Standard I/O */
#include NODEH       /* Node Definition */
#include PROCCH      /* Procedure Types/Nums */
#include "include/util.h" /* Utility macros */

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

D.2 Parallel C Program

```c
h = makesnode();
h->value = w;
h->left = NIL;
h->right = NIL;
return(h);
}

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

if ((n == 0))
    puts("\n");
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(" \n");
        else
            puts(" | \n");
    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))
{
    if (h->right)
        PrintSubTree(r,(n + 1),(path + 2));
    puts(" |
");
    put(n);
    puts(" | |
");
    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);
        put(" ");
        InOrder(r);
    }
}

HANDLE *
    /
    RandTree(n)
    /
    HANDLE *h;
{
    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
/***************/
Bismerge(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 (elementexchange)
        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_Bmerge(rl, sp_l, dir);
          SS_Bmerge(rr, sp_r, dir);
        }
      else /* more than one proc, fork off a procedure call */
        { 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_ptr[p];
          *((short *) p_arglist)++ = p_pending;
          *((int *) p_arglist)++ = DD_Bmerge;
          *((int *) p_arglist)++ = p_endpoint;
          *((HANDLE *) p_arglist)++ = rl;
          *((HANDLE *) p_arglist)++ = sp_l;
          *((int *) p_arglist)++ = dir;
          START(p); /* let processor p start */
          Bmerge(rr, sp_r, dir);
          WAIT_ZERO(p_pending);
          RELEASE_PENDING; /* release pending lock */
        }
      my_endpoint = input_endpoint;
    }

root->value = sp_l->value;

}

void
/***********
SS_Bmerge(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)
            { elementexchange
                { SwapValue(pl,pr);
                    SwapRight(pl,pr);
                    pl = pl->left;
                    pr = pr->left;
                }
            else
                { pl = pl->right;
                    pr = pr->right;
                }
        else
            { SwapValue(pl,pr);
                SwapLeft(pl,pr);
                pl = pl->right;
                pr = pr->right;
            }
        }
    }
if ((root->left != NIL))
    { rl = root->left;
        rr = root->right;
        sp_l = NewNode(root->value);
        { SS_Bismerge(rl,sp_l,dir);
            SS_Bismerge(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;
  Bmerge(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 == NULL))
     { 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_l, (! dir));
       }
       root->value = sp_l->value;
       SS_Bmerge(root, sp_r, dir);
     }
}

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

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

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

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

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

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


Bibliography


Bibliography


