

**AUTOMATIC GENERATION OF
DATA-FLOW ANALYZERS:
A TOOL FOR BUILDING OPTIMIZERS**

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Steven Weng-Kiang Tjiang
July 1993

© Copyright 1993 by Steven Weng-Kiang Tjiang
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

John L. Hennessy
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Mark A. Linton

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Daniel Weise

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Acknowledgements

I would like to thank Professor John Hennessy for being my advisor, and for building an excellent environment at the Computer Systems Laboratory for doing research and writing interesting software. This thesis would not have been possible without his advice and his support, confidence and patience.

I would like to thank Professor Daniel Weise and Professor Mark Linton, the other members of my reading committee. I would also like to thank Professor Monica Lam and Professor Donald Iglehart for serving on my orals committee.

The SUIF compiler system would not have been possible without the piglets: Saman Amarasinghe, Rob French, Monica Lam, Amy Lim, Dror Maydan, Jason Nieh, Karen Pieper, Mike Smith, Bob Wilson, and Michael Wolf. They suffered through the early, unstable stages of the compiler. Their unswerving devotion and attention to SUIF kept the entire compiler system working. To the piglets, I owe a special thanks.

Graduate school can be lonely and depressing. Fortunately, the bad moments were balanced by good times spent with my family and friends (Annette Ng, Eileen and Teresa Chin, Elizabeth Buss, Tim Corle, Bruce King, Steve Pollock, Geoff Reeves, Miko Yamaguchi) and by good times spent commiserating with the beer gang (K. Gopinath, Ramsey Haddad, Shing Kong, Scott McFarling, Steve Richardson, Larry Soule, Malcolm Wing, and Michael Wolf). Dave Ditzel and SUN labs graciously gave me a quiet hideaway to work. Special thanks goes to Charlie Orgish, Margaret Rowland and Darlene Hadding for keeping everything running smoothly in CIS.

Several people played pivotal roles during my Stanford years. Al Aho and Peter Weinberger, my unofficial mentors at Bell Labs, took time out of their busy schedule to talk with me when they came to visit Stanford. Monica Lam inspired me and the piglets with her keen insight, vivaciousness, and her willingness to chat and listen. Michael Wolf and Kurt Keutzer put life, gainful employment, research, and thesis all into perspective for me. Miko Yamaguchi and Michael Wolf kept me running and sane. Annette Ng, my wife, gave me patience, support and love, without which I would surely have given up.

This research has been supported by DARPA under contract N00014-87-K-0828 and N00039-91-C-0138.

To
my grandparents,
my parents,
and my wife.

Abstract

Modern compilers generate good code by performing global optimizations. Unlike other functions of the compiler such as parsing and code generation which examine only one statement or one basic block at a time, optimizers examine large parts of a program and coordinate changes in widely separated parts of a program. Thus optimizers use more complex data structures and consume more time. To generate the best code, optimizers perform not one global transformation, but many in concert. These transformations can interact in unforeseen ways.

This dissertation concerns the building of optimizers that are modular and extensible. It espouses an optimizer architecture, first proposed by Kildall, in which each phase is based on a data-flow analysis (DFA) of the program and on an optimization function that transforms the program. To support the architecture, a set of abstractions—flow values, flow functions, path simplification rules, action routines—is provided. A tool called *Sharlit* turns a DFA specification consisting of these abstractions into a solver for a DFA problem. At the heart of *Sharlit* is an algorithm called path simplification, an extension of Tarjan's fast path algorithm. Path simplification unifies several powerful DFA solution techniques. By using path simplification rules, compiler writers can construct a wide range of data-flow analyzers, from simple iterative ones, to solvers that use local analysis, interval analysis, or sparse data-flow evaluation.

Sharlit frees compiler writers from the details of how these various solution techniques. The compiler writer can view the program representation as a simple flow graph in which each instruction is a node. Data structures to represent basic blocks and other regions are automatically generated. *Sharlit* promotes modularity by making it possible to build more complex data-flow analyzers from simpler ones. *Sharlit* promotes extensibility because it is easier to add new flow functions and path simplification rules that represent new kinds of instructions to existing analyzers.

A complete optimizer has been built using *Sharlit*. Measurements showed that this optimizer can compete in code quality with commercial optimizing compilers.

Contents

Acknowledgements	iv
Abstract	vi
Contents	vii
List of Figures	xi
List of Tables	xiii
List of Procedures	xiv
Chapter 1	
Introduction	1
1.1 Sharlit—A Tool for Building Scalar Optimizers	2
1.1.1 Kildall’s Model for Optimizers	5
1.2 Integrating Parallelization and Scalar Optimization	5
1.2.1 Interactions between Parallelization and Scalar Optimization	6
1.2.2 Stanford University Intermediate Form	7
1.3 Organization of Dissertation	8
Chapter 2	
Data-flow Analysis—A Basis for Optimizers	10
2.1 Data-flow Analysis in Optimizers	11
2.1.1 Example: Reaching Definitions	12
2.1.2 Example: Liveness Analysis	13
2.1.3 Example: Constant Propagation	14

2.2	Solving Data-flow Analysis	15
2.2.1	Iteration: Exploiting Monotonicity	15
2.2.2	Elimination: Exploiting Flow Graph Structure	17
2.2.3	Sparse Data-flow Evaluation Graphs: Skipping Regions	19
2.3	Deficiencies of Existing Data-flow Analyzers	20
2.4	Automatic Generation of DFAs	21

Chapter 3

	Path Simplification	23
3.1	Tarjan's Fast Path Algorithm	24
3.1.1	Efficient Implementation of TFPA	25
3.1.2	Example: Applying TFPA	25
3.1.3	Path Compression	27
3.2	Path Simplification: An Extension of TFPA	27
3.2.1	Control-Flow Analysis	30
3.2.2	Simplify	31
3.2.3	Header Forest Routines	36
3.2.4	Reduce	36
3.2.5	Iterate and Propagate	37
3.3	Conclusion	38

Chapter 4

	Writing Data-flow Analyzers with Sharlit	40
4.1	Preliminaries	41
4.2	Example of Iterative Analysis	43
4.3	Eliminating Flow Variables	47
4.4	Path Simplification Rules	49
4.5	Local Analysis: Summarizing Extended Basic Blocks	53
4.6	Elimination: Summarizing Loops	55
4.6.1	Irreducible Graphs	56
4.7	Sparse Data-flow Evaluation Graphs	58
4.8	How Rules Compute Paths	58
4.8.1	Absorb Rules Save Space	59
4.8.2	Using Rules in Path Computation	61
4.9	Summary	61

Chapter 5	
An Architecture for Modular and Flexible Optimizers	63
5.1 Data-flow Analyzers and Control-Flow Graphs	64
5.1.1 Data-Flow Analyzers or Solvers	64
5.1.2 Control-Flow Graphs	67
5.1.2.1 Control-Flow Analysis	69
5.2 Combining Data-flow Analyzers	74
5.2.1 Chains	74
5.2.2 Products	75
5.2.2.1 An Example: Constant Propagation with SDFEGs	75
5.2.2.2 Product Flow Values and product specifications	78
5.2.2.3 Modifying and Non-Modifying Solvers	78
5.2.2.4 Functions of Generic Solver for Supporting Products	78
5.2.2.5 Initialization	79
5.2.2.6 Path Simplification	81
5.2.2.7 Reduction	82
5.2.2.8 Iteration	83
5.2.2.9 Propagation	83
5.3 Summary	85
Chapter 6	
A Prototype Optimizer	86
6.1 Structure of the Prototype Optimizer	86
6.1.1 Flow Graph Nodes	89
6.2 Code Motion	89
6.2.1 Value Numbering	90
6.2.2 Availability and Partial Availability	92
6.2.3 Placement: Where to Insert Expressions	93
6.2.4 Implementation of Placement with Sharlit	96
6.2.5 Code Generation	97
6.3 Strength Reduction	97
6.3.1 Value Numbering	99
6.3.2 Changes to the EPR Algorithm	99
6.4 Induction Variable Rewrite	100
6.4.1 Representing Relationships between Induction Variables	102
6.4.2 Representing Loop Structures	103
6.4.3 Computing Induction Variables and Their Step Size	103

6.4.4	Propagating Induction Variable Relationships	105
6.5	Conclusions and Results	105
Chapter 7		
SUIF		107
7.1	Overview of the SUIF IL	107
7.1.1	Registers	108
7.1.2	Paths	111
7.1.3	High-SUIF	113
7.2	Annotations	114
7.3	Overview and Status of the SUIF compiler system	115
7.3.1	High-level Transformations and Analysis	117
7.3.2	Status and Experience	117
Chapter 8		
Conclusion		118
8.1	Future Work	120
8.1.1	Flow Values	120
8.1.2	Efficient Iteration	120
8.1.3	Flow Graphs and Incremental Updates	121
8.1.4	More Experience with Sharlit	121
Bibliography		123

List of Figures

Figure 1-1	The structure of an optimizer	3
Figure 1-2	A simple flow graph	5
Figure 1-3	Example of a simple loop	6
Figure 2-1	Eliminating back-edges	17
Figure 2-2	Two flow graphs and their SDFEGs	20
Figure 2-3	Skipping over identity nodes	21
Figure 3-1	A simple graph	26
Figure 3-2	Header forest after loop D	26
Figure 3-3	Graph and header forest after processing E within loop A	27
Figure 3-4	Header forest after processing F	28
Figure 3-5	Header tree after processing outermost region	28
Figure 3-6	Outline of how Sharlit solves a data-flow problem	30
Figure 3-7	Basic path expression computation in Simplify_node	32
Figure 3-8	Path simplification fails	35
Figure 3-9	Simplification after being stopped by an irreducibility	35
Figure 4-1	A phase of the an optimizer	42
Figure 4-2	Structure of a data-flow analyzer	42
Figure 4-3	Using iteration to solve for available expressions	45
Figure 4-4	Has_meet and has_var nodes	48
Figure 4-5	Applying the “ p create” rule	50
Figure 4-6	Applying the “ p create f ” rule	50
Figure 4-7	Applying the “ p absorb f ” rule	51
Figure 4-8	Applying the “ p star” rule	51
Figure 4-9	Applying the “ p_1 join p_2 ” rule	51
Figure 4-10	Example of applying several path simplification rules	52
Figure 4-11	Copying paths	52
Figure 4-12	Simplifying extended basic blocks	53
Figure 4-13	Using local analysis to solve for available expressions	54
Figure 4-14	Flow function declaration with in and out functions	55

Figure 4-15	Enhancing solver to use interval analysis	57
Figure 4-16	Path simplifying irreducibilities	57
Figure 4-17	Rules for doing SDFEG	58
Figure 4-18	Example of simplification SDFEG	59
Figure 4-19	Reusing flow functions in sequences of node	60
Figure 5-1	Computing the data-flow solution at a node	67
Figure 5-2	Example of control-flow analysis	73
Figure 5-3	Creating a chain	75
Figure 5-4	Multiplying two SDFEGs to do constant propagation	76
Figure 6-1	Structure of the SUIF optimizer	87
Figure 6-2	Redundancy and partial redundancy	91
Figure 6-3	How partially redundant insertions occur	95
Figure 6-4	Using partial-availability elimination for strength reduction	98
Figure 6-5	Induction variable rewrite	100
Figure 6-6	Loop summarization	101
Figure 6-7	Control-flow graph for a loop	103
Figure 7-1	Program fragment and associated SUIF code	109
Figure 7-2	A Path Forest	112
Figure 7-3	The SUIF compiler system	116

List of Tables

Table 5-1	Functions provided by Sharlit for solving DFA problems	65
Table 5-2	Functions generated by Sharlit	66
Table 5-3	Functions for building and modifying the control-flow graph	69
Table 5-4	Functions used by generic solver to access flow graph	70
Table 5-5	Iteration step of a product solver	77
Table 5-6	Functions of generic solver for supporting products	79
Table 6-1	Fields of a flow graph node (class IRnode)	89
Table 6-2	Comparison of SUIF optimizer with a commercial optimizer	106

List of Procedures

Procedure 3-1	Control-flow analysis	31
Procedure 3-2	Simplifying a graph	32
Procedure 3-3	Computing the simplified graph	33
Procedure 3-4	Header forest routines	34
Procedure 3-5	Reduce	36
Procedure 3-7	Propagation	37
Procedure 3-6	Solution with iteration	38
Procedure 4-1	Solution by iteration	44
Procedure 4-2	Reduce revisited	47
Procedure 4-3	Propagation: finding solutions at eliminated nodes	56
Procedure 5-1	Control-flow analysis in detail	72
Procedure 5-2	Paraphrase of <code>DF::iterate</code> and <code>DF::propagate</code> .	80
Procedure 5-3	Initialization phase of product solver	80
Procedure 5-4	Sketch of new <code>Simplify_node</code> and <code>simplify_region</code>	81
Procedure 5-5	The reducer of a product solver	82
Procedure 5-6	Reconstruction of product flow value and the iterator	84
Procedure 5-7	The product propagator, <code>CP::propagate</code>	85

1

Introduction

Compiler construction in the last decade has increasingly emphasized global analyses and transformations of programs—for example, scalar optimizations, parallelization, and vectorization. Such global techniques are important in at least two ways. First, they improve the usability of sophisticated, high-performance computers by letting programmers program these computers in high-level languages with minimal performance penalty. Second, they complement the hardware in achieving high performance because they use global information to generate code that can use architectural features such as registers and multiple functional units more effectively. Such global information would be too expensive to discover with pure hardware techniques, would be too unreliable if given by the programmer, or would be too tedious for the programmer to give.

Unfortunately, increasing reliance on global techniques makes compilers larger and more complex. Building such a compiler constitutes a substantial undertaking, requiring years of development. The compiler is larger because generating the best code requires not one global transformation, but many working in concert. The compiler is more complex because these transformations can interact in unforeseen ways.

The difficulty of compiler construction is impeding the development, evaluation and adoption of new computer architectures. We need to make global analyses and transformations easier to build. We need a compiler that we can easily re-targeted to new processors so that the substantial effort invested in building the compiler will not go to waste. To re-target compilers for high-performance computers, we need more than re-targetable instruction selection. We need to fine-tune the global transformations, and to add new transformations. We need a compiler that is flexible and that is extensible.

Scalar optimizers exemplify this inflexibility. This is because global analyses and transformations consume more time than traditional compiler algorithms such as parsing and instruction selection. Efficiency concerns have resulted in scalar optimizers burdened with

implementation details that obscure the algorithms. The sequences of transformations in these compilers are rigid: To increase efficiency, each optimization depends on what other optimization or analyses goes before and after it. The rigid ordering hinders the reordering of the transformations and the addition of new ones.

This dissertation addresses two issues in building flexible and extensible compilers. A major portion of the thesis discusses Sharlit, a tool that simplifies the building of data-flow analyzers and scalar optimizers—both of which form significant portions of any compiler. This thesis also presents a new intermediate language SUIF (Stanford University Intermediate Form) that integrates conventional scalar optimizations with high-level transformations such as parallelization and high-level analyses such as dependence analysis. Both Sharlit and SUIF are part of the SUIF compiler, a parallelizing and optimizing compiler being developed at Stanford.

1.1 Sharlit—A Tool for Building Scalar Optimizers

Sharlit is a tool for building scalar optimizers. An optimizer performs program transformations such as constant propagation and folding, forward and backward code motion, removing useless computations, and register allocation [2]. These transformations are applied generally to computations involving scalars—including addressing computations for array accesses.

Figure 1-1 shows the structure of an optimizer. It is logically composed of several phases. The front-end of the compiler translates a program into an intermediate language (IL). The optimizer reads each procedure of the IL program. Each phase of the optimizer then analyzes or transforms the procedure. A typical sequence of phases would be to do value numbering, constant propagation, code motion, and dead store elimination. When the phases complete, the optimizer writes out the IL program.

The larger scope of scalar optimizations sets it apart from other compiler algorithms such as parsing and instruction selection. Parsing and instruction selection need to examine only one statement or one basic block at a time. In contrast, optimizers need to examine and change large portions of the program all at once. The larger scope means optimizers must detect and use relationships among widely separated parts of a program. Compared to the local information used in front-ends and code generators, these relationships are more complex, and more time-consuming to extract. Furthermore scalar optimizers have lots of phases. Each phase has different information requirements; each has particular demands on the program representation.

Despite the complexity of optimizers, few tools exist for building them. This situation stands in stark contrast with that of other parts of the compiler where years of experience

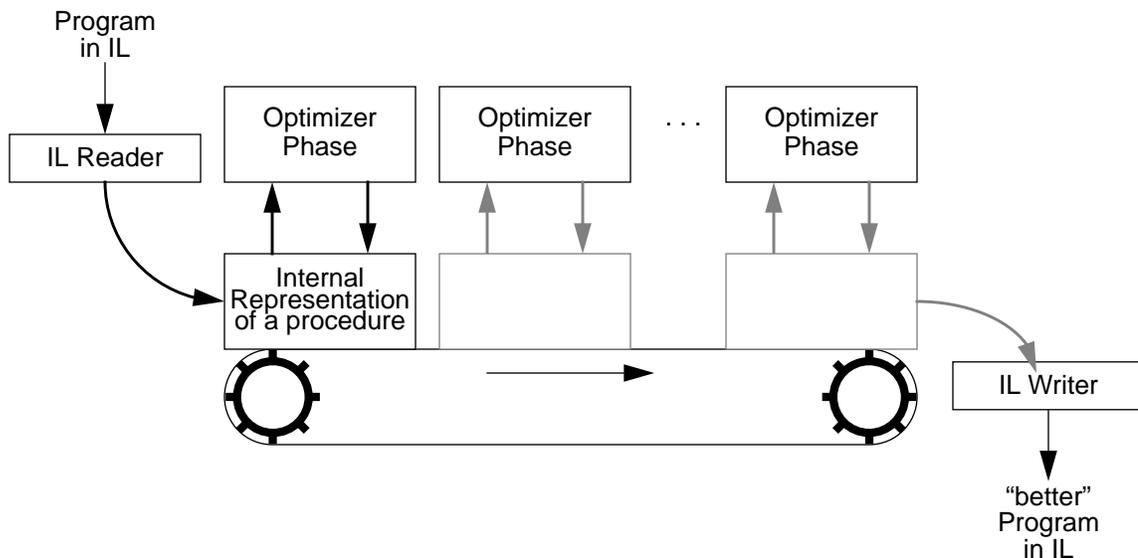


Figure 1-1 The structure of an optimizer

Conceptually, optimizers are organized as a fixed sequence of phases. Before optimization, the source program is translated into an intermediate language (IL) by a front-end. The optimizer reads the IL program one procedure at a time, converting it into an internal representation (IR), typically a control-flow graph. Each optimizer phase operates on the flow graph, analyzing and transforming it. When all the phase are finished, the IR is written out as a better or more efficient IL program

have culminated in models and structures—embodied in specialized tools—with which to construct these other parts. For example,

- Lexical analysis generators: LEX.
- Parser generators: YACC[48].
- Front-end analyses: Attributed grammars [72].
- Instruction selection: Abstract Interpretation[27], twig[1], Graham-Glanville code-generator generators[38, 45, 44], attributed parsing[32, 33].

These tools provide a layer of abstractions to hide implementation details from the compiler writer, and to provide powerful algorithms that can be tailored to the needs of compiler writers. With the tools, writing these other compiler parts has become much easier.

The lack of tools, together with a justified concern for efficiency, have meant that most optimizers are built in an ad hoc manner, with implementation details that complicate transformations unnecessarily. The use of basic blocks, for example, imposes a two-level structure on transformations. There is a local level that deals with analysis and transformations within basic blocks, and a global level that deals with the control-flow graph whose nodes are basic blocks. The two levels are introduced to lower the space and time required to per-

form data-flow analysis (DFA). However, the levels force DFA to proceed in three steps. A local analysis phase computes the data-flow effect of each basic block by composing the data-flow effects of its instructions. A global solution phase uses the computed effects to find the solution of the DFA problem at the entrance and exit of each block. Finally an ad hoc propagation step computes the data-flow information at each instruction from the information at the entry or exit of the block.

Sharlit insulates compiler writers from implementation details of optimizers, just as other tools do for other parts of the compiler. Sharlit provides a more abstract view of DFA consisting of the following abstractions:

- *Flow graphs* consists of nodes that are data structures specified by the compiler writer.
- *Flow values* represents the data-flow information that flows through the flow graph. A solution to the DFA problem consists of a set of flow values, one for each node.
- *Flow functions* represent data-flow effects, the effects on flow values as they pass through nodes.
- *Action routines* use the DFA solution to perform program optimizations. The routines are analogous to action routines of a parser description.
- *Path simplification rules* show how to combine flow functions into other flow functions.

These abstractions let the compiler writer describe DFA, unencumbered by the implementation details of the flow graph, and of how the DFA is solved. In particular, basic blocks are not necessary for efficient data-flow analysis—The flow graph is simpler, because each node is an individual IR instruction. These abstractions are general in that they can implement a wide range of program analyses, from those based on traditional bit-vector to those based on sophisticated symbolic analysis. The key to achieving this versatility is an algorithm called path simplification.

Path simplification, a version of Tarjan’s fast path algorithm [82], computes a *path expression* for each node u in the flow graph. This expression represents all paths from the source to u . A path expression is a regular expressions built from node labels and the operators: \cdot , $+$, and $*$. For example, in Figure 1-2, the path expression from the exit of the source node to the exit of node D is $A ((B + C) DA)^* (B + C) D$. Sharlit uses path expressions in data-flow analysis by interpreting the node labels as flow functions, path concatenation(\cdot) as function composition, path addition($+$) as taking the *meet* of functions, and the Kleene-star¹ ($*$) as finding the flow function for a loop. In short, interpreting a path expression p in this manner yields the data-flow effect of the paths represented by p .

1. $A^* = \varepsilon + A + AA + AAA + \dots$ where ε is the empty path or identity function.

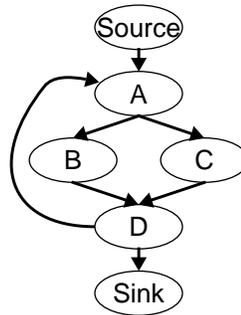


Figure 1-2 A simple flow graph

The compiler writer tells Sharlit how to perform this interpretation with path simplification rules. The rules can be used to implement local analysis or interval analysis, and to compute sparse data-flow evaluation graphs. The rules can be applied to both reducible and irreducible flow graphs.

1.1.1 Kildall's Model for Optimizers

Using Sharlit, compiler writers would organize each phase of their optimizers as a data-flow analyzer followed by an *optimization function* that uses the solution of the DFA problem to modify the program. This is an organization first proposed by Kildall [54]. This organization is general in that it can implement most classical optimizations. The structure of these optimization is usually: a data-flow analysis step that collects information, followed by an IR graph traversal that checks at each node whether the collected information enables the optimization. For example, Aho, Sethi and Ullman[2], and Chow[20] use such an organization to describe their optimizations.

Sharlit extends Kildall's model with path simplification and with techniques for combining simpler data-flow analyzers into more complex ones. With the two extensions, compiler writers can make optimizers simpler, modular, and extensible. The optimizations operate on simple flow graphs in which each node is an instruction. Each phase of the optimizer is a module—consisting of a data-flow analyzer with its parts expressed as flow functions and simplification rules. If format of the IR is changed, the changes can be accommodated by adding new flow functions and simplification rules. Each analyzer in turn can be composed of smaller, simpler analyzers.

1.2 Integrating Parallelization and Scalar Optimization

Another aspect of compilation covered by this thesis is how scalar optimizations and parallelizing optimizations can be integrated. Compiling programs to use parallelism and

```

1      n = 49;
2      k = 0;
3      for(i = 0; i<=n; i++){
4          A[k] = A[200];
5          k += 2;
6      }

```

Figure 1-3 Example of a simple loop

memory hierarchy efficiently requires both parallelizing transformations and scalar optimizations, which interact in many ways. Scalar transformations and analyses are prerequisites to many parallelizing transformations. Likewise, information gathered in parallelism analyses can improve scalar optimizations. For machines with instruction-level parallelism, scalar and parallelizing transformations are intimately linked. The key challenge in integrating these transformations is that they require different kinds of knowledge and manipulate the code differently.

1.2.1 Interactions between Parallelization and Scalar Optimization

Existing parallelizing compilers have two stages. In the first stage the compilers, using source or abstract syntax trees as its high-level representation, parallelize a program via a source-to-source translation. In the second stage the parallelized source is compiled and optimized by a traditional scalar compiler.

One can distinguish scalar from parallelizing transformations by the information they use. Scalar optimizations rely on information gathered by traditional data-flow analysis, while parallelizing transformations rely on data-dependence information, information that tells us whether two array references can refer to the same location, and if so, the relationships between them. Many compilers strictly separate the two categories of transformations by using two representations: a low-level one such as Ucode[37] and a high-level one such as abstract syntax trees. When there are two representations, many of the functions provided by scalar phases must be re-implemented for each representation, and high-level analyses can no longer directly communicate with low-level transformations.

Scalar phases are valuable for preparing the code for data-dependence testing. For example, many dependence tests require that the array indexes are affine functions of the loop index variables, and the bounds of the arrays are constants. Consider the code shown in Figure 1-3. Before testing, we apply *induction variable expansion*¹ and constant propagation, two scalar optimizations. Induction variable expansion translates k into $2i$, an

1. Induction variable expansion detects induction variables within for-loops, and expresses them in terms of the loop index.

affine function. By propagating the constant to the variable n , the data-dependence tester can tell that $A[k]$ cannot refer to $A[200]$ and the loop is therefore parallelizable.

Scalar phases simplify many transformations by cleaning up after them. Constant propagation, for example, can potentially create dead code, which must be eliminated before scalar privatization. Not only does this reduce the work for scalar privatization, but privatizing dead code has a large performance penalty. Moreover, eliminating dead code is also necessary to remove code made redundant by scalar optimizations. In both instances, the SUIF compiler eliminates dead code with the same phase.

If, in the situations above, we had used two totally different representations for scalar and parallelizing transformations, then we would have to write two different sets of optimizations, as in source-to-source compilers. There would, for example, be two different dead-code elimination phases.

When one IL can support the two categories of transformations, the scalar phases and parallelizing phases can be freely intermixed. This freedom permits us to take a tool-based approach to building the SUIF compiler. Each phase is like a tool. A tool can prepare the code for other tools, just as constant propagation and induction variable expansion does for data dependence analysis. A tool can clean up after others. Because each tool is simple, it is easy to implement and test.

Integrating the two levels has another advantage, the ease with which information can flow from high-level to low-level phases. In superscalar instruction scheduling, for example, it is not sufficient just for the high-level analyzer to label a loop as a DOALL loop. Even non-DOALL loops, those with loop-carried data dependences, contain useful parallelism. In general, high-level data-dependence information must be available to a superscalar scheduler, so that it has maximum flexibility in reordering loads and stores within loops.

1.2.2 Stanford University Intermediate Form

The SUIF compiler has only one program representation, integrating both high and low levels in one intermediate language. Both parallelizing and scalar phases read and write programs in the same intermediate language. This integration offers two key advantages over existing compilers.

- *No duplication of effort.* Because they use two different representations, traditional compilers require two versions of many transformations and analyses, one in the first stage when high-level representation is used and another in the second stage when low-level representation is used. Instead, our compiler has only one version.

- *Availability of high-level information at the low level.* Having two representations hinders the communication of high-level information to low-level transformations. Consider a low-level transformation like instruction scheduling. For it to use high-level information such as data-dependence and aliasing information, the information must be attached to individual loads and stores. Doing this at the source level, which is not sufficiently fine grained to identify individual memory references, requires auxiliary files or ad hoc commenting conventions. Our compiler provides a simple mechanism called annotations to attach information to individual instructions.

The SUIF intermediate language supports a wide range of both high- and low-level transformations. SUIF has a small core of simple instructions that manipulate an infinite number of virtual registers. Although this core is commonly considered to be suitable only for low-level scalar optimizations, this thesis shows how the core can be made suitable for high-level transformations. For example, SUIF has explicit representations for high-level control structures (for-loops, if-then-else) and an array indexing instruction; both are features necessary for data-dependence testing.

The SUIF compiler is a set of scalar and parallelizing phases that can be reordered. The scalar phases are implemented with Sharlit. By adding new flow functions, we extend these phases to recognize high-level control flow and indexing instructions. Once extended, the scalar phases can be used at both the high- and low-levels, before and after parallelization.

1.3 Organization of Dissertation

This rest of this thesis is divided into seven chapters.

The second chapter discusses data-flow analysis and how it can serve as a model for optimizers. I show several DFA problems commonly used in optimizers, and outline several techniques used to solve the problems.

The third chapter presents our path simplification algorithm. Path simplification is my modification of Tarjan's fast path algorithm. I show the algorithm in detail, and show how Sharlit uses it to on a flow graph to solve DFA problems efficiently, even when they are posed on graphs in which the nodes are individual IR instructions.

The fourth chapter demonstrates Sharlit's versatility and how Sharlit can generate space-efficient solvers. I give several examples that compute reaching definitions and available expressions. These examples demonstrate how progressively more sophisticated data-flow analyzers can be specified.

The fifth chapter presents the architecture that Sharlit provides for building optimizers. This architecture includes abstractions for control-flow graphs, and for combining Sharlit solvers (data-flow analyzers).

The sixth chapter discusses the SUIF scalar optimizer constructed using Sharlit. I detail some of the optimization algorithms: partial redundancy elimination, strength reduction and induction variable rewrite. I outline the other optimizations. I also give experimental results showing that this optimizer is effective.

The seventh chapter presents SUIF, and describes how macro-expansion and certain conventions in the use of names permit us to integrate both types of program transformations.

The eighth and final chapter concludes the thesis, and indicates some directions for future research.

2

Data-flow Analysis—A Basis for Optimizers

To determine which program transformations to apply and where in the program to apply them, the optimizer needs information about the run-time behavior of a program. This information is gathered by data-flow analysis (DFA). DFA computes data-flow information, which are assertions or conditions on program states at every point in the program. Determining whether such assertions are true on every execution of a program can be tantamount to solving the halting problem. Thus, data-flow analysis can only estimate the run-time behavior—in general, DFA can determine whether the assertions are true or false or undetermined. A general technique to do this estimation is abstract interpretation [22], which maps a user program into equations over some abstract domain. Solving these equations yields the desired data-flow information.

DFA has been extensively studied in the literature. The properties of DFA problems have been formalized by many authors: Cousot [22], Graham and Wegman [39], Hecht [42, 43], Kam and Ullman [52, 53], Marlowe and Ryder [62], Ryder [76], and Tarjan [83]. Efficient algorithms to solve DFA problems have been developed by Choi et al [19], Graham and Wegman [39], Hecht and Ullman [43], Horwitz et al [46], Tarjan [82], and Ullman [86].

Kildall [54] was the first to recognize that program analysis can serve as a basis for implementing optimizations. By choosing the appropriate abstract domain, a data-flow analyzer can compute not only simple information such as reaching definitions and liveness, but detect recurrences and perform code motion. Several authors—Chow [20], Joshi and Dhamdhere [50, 51], and Morel and Renvoise [65]—have used Kildall’s model; they used data-flow analysis as the centerpiece of their optimizations. This dissertation presents a tool that supports Kildall’s model directly.

The chapter gives a general model for DFA problems that are commonly solved in optimizers. It describes the concepts of flow values, flow functions, lattices, and meet functions. It gives several methods which are commonly used to solve DFA problems. It closes by discussing the deficiencies of current implementations of data-flow analyzers.

Sharlit is primarily concerned with intraprocedural optimizations. Thus the model of DFA we use applies only to a single procedure. Below, the word *program* should be taken to mean *procedure*.

2.1 Data-flow Analysis in Optimizers

Many optimizers perform DFA on programs represented as a directed graph called a control-flow graph (CFG). For many, the CFG consists of nodes that are basic blocks, and the edges represent flow of control. Instead, I propose that DFA be performed on a graph in which each node is simpler, representing one operation or instruction. The nodes of the graph are $s = u_1, u_2, \dots, u_N = t$ where s and t are respectively the unique source and final nodes of the CFG, and N is the number of nodes in the CFG. Usually each of these nodes are simple instructions of the form:

$$\begin{aligned}x_0 &\leftarrow x_1 \\x_0 &\leftarrow o(x_1, x_2, \dots)\end{aligned}$$

where $\{x_0, x_1, x_2, \dots\}$ are either program variables or constants and o is some operator. An *execution path*, a trace of all the operations executed on some execution of a procedure is a sequence $p_1 p_2 \dots p_l$ where each p_i is one of the nodes. Within this path, we can refer to one operation as being executed *before* or *after* another.

The model of DFA that Sharlit uses has the following parts:

- *flow values*: Sharlit represents an assertion about the program state with a flow value. Each node u of the CFG has two *flow variables* to hold flow values: $I(u)$ representing an assertion true before the node, and $O(u)$ representing an assertion true after the node.
- *meet operator*: When several flow graph paths meet, Sharlit combines flow values with the commutative meet operator which is frequently written as \wedge .
- *flow functions*: Sharlit associates a function f_i with each node u_i . The function maps flow values to flow values, and represents the data-flow effect of the instruction at u_i .
- *flow equations*: Using flow functions and the flow variables, Sharlit forms equations relating the variables

$$\{I(u_1), I(u_2), \dots, I(u_N), O(u_1), O(u_2), \dots, O(u_N)\}.$$

The equations will be composed of flow functions and meet operators. We will give examples of such equations in the following subsections. To solve the DFA, the program is mapped into a set of flow equations. A solution to the equations consists of two flow values for the variables $I(u_i)$ and $O(u_i)$ for each node u_i .

To the above parts, Sharlit adds path simplification rules and action routines. The rules are used to generate efficient data-flow analyzers. Action routines are given the solutions $I(u_i)$ and $O(u_i)$ which they use to modify the program. Rules and action routines are discussed in later chapters.

The following examples illustrates how to describe several data-flow analyses commonly found in existing optimizers with this model. Our examples do not use basic blocks, thus there is no need to explain how to arrive at the data-flow effect of a basic block from the effects of each instruction of that block, nor to distinguish local from global data-flow information and effects.

2.1.1 Example: Reaching Definitions

Reaching-definitions analysis determines which assignments to a variable in the program affect which uses of a variable in the program. Normally, this problem is solved simultaneously for several variables. However, we simplify the description by only stating the problem for a single variable x .

Let the set $D = \{d_1, d_2, \dots, d_m\}$ be the set of definitions—operations in a procedure that may define or assign values to x . A definition d *reaches* a node u if there is at least one execution path¹ that executes d before executing u , and there are no other definitions executed between d and u . In that execution path the value assigned to x by d is the value of x at u . A solution to the reaching definitions problem consists of a subset R of D reaching each node. Thus, for every possible execution of a procedure the value of x at a node will equal the value assigned to x by some definition in the set R .

The parts of the reaching definitions problem are given by the following definitions.

- *flow values*: We represent definitions reaching a node as subsets of D . Most optimizers use bit sets to represent the subsets of D .
- *meet operator*: When several control-flow edges meet, a definition reaching the meeting point must have arrived on one of the control-flow paths. Therefore the meet operator \wedge is set union \cup .

1. To avoid the halting problem, all loops are assumed to eventually exit. Thus, all execution paths are assumed to be unbounded but finite.

- *flow functions*: If a node u_i is a definition d_j then u_i 's flow function is a constant function f_i such that for all s , $f_i(s) = \{d_j\}$. If u_i is not a definition then f_i is the identity, that is for all s , $f_i(s) = s$.
- *flow equations*: For a CFG, the set of equations are:

$$I(u_1) = \emptyset$$

$$I(u_i) = \bigcup_{u_j \in \text{predecessor}(u_i)} O(u_j)$$

$$O(u_i) = f_i(I(u_i))$$

Thus, there are two equations for each node u_i , one for the input and one for the output.

Some optimizers use reaching definitions to do constant propagation. If all the definitions of the variable x that reach a node u assign the same constant c to x then the optimizer can directly replace all uses of x in the node u with c . This test of the definitions can be done in an optimization function as suggested by Kildal.

2.1.2 Example: Liveness Analysis

The reaching-definitions problem propagates information *forward* in the CFG—it determines which definitions may have been executed *before* a node. But DFA problems can also propagate information *backward*. One example is liveness analysis, which determines the uses of variables that executed *after* a node. The liveness problem for a single variable x can be stated as follows.

Let the set $U = \{w_1, w_2, \dots, w_m\}$ be the set of uses—operations in a procedure that use the variable x . The variable x is *dead* after a node u if all execution paths leading from u to the exit of the procedure do not execute any member of U ; otherwise x is *live*. A solution to the liveness problem consists of an indication of whether x is live or dead at each node.

Just as for reaching definitions, we can define flow values, the meet operator, flow functions, and flow equations. But this time, the role of I_i and O_i are reversed, showing that information propagates backwards. The parts are defined below.

- *flow values*: We can represent the liveness of x by a boolean value.
- *meet operator*: Because information flows backwards, we use the meet operator not at a point where control-flow edges meet, but at a point where several control-flow edges leave a node. At such a point, x is live if it is live at any of the successors, thus the meet operator is boolean disjunction \vee .
- *flow functions*: If a node u_i is a use w_j then f_i is a constant function such that for all s , $f_i(s) = \text{true}$; if u_i is not a use then $f_i(s) = s$ for all s .

- *flow equations*: For this problem, the flow equations are:

$$O(u_N) = \text{false}$$

$$O(u_i) = \bigvee_{u_j \in \text{successors}(u_i)} I(u_j)$$

$$I(u_i) = f_i(O(u_i))$$

Optimizers use liveness analysis to detect computations that store values in dead variables. Such computations can be removed, because their results will never be used.

2.1.3 Example: Constant Propagation

In the previous examples, all the flow functions were either identity or simple constant functions. However, some DFA problems have flow functions that resemble an interpretation of a program's instructions. One such problem is Kildal's constant propagation problem [54].

This problem tracks the values of the variables and performs computations with the tracked values. In this DFA a flow value maps each program variable to a value. We can represent such a flow value as a set of ordered pairs (x, c) in which x is a program variable, and the value of c is either a constant or the special value \perp called *bottom*. If the pair (x, c) is in $I(u_i)$ and c is a constant then the variable x will have the value c just before the node u_i is executed. If c is the special value \perp , that indicates that the DFA cannot determine whether x is a constant or not— x may be a constant but the DFA cannot determine the value of that constant.

Constant propagation requires that we extend the operations of the program to accept \perp . If o is an operator, then define o_\perp such that

$$o_\perp(x_1, x_2, \dots) = o(x_1, x_2, \dots)$$

if none of the arguments $\{x_1, x_2, \dots\}$ is bound to \perp , otherwise $o_\perp(x_1, x_2, \dots) = \perp$.

The parts of the DFA are defined below:

- *flow values*: A flow value contains an ordered pair (x, c) for each program variable x .
- *meet operator*: We define the operator \wedge as follows: $(x, c) \in A \wedge B$ if and only if $(x, c) \in A$ and $(x, c) \in B$, otherwise $(x, \perp) \in A \wedge B$.
- *flow functions*: We define the flow functions according to the type of the instruction:
 - If the node u_i is the instruction $x_o \leftarrow x_1$ then

$$(x_0, c) \in f_i(S) \quad \text{if } (x_1, c) \in S$$

$$(y, c) \in f_i(S) \quad \text{if } (y, c) \in S \text{ for all } y \neq x_0$$

In other words, f_i is the identity for all $(y, c) \in S$ where $y \neq x_0$.

- If the node u_i is the instruction $x_o \leftarrow o(x_1, x_2, \dots, x_k)$ then

$$(x_0, o_{\perp}(c_1, c_2, \dots, c_k)) \in f_i(S) \quad \text{if } (x_1, c_1) \in S, \dots, (x_k, c_k) \in S$$

$$(y, c) \in f_i(S) \quad \text{if } (y, c) \in S \text{ for all } y \neq x_0$$

- *flow equations*: The flow equations are defined as in the reaching definitions problem, with the set union operation replaced by the meet operator defined on the mappings, as described above. The initial value of the input flow variable for the start node is:

$$I(u_1) = \{ (x, \perp) \mid \text{for all variables } x \}$$

Thus, solving the constant propagation problem requires translating the operations of the program into more complex functions than the previous examples.

Instead of using reaching definitions to propagate constants, the optimizer can use the above DFA directly: If $(x, c) \in I_i$ and c is not \perp then any uses of x in the node u_i can be replaced by c .

2.2 Solving Data-flow Analysis

After posing a DFA problem by defining the flow values, the meet operator, and the flow functions, and after mapping the procedure into a set of flow equations, compiler writers have several methods at their disposal to solve the equations. The commonly used techniques fall into three classes: iteration, elimination, and sparse data-flow. The following sections discuss briefly all three, and gives the conditions under which they work best. Sharlit combines all three into one solution method—more on this in the next chapter.

2.2.1 Iteration: Exploiting Monotonicity

Iteration, the simplest of the three, is the main method used to solve DFA problems in optimizers. Iteration requires no analysis of the CFG, and works directly on reducible and irreducible flow graphs. To solve a set of data-flow equations with iteration, set all the flow variables $I(u_i)$ and $O(u_i)$ to an initial value, then evaluate the equations repeatedly—as if they were assignments—until reaching a fixed point for $I(u_i)$ and $O(u_i)$. The initial value used depend on the particular DFA problem, as described below.

The above procedure will converge if the flow values and flow functions obey these two

properties:

- *Flow values form a semi-lattice [61] with finite descending-chain condition (FDCC):* The meet operator of the DFA must satisfy the relationship $x \wedge y \leq x$ where \leq is the partial order of the semi-lattice. The finite descending-chain condition states that every non-increasing sequence

$$I_o \geq I_1 \geq I_2 \geq \dots$$

of flow values must be finite. Intuitively the partial order of the lattice compares the relative information content of two flow values. If $I \leq J$ then J has more information than I [62].

- *Monotonicity:* The flow functions $\{f_n\}$ satisfy the relationship

$$I \leq J \Rightarrow f_i(I) \leq f_i(J)$$

where I and J are flow values. Monotonicity ensures that during iteration I_i and O_i are assigned progressively smaller flow values. Thus if the initial value of I_i and O_i are larger (with respect to the partial order \leq of the semi-lattice) than some fixed point, then iteration will converge to them. The finite descending-chain condition guarantees that convergence occurs in a finite time.

- *Distributivity:* Every flow function f of the DFA problem distributes over the meet operator, that is

$$f(x \wedge y) = f(x) \wedge f(y)$$

Though not necessary for convergence, distributivity pertains to the quality of the solution found by iteration. Distributivity guarantees that the fixed point found by iteration will be the largest possible (with respect to \leq). Since every solution to the DFA problem must be a fixed point, distributivity ensures that no solution has more information than the solution found by iteration.

All three DFA problems described in the previous sections satisfy the above conditions, and they can be solved by iteration. For example, in reaching definitions the subset of D forms a semi-lattice with the partial order \leq being the set inclusion \subseteq ; both types of flow functions, the identity and $f_i(S) = \{d_j\}$ are monotonic; and since D is finite, every descending chain must be finite.

Although iteration requires no CFG analysis, examining the graph to create an order in which the equations are evaluated will make iteration converge faster. Hecht and Ullman [43] have shown convergence is asymptotically fastest—requiring the fewest equation evaluations—when the equation order corresponds to a topological sort of the graph ignoring the back-edges of loops. This order, called *reverse post-order*, can be computed in one

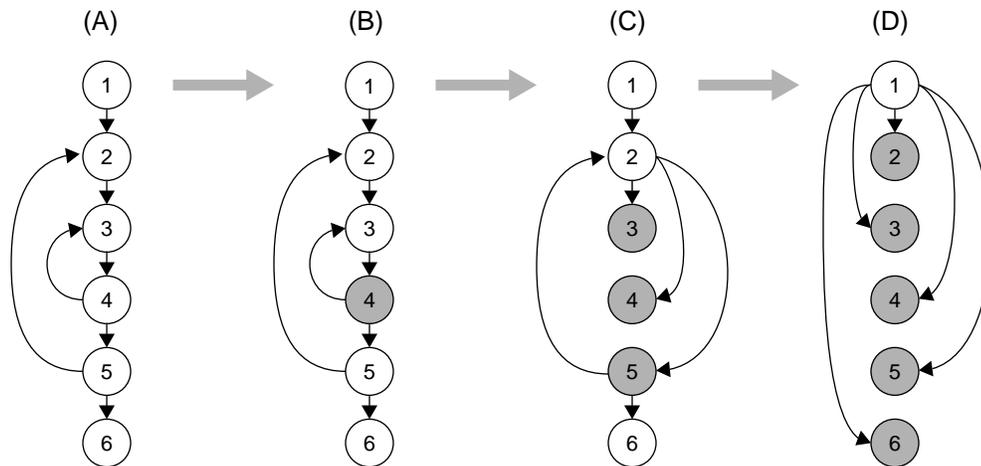


Figure 2-1 Eliminating back-edges

We show a graph as its back-edges are eliminated. We shade each node as they are processed. The process of removing back-edges also computes new flow equations relating nodes to a loop header. We indicate this with new edges.

depth-first traversal of the flow graph. In practice, we can decrease the number of equation evaluations by using the techniques of Horwitz, Demers, and Teitelbaum [46]. However, their techniques require a control-flow analysis, more complicated than a simple depth-first walk of the graph and similar to that used in elimination algorithms.

2.2.2 Elimination: Exploiting Flow Graph Structure

Elimination methods [3, 39, 76, 82, 86] use the structure of the CFG to solve DFA. They have two chief advantages.

First, they can be faster than iteration. The improvement in analysis time depends on how the time spent discovering the program structure (control-flow analysis) compares with the time spent evaluating the flow equations. If the flow functions are simple as in most bit-vector DFA problems, then equation evaluation is fast when compared with control-flow analysis. In that case, Hecht and Ullman has shown that elimination has roughly the same performance as iteration [43]. On the other hand, if the flow functions are more complex as in DFA problems involving symbolic analysis then the advantage swings in favor of elimination methods (assuming that finding the data-flow effect of a loop can be done quickly).

Second and more importantly, elimination methods discover program structure. This information is useful in detecting and transforming recurrences, and in solving DFA problems without finite descending chains.

Removing back-edges of loops is the key idea behind elimination methods. If a flow graph has no loops then just one iteration of the graph that evaluates the flow equations in

reverse post-order will solve the data-flow problem. If a flow graph has loops, a flow equation can depend on a variable whose value may change when evaluating another equation later in the order. In Figure 2-1(A), the nodes are numbered in reverse post-order, and the input variable $I(3)$ of node 3 depends on the output variable $O(4)$ of node 4. If $O(4)$ changes then another iteration of the graph is required to recompute $I(3)$.

Elimination removes back-edges by rewriting the flow equations and replacing the flow functions of nodes—of course, without changing the final solution to the DFA problem. For example, in Figure 2-1(C), the back-edge from node 4 to node 3 is removed by rewriting the flow equation for $I(3)$ to depend only on $O(2)$; this will require computing a new flow function for node 3, a flow function that reflects the data-flow effects of the loop formed by node 3 and 4. In our example the new flow function f'_3 is defined together with the flow equation for $O(3)$:

$$O(3) = f'_3(I(2)) = f_3(I(2)) \wedge f_3 \cdot f_4 \cdot f_3(I(2)) \wedge f_3 \cdot f_4 \cdot f_3 \cdot f_4 \cdot f_3(I(2)) \wedge \dots$$

We can also write the function as:

$$f'_3 = f_3 \wedge f_3 f_4 f_3 \wedge f_3 f_4 f_3 f_4 f_3 \wedge \dots$$

To rewrite equations, the data-flow analyzer must know how to combine flow functions in the following ways:

- $h = fg$: The composition of two flow function.
- $h = f \wedge g$: The meet of two flow functions, defined in the obvious way:

$$h(x) = f(x) \wedge g(x) \text{ for all } x$$

- $h = f^*$: The *fixed point* of a flow function, defined in the following way:

$$h(x) = x \wedge f(x) \wedge f(f(x)) \wedge \dots \text{ for all } x.$$

Using the rules above, we can write $f'_3 = f_3 (f_4 f_3)^*$ if the DFA problem is distributive.

Elimination repeatedly applies back-edge elimination on the reducible graph until it is acyclic. Figure 2-1, for example, shows such a repeated application. In part (B) and (C) the innermost back-edge is removed, and in (D) the next outer back-edge is removed. A side-effect of back-edge removal is that it computes new flow function relating nodes to a loop header.

Elimination methods have two disadvantages. First, because they manipulate flow functions, their implementation are more complex than iteration. Second, practical elimination algorithms operate only on reducible flow graphs. Although there are elimination methods that work on all graphs, they are even more complex to implement. Existing compilers deal with irreducibilities either by node-splitting which enlarges the graph, or by isolating the irreducibility in a small subgraph to which iteration is applied [2].

2.2.3 Sparse Data-flow Evaluation Graphs: Skipping Regions

Sparse DFA problems—those in which many of the flow functions are identity functions—can be solved efficiently with sparse data-flow evaluation graphs (SDFEG) [19], a generalization of the static single assignment representation (SSA) [25]. Sparse DFA problems occur, for example, when we wish to consider only a subset of a procedure’s variables: for example, when trying to find the reaching definitions for a single variable as in Section 2.1.1. Partitionable DFA problems [19] can be partitioned into a set of sparse DFA problems: for example, SSA can be separated into a set of sparse DFA problems, one for each variable. Sparse techniques—such as SSA, its extensions, and their applications—have been studied extensively recently by many authors: Alpern et al [8], Ballance et al [12], Cytron et al [25, 26], Ferrante et al [29], and Wegman and Zadeck [89].

Solving a DFA problem using a SDFEG is efficient because a SDFEG contains a subset of nodes in the original flow graph, those whose flow functions are not identity, and those at which flow values—generated at distinct nodes—meet. The edges of the SDFEG connect only these nodes and represent identity paths within the original flow graph, paths that have no effect on the flow values. Instead of iterating on the entire flow graph, the same DFA problem is solved by iterating on the smaller SDFEG. Figure 2-2 shows two flow graphs and their corresponding SDFEGs.

Besides being smaller than the original flow graph, a SDFEG has fewer variables. Every flow graph node u not in the SDFEG lies on an identity path, hence is connected to at least one node v in the SDFEG via an identity path. Therefore the variables $I(u)$ and $O(u)$ will equal $O(v)$. Thus the variables $I(u)$ and $O(u)$ can be eliminated. In Figure 2-2 the flow variables of B have the same value as the output flow variable of A.

SDFEGs overcome a problem shared by both iteration and elimination: both maintain data-flow information at nodes that do not change nor have any interest in the information. For example, in the reaching definitions problem of Section 2.1.1, nodes that do not reference x still have flow variables containing reaching definitions. Instead, sparse techniques eliminates these flow variables by recognizing that they are equal to another flow variable in the SDFEG.

In Figure 2-3, iteration would propagate a flow value modified at node 2 through nodes 3, 4, 5, 6, and 7 even if their flow functions are all identity. SDFEG skips over those nodes and relate I_8 directly to O_2 . In addition, because

$$I_3 = O_3 = I_4 = O_4 = \dots = I_7 = O_7 = O_2,$$

nodes 3, 4, 5, 6, and 7 do not need flow variables at all but instead can refer to O_2 directly. Using these equivalences to eliminate flow variables could save a substantial amount of space when solving sparse DFAs.

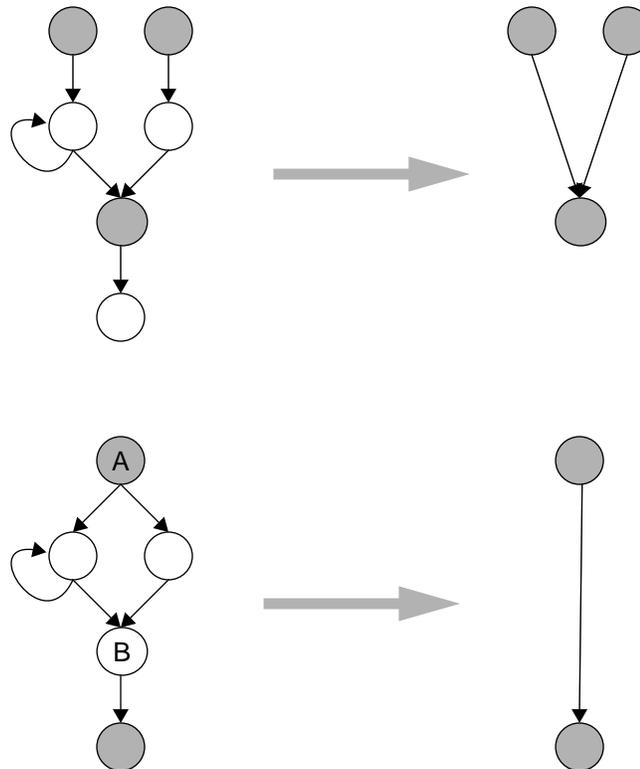


Figure 2-2 Two flow graphs and their SDFEGs

The flow graphs are on the left; their corresponding SDFEGs are on the right. In all the graphs the shaded nodes have non-identity flow functions. In the lower flow graph, the SDFEG does not contain the meet node because the flow values reaching the meet node are the same, as they are both generated at the same node. SDFEG eliminates the input and output flow variables at the node B because they will have the same value as the output flow variable of node A.

2.3 Deficiencies of Existing Data-flow Analyzers

Because data-flow analysis consumes a lot of time and space, compiler writers are concerned with implementing the analyzers efficiently. This concern has led to two deficiencies in existing optimizers. First, many compiler writers lower time and space requirements by limiting the kinds of DFA they compute. Unfortunately, such limits restrict the kinds of code optimizations that can be done, a restriction contrary to our goal of building an extensible optimizer. Second, optimizers are designed to use control-flow graph with two levels. Each node of the graph is a basic block. The two-level graph reduces the number of nodes and flow variables (not every instruction will need the flow variables I and O) required during DFA. Unfortunately, the two-level graph obscures data-flow analysis and code transformation.

The major drawback of the two-level approach is that it complicates the programming of DFAs. Each analyzer has a local analysis and a propagation phase. Local analysis com-

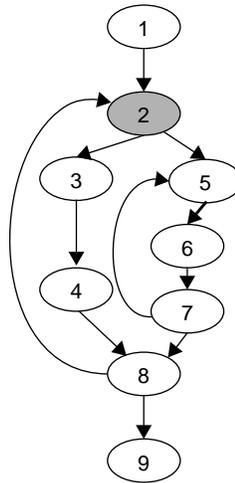


Figure 2-3 Skipping over identity nodes

In the above graph, the only node that is not identity is node 2. If the information is needed at node 8, an SDFEG sends the information directly there without having to propagate information through nodes 3, 4, 5, 6, and 7.

puts the flow function of a basic block by composing the flow function of its instructions. The propagator uses the global DFA solution at each basic block to compute the local solution at each of its instruction. These two phases are custom built by the compiler writer.

A second drawback is that compiler writers must understand the DFA at two levels before they can build prototype DFAs. The two levels appear in most published descriptions of DFAs [2, 20, 42]. Given that any DFA can be understood and solved knowing only the data-flow effect of each instruction, the two-level structure is an implementation detail and constitutes unnecessary concepts.

A third drawback is that the two-level structure encourages non-modularity. To minimize the number of CFG traversals, compiler writers often combine the local analysis of several DFA problems in one traversal, and the corresponding propagators in another traversal. Hence, the compiler writer must shoulder the burden of orchestrating these phases so they will perform as desired.

2.4 Automatic Generation of DFAs

Instead of limiting the analyses or using basic blocks, this dissertation takes a different approach to efficient DFA. Data-flow analysis plays a central role in implementing optimizations as Kildal suggested. We provide a tool, Sharlit, and an associated framework with which to construct data-flow analyzers.

In this framework, compiler writers can describe DFA problems in a modular fashion. Data-flow effects of instructions can be described with flow functions and flow values with-

out details of how DFA problems. Compiler writers can pose their DFA problems as if each node of the CFG is an instruction. The details required to solve the problems are described separately, with path simplification rules. From the two descriptions, Sharlit generates efficient data-flow analyzers. The analyzers can use all three DFA solution techniques described in this chapter. The analyzers can be combined to form more complex analyzers. The key concept to these capabilities is path simplification.

3

Path Simplification

The crux of my thesis is that optimizers will be easier to build if they are built using a framework espoused in the previous chapter, a framework in which optimizations are performed by automatically generated data-flow analyzers. The success of such an approach depends on how general and how efficient the automatically generated data-flow analyzers are. This chapter shows that analyzers fulfilling these requirements can be generated.

Sharlit’s analyzers can solve a wide range of program-analysis problems, from the traditional bit-vector based DFAs to ones using sophisticated symbolic analysis. The analyzers use one general algorithm—*path simplification*—to solve these DFAs. This algorithm is based on Tarjan’s fast path algorithm [82]. Path simplification extends these previous algorithms by integrating iteration, elimination and sparse data-flow evaluation techniques into one algorithm.

Compiler writers can pose their DFA problems directly on a CFG in which each node is an instruction. Such graphs have many flow equations and flow variables. To solve DFA problems on such graphs efficiently, Sharlit’s data-flow analyzers use path simplification to reduce the number of flow equations and nodes in the graph, then solve the equations to arrive at the solution for the original CFG.

Before we begin, we should emphasize that the compiler writer specifies a DFA problem to Sharlit by supplying the following information:

- An implementation of the flow value.
- A set F of flow functions, one function for each kind of instruction.
- A set of path simplification rules that describe how to compose and how to join certain pairs of flow functions in F , and how to take the Kleene-star of some of the flow functions in F .

This chapter is concerned with path simplification, the process by which automatically generated analyzers (also called solvers) reduce the size of graph. It is important to remember that it is the generated analyzer that simplify paths, not Sharlit. The next chapter will give examples of Sharlit specifications.

3.1 Tarjan’s Fast Path Algorithm

Sharlit’s generated data-flow analyzers are based on Tarjan’s fast path algorithm (TFPA). We use this algorithm because it can be customized to solve different DFA problems, and because it is efficient. In this section we give an outline of TFPA.

Tarjan’s TFPA algorithm solves the *single-source path problem (SSPP)* [82]. In an accompanying paper [83], Tarjan recognized that the SSPP is a generalization of many graph problems, one of which is data-flow analysis. SSPP finds a *path expression* representing all paths from the source to every node in a graph. By convention a path from node u to node v starts at the exit of node u and ends at the exit of node v . Path expressions are regular expressions built from node labels and the operators: \cdot , $+$, and $*$ (See Section 1.1—need a page number reference here).

We can turn any SSPP algorithm into a DFA algorithm by interpreting paths expressions as data-flow effects. This interpretation would map node labels to flow functions of the DFA problem, and it would map path concatenation(\cdot) to the composition of flow functions, path addition($+$) to taking the *meet* of flow functions, and the Kleene-star ($*$) to finding the flow function for a loop. Using the mappings, an SSPP algorithm would construct flow functions that represent the data-flow effects of paths in the flow graph. Therefore, solving SSPP is equivalent to finding flow functions from the source to the exit of each node. Evaluating the flow functions yields the solution to the DFA problem.

TFPA uses the loop structure of a reducible graph to solve SSPP. It is similar to elimination algorithms such as Graham and Wegman [39], and Ullman [86]. Like their algorithm, TFPA finds the natural loops [2] of a program, then removes the back-edges of the loops from inner-most loop out by replacing edges. The key steps in the algorithm are:

1. *Control-flow analysis (CFA)*: Determine the loop structure of the flow graph. The information computed lets later steps determine which loops are the innermost, which loop a node belongs to, and which nodes are the headers of loops.
2. Apply the following until the graph is acyclic:

Simplify-Reduce: For each node u of an *innermost* loop,

 - Compute a path expression that represents the paths from the header h (the entry node of the loop) to u .

- Replace u with a new node u' which has h as its sole predecessor. It is important to note that all of u 's edges is replaced with exactly one edge. This fact is key in representing the reduced graph efficiently.
 - Label u' with the path expression from h to u .
3. *Solve*: For each node u , follow the path in the reduced graph from the source to u , concatenating the path expressions that label nodes along the path. The path is unique and the computed path expression is the path expression from the source to u . By the time TFPA performs the third step, the reduced graph is acyclic, in fact a tree because the edges of u has been replaced by one edge to its header, also its parent in the tree. The tree-ness of the reduced graph means that during *Solve*, TFPA can save work by factoring out the work—that is, the path expression for a node u need only be computed once, instead of once for every child of u .

The example below demonstrates each of the steps of TFPA.

3.1.1 Efficient Implementation of TFPA

In the *Simplify-Reduce* step of above algorithm, there are many ways to implement edge replacement. TFPA is notable because of the efficient and elegant way in which it replaces edges.

A naive implementation that explicitly updates the graph would be slow. Instead, TFPA and path simplification use an auxiliary data-structure, the *header forest*, to record the sequence of graphs reductions and computed path expressions. The header forest represents the reduced graph efficiently by exploiting the fact that *Simplify-Reduce* replaces all of a node's edges with exactly one edge—the step *Simplify-Reduce* turns the innermost loop body into a tree. Initially the header forest contains all the nodes in the flow graph and no edges. When TFPA has computed a path expression p from a header h to a node u in its region, TFPA adds an edge linking h to u , and relabels u with the path p . After an application of *Simplify-Reduce* we can think of the reduced graph as a combination of the header forest and the original graph. Section 3.1.2 demonstrates this viewpoint.

3.1.2 Example: Applying TFPA

This example shows the header forest as TFPA is applied to the graph in Figure 3-1. The first step CFA finds the loop structure and the interval headers, which for this graph are A and D. Also the source node is considered the header of an *outermost interval*, as though a back-edge connects the final node S to the source. Starting at the deepest interval, which is headed by D, TFPA computes the path for E, which is simply the expression E. This is recorded in the header forest in Figure 3-2.

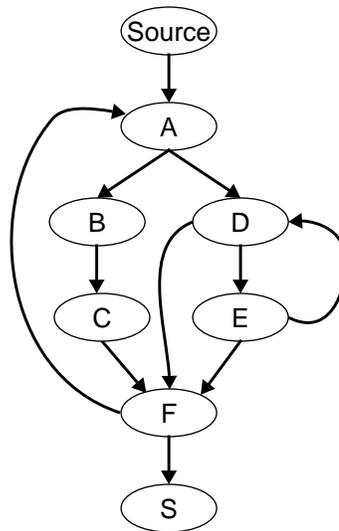


Figure 3-1 A simple graph

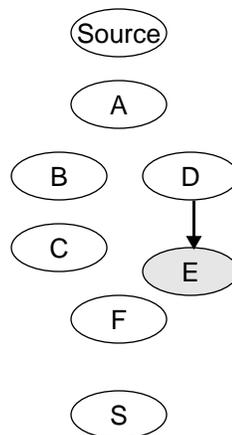


Figure 3-2 Header forest after loop D

The next loop to be processed has A as its header. The path from A to B is B and from A to C is BC. When we come to D, we must account for the back-edge from E, which gives us the path $D(ED)^*$. Once the paths through the back-edge have been subsumed in the path expression computed for D, the path expression for E can be computed and is $D(ED)^*E$. These path expressions are recorded in the header forest shown in Figure 3-3. Figure 3-3 shows also the hypothetical reduced graph composed from the header forest and the original graph—shaded nodes indicate nodes that have been replaced and that corresponds to

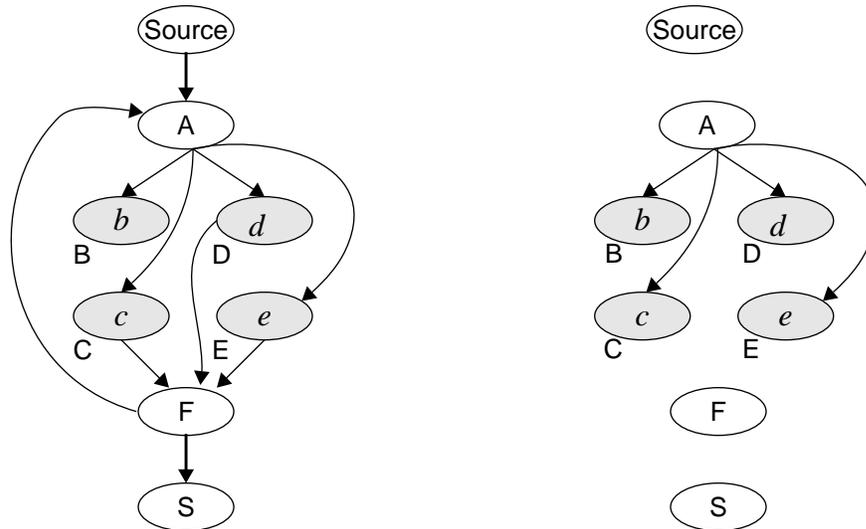


Figure 3-3 Graph and header forest after processing E within loop A

The shaded nodes in the graph (left) corresponds to the shaded nodes in the header forest (right). The new node label b is the path expression B ; c is BC ; d is $D(ED)^*$; e is $D(ED)^*E$. We leave the old labels nearby, just under the new nodes, to remind us of the original nodes.

shaded nodes in the header forest. The header forest after processing F is shown in Figure 3-4. A final application of *Simplify* gives the header forest in Figure 3-5. We can now implement the step *Solve* by computing path expressions for each node in a top-down traversal of the header tree.

3.1.3 Path Compression

When processing the region headed by the source, *Simplify* does not find path expressions for nodes B, C, D, and E, but does for node F. Nodes B, C, D, and E are left out because, once a node is already in the header forest, *Simplify* considers the node again only on demand—that is only if its path expression is required to compute the path expressions of other nodes. In the case of node F, its path expressions is computed because it is necessary in computing the path expression of the node S. Tarjan [82] calls this on-demand computation *path compression*. He shows that path compression is what gives TFPA its $O(N \log N)$ complexity where N is the size of the flow graph.

3.2 Path Simplification: An Extension of TFPA

As stated earlier in the chapter, we can use TFPA to solve DFA problems by interpreting path expressions as data-flow effects. That correspondence between path expressions and flow

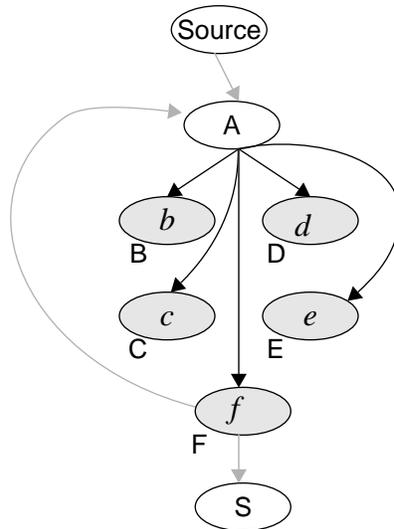


Figure 3-4 Header forest after processing F
 The label f is $(BC+D(ED)^*+D(ED)^*E)F$.

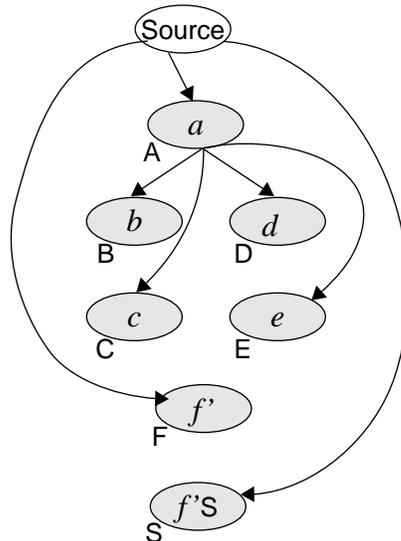


Figure 3-5 Header tree after processing outermost region
 The label a is $A(fA)^*$ and f' is $A(fA)^*f$ where f is from Figure 3-4. With this header forest, we can solve SSPP by walking the tree and concatenating path expressions or node labels.

functions is exact when the set of flow functions F is complete. We call F *complete* if combinations of functions in F satisfies the following conditions:

- the composition of any two flow functions in F is also a member of F .
- the join of any two flow function in F is also a member of F .
- the Kleene-star of a flow function is also a member of F .

Completeness guarantees that *every* path expression computed during TFPA can be interpreted as a flow function. When the set F is incomplete, a path expression may not have a corresponding flow function. In Sharlit, the compiler writer defines the set F and describes how its functions are combined. Therefore, incompleteness can occur when the compiler writer does not provide one of the combinations.

Path simplification computes only path expressions that can be interpreted as flow functions. When the set of flow functions is complete, path simplification behaves just as TFPA would—both will find expressions for paths between a node and one of its loop header. When the set is incomplete, we can no longer use TFPA but path simplification still works. But now, it finds only representable path expressions between a node and one of its dominators.

It is this ability that gives path simplification its versatility. Depending on how the flow functions are incomplete, path simplification behaves like a data-flow analyzer that does local analysis, that computes SDFEG, or that is identical to TFPA. Section 3.2.2 will explain how incompleteness causes these behaviors.

Incompleteness lets path simplification handle irreducible graphs. Though rare, irreducible flow graphs do occur in practice. The majority of programs have reducible flow graphs but irreducible graphs appear when we solve backward data-flow problems—the reverse of a reducible graph is not necessarily reducible.

Path simplification splits the *Simplify-Reduce* step of TFPA into two steps: simplify and reduce. Figure 3-6 shows how these two steps relate to the other steps of a data-flow analyzer generated by Sharlit. In following sections, we discuss each step in detail. The steps of path simplification compute and use the following information about each node.

- `ancestor[u]`: A pointer to the node that is the *ancestor* of node u in the header tree. Initially `ancestor[u]` contains the nil pointer, indicating that it does not point to any node.
- `flow[u]`: The flow function of the node u . The compiler writer provides this node to flow function mapping (see the next chapter).
- `nf[u]`: The flow function computed during reduce step. It is the flow function from `ancestor[u]` to u which is the flow function used to relabel nodes in the header tree of Section 3.1.2. Initially `nf[u]` equals `flow[u]` for all u .

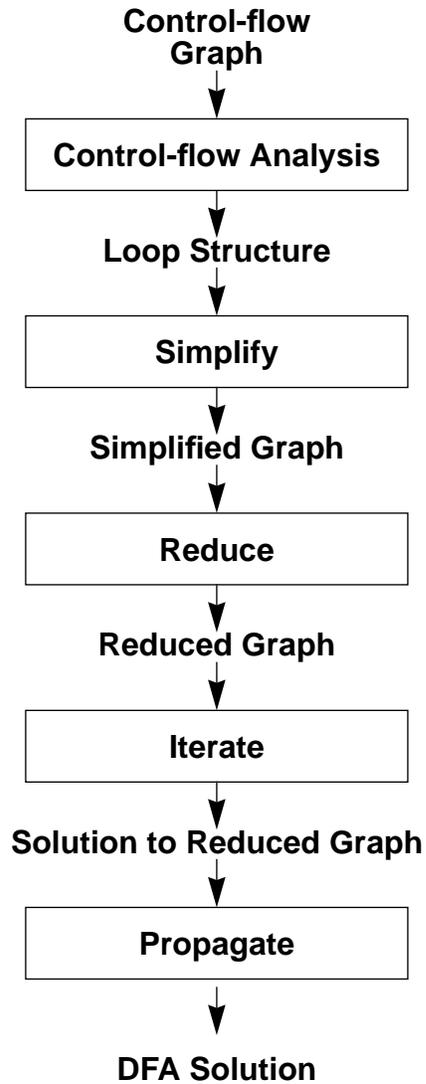


Figure 3-6 Outline of how Sharlit solves a data-flow problem

- `keep[u]`: A flag to indicate that the node should be kept during reduce step. Initially the flag is false for all u .
- `solved[u]`: A flag to indicate that iteration has found the solution for the output variable of u . Initially the flag is false for all u .

3.2.1 Control-Flow Analysis

In Procedure 3-1, control-flow analysis (CFA) discovers the looping structure of flow graphs. It presents the information as a list of loop bodies ordered so that more deeply-

nested loops appear first. Each loop body is represented as a topologically-sorted list of nodes in the loop. For example, applying CFA to Figure 3-1 gives the lists of loops as

$$L_1 = \{E\}, L_2 = \{B, C, D, F\}, L_3 = \{A, S\}, \text{ and } L_4 = \{\text{Source}\}.$$

Control-flow analysis also computes a reverse post-ordering of the nodes, an ordering that will be used in iteration and propagation.

Our control-flow analysis uses an algorithm of Tarjan's [81] which can compute the above information in $O(N \log N)$ time. This algorithm traverses the graph twice. The first depth-first traversal computes the reverse post-ordering of the graph, the back-edges, and the loop headers. A second traversal uses the loop headers and back-edge information to compute the lists of nodes described above. This is the same amount of work as CFA in existing optimizers [20, 24]. For more detail on this algorithm, see Section 1.1.2.1.

3.2.2 Simplify

After control-flow analysis, the first step of path simplification computes path expressions from the innermost loops outward (See Procedure 3-2). Within each loop L , simplification visits nodes in topological order. Because the header of L is actually in an outer loop, the header is visited only after all other nodes of L .

Procedure 3-3 shows the path computation for a single node. Figure 3-7 depicts the computation of path expression for the node u whose flow function is f . Within `simplify_node`, path computations are enclosed in quotes to indicate that the computations should be interpreted as operations—compositions, joins, and Kleene-star—on flow func-

```
cfa(G) /* Control-flow analysis */
{
    Let G be the control-flow graph.
    Perform a depth-first traversal to discover the loop structure
        of G.
    The output is a list of lists [L1, ..., Lk]
        such that depth(L1) >= depth(L2) >= ... >= depth(Lk),
        and each Li is a topologically ordered set of nodes in a
        loop body
}
```

Procedure 3-1 Control-flow analysis

Control-flow analysis in Sharlit is similar to that in existing optimizer. First, a depth-first traversal finds a reverse post-ordering of the nodes—a topologically ordering of the nodes ignoring back-edges. Next this ordering is used to compute the lists of loop bodies. Procedure 3-6 uses the reverse post-ordering during iteration. This CFA step needs to be performed once for several different data-flow analyzers.

```

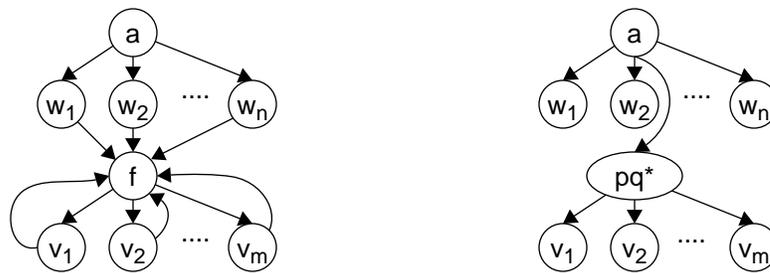
simplify(G)
{
  for(L in G as computed during cfa)
    simplify_region(L)
}

simplify_region(L)
{
  for(u in body of L)
    simplify_node(header(L), u, 0)
}

```

Procedure 3-2 Simplifying a graph

Path expressions are computed by visiting the innermost loops first. Within each loop, the nodes are visited in topological order. Note that this means the header of a loop is visited *after* the nodes in a loop.



$$p = (\text{eval}(w_1) + \text{eval}(w_2) + \dots + \text{eval}(w_n))f$$

$$q = (\text{eval}(v_1, u) + \text{eval}(v_2, u) + \dots + \text{eval}(v_m, u))f$$

Figure 3-7 Basic path expression computation in Simplify_node

We show a graphical schema of the various nodes involved in path computation and how they are combined. Here, all the incoming edges of the node u with flow function f are replaced with one edge and a new path expression.

tions. These operations are accomplished by invoking path simplification rules, which will be described in the next chapter.

The first step of `simplify_node` computes q , the path expression of loops starting at the node u and ending at the nodes v_1, v_2, \dots, v_n within loops dominated by u . Computing q requires calls to `eval` (See Procedure 3-4 on page 34) which performs path compression. The second step calls `eval` to path compress other predecessors w_1, w_2, \dots, w_m of u . The third step finds the new ancestor of u , which is a dominator of u . The final and fourth step computes the path expression from the ancestor to u .

```

simplify_node(h,u)
{
  /* Step 1. do back-edges */
  Let {v1,v2,...vn} be nodes such that(vi,u) is a back-edge,
    for i=1,...,n;
  if(n==0)
    q = "identity";
  else
    q = "(eval(q1,u)+eval(q2,u)+...+eval(qn,u)).nf[u]";

  /* Step 2. do other edges */
  Let {w1,w2,...wm} are nodes connected to u but
    (wi,u) is not a back-edge, for i=1,...,m;
  eval(wi) for i=1,...,m;

  /* Step 3. determine the ancestor of u */
  if(ancestor[w1] == ancestor[w2] == ... == ancestor[wm])
  {
    a = ancestor[w1];
    if(a==0 && m==1)/* special: unique predecessor */
      a = w1;
  } else
    a = 0;

  /* Step 3. compute new flow function */
  p = "(eval(w1,a)+eval(w2,a)+...+eval(wm,a)).nf[u]"
  p = "p.q*";
  if(p!="bottom"){
    ancestor[u]=a;
    nf[u]=p;
    return 0;
  } else {
    keep[u]=1;
    keep[vi]=1 for i=1,...,k;
    keep[wi]=1 for i=1,...,l;
    return 1;
  }
}

```

Procedure 3-3 Computing the simplified graph

The procedure, `simplify_node`, computes the path expression for a single node u . The procedure determines the ancestor of u which is a dominator of u such that the path from the ancestor to u is representable as a flow function. That flow function is kept in `nf[u]`. The other important piece of information is `keep[u]` which indicates to the reduce step that u should remain in the reduced graph.

```

eval(u)
{
  if(ancestor[u]!=nil){
    path_compress(u);
    return nf[u];
  }else
    return "identity";
}

eval(u,h)
{
  if(ancestor(u)!=nil)
    path_compress(u);
  if(u==h)
    return "identity";
  if(ancestor[u]==h)
    return nf[u];
  return bottom;
}

path_compress(u)
{
  if(ancestor[ancestor[u]]!=nil){
    path_compress(ancestor[u]);
    f = "nf[ancestor[u]].nf[u]";
    if(f!=bottom){
      nf[u]=f;
      ancestor[u]=ancestor[ancestor[u]];
    }
  }
}

```

Procedure 3-4 Header forest routines

There are two versions of `eval`. This first version computes the path expression from the root of the header tree in which `u` belongs to `u`. An auxiliary function `path_compress` does most of the path computation.

This second version of `eval` computes the same path expressions as the above version. However, it only returns the new path expression if the second argument `h` is the ancestor of `u`.

This procedure travels up the path from `u` to the root of the header tree in which it belongs. Along the way, it computes a path expression for each node `v` along the path, a path expression from the root to `v`. Because of possible simplification failures, this procedure will only compress up as far as it can. This works because functional composition (path concatenation) must be associative.

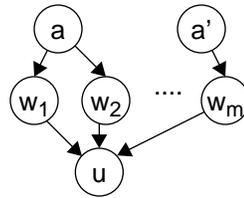


Figure 3-8 Path simplification fails

Path simplification fails a f because its predecessors have different ancestors.

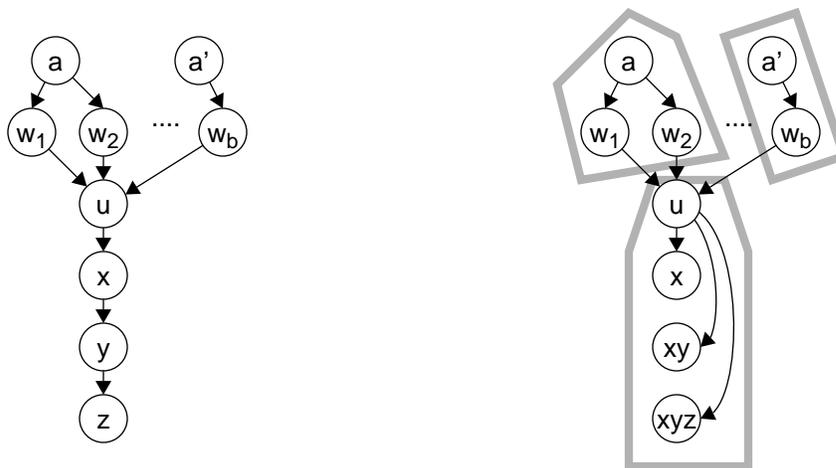


Figure 3-9 Simplification after being stopped by an irreducibility

Even after simplification is stopped at u , we can continue to find ancestors for nodes following u . The simplifier sets the ancestor of x , xy , xyz to u . The outlined subgraphs on the right indicate the nodes that form header trees in the header forest.

Path computations may fail either because the compiler writer has not given a flow function to represent some combination of flow functions, or because the graph is irreducible. In both cases, the quoted expression returns a special predefined flow function f_{\perp} , represented as “bottom” within `simplify_node`.

It is instructive to see what `simplify_node` does when confronted with an irreducibility, as in Figure 3-8. An irreducibility manifests itself when the predecessors of u do not have a common ancestor. This causes the common ancestor tests to fail in step 3 of `simplify_node`. This failure will result in `nf[u]` being set to “bottom” in step 4.

Even after a path computation fails, the procedure `simplify` continues to find path expressions, albeit the path expression may no longer be from headers to nodes. Figure 3-7 shows what happens. If path simplification was blocked at u , simplification will continue with u as the root of a new header tree. Although the node u is not a loop header, we can

```

reduce(G)
{
  Let M be an empty list of nodes;
  /* Traverse G in reverse order */
  for(u in reverse(reverse-post-order(G))) {
    if(u is a header || keep[u]) {
      M->prepend(u);
      if(ancestor[u]!=nil)
        keep[ancestor[u]]=u;
      else
        keep[v]=1 for all v, predecessor of u;
    }
  }
  return M;
}

```

Procedure 3-5 Reduce

This procedure computes the reduced graph by back-propagating the keep flag. The procedure assumes that the reduced graph contains all the loop headers. This works because in `simplify_node`, we have set the keep flag of all predecessors which are connected to `u` through back-edges.

still simplify paths leading from it to nodes that it dominates. Chapter 1 shows how incompleteness is used to make simplification perform local analysis on basic blocks and extended blocks, and build SDFEGs.

3.2.3 Header Forest Routines

Procedure 3-4 gives the routines that manipulate the header forest. The header forest is implemented through the pointers `ancestor[u]` computed during `simplify_node`. Path compression computes path expressions for paths within the header forest. This is done only on demand—only when `simplify_node` calls the two `eval` procedures.

3.2.4 Reduce

We can imagine that the procedure `simplify_node` computes a *simplified* graph. We consider it simpler because it may have fewer back-edges. It may still have back-edges because of path computation failure. The simplified graph has the same nodes as the original flow graph but its edges are defined as follows for each node `u`:

- If `ancestor[u]=a` then the simplified graph has a single edge from `a` to `u`. We say that the edges incident on `u` of the original graph has been *replaced*. The data-flow effect from the exit of `a` to the exit of `u` is stored in `nf[u]`.

```

propagate(L)
{
  for(u in L){
    if(!solved[u]){
      I=meet(O[v1],O[v2],...,O[vn])
        where v1, ..., vn are predecessors of u;
      O[u]=(flow[u])(I);
    }
  }
}

```

Procedure 3-7 Propagation

This procedure walks the graph in topological order, computing flow variables that have not previously been computed by Procedure 3-6.

- If `ancestor[u]=nil` then the simplified graph retains the edges incident on u of the original graph. The data-flow effect from the entrance to the exit of u is stored in `flow[u]`.

If the path simplification rules correctly compute compositions, joins, and Kleene-star correctly, then solving the DFA problem on the simplified graph gives the same solution as solving the original flow graph.

We can make the simplified graph even simpler by removing nodes with the procedure `reduce`, shown in Procedure 3-5. Clearly, making the reduced graph smaller will reduce the work performed during iteration. The procedure `reduce` partitions the graph into two graphs, a reduced graph and all other nodes whose DFA solutions can be computed by propagating the solutions of the nodes in the reduced graph. The reduced graph is computed in a single scan of the graph in reverse of the reverse post-order. During the scan, the procedure `reduce` back propagates the keep flag—set earlier in `simplify_node`. The procedure `reduce` does not need to make more than one scan because nodes connected to u through back-edges always follow u in the reverse post-order (see Hecht [42])—these nodes have already had their keep flag set in `simplify_node`. Not only does `reduce` determine the nodes in the returned graph, it also returns M , the reverse post-order of the reduced graph.

3.2.5 Iterate and Propagate

Iteration (Procedure 3-6) can now efficiently solve the DFA problem by iterating only on the much smaller reduced graph. Iteration follows the edges in the reduced graph during evaluation, hence the test of `ancestor[u]`. If all the back-edges are eliminated then we

```

iterate(M,limit)
{
  iter=0;
  changed=0;
  while(!changed && iter<limit){
    for(u in M){
      if(ancestor[u])
        I=O[ancestor[u]];
      else
        I=meet(O[v1],O[v2],...,O[vn])
              where v1, ..., vn are predecessors of u
      if(ancestor[u])
        V=(nf[u])(I);
      else
        V=(flow[u])(I);
      if(O[u]!=V){
        O[u]=V;
        changed=1;
      }
    }
    iter++;
  }
  solve[u]=1 for all u in M
}

```

Procedure 3-6 Solution with iteration

This procedure solves the data-flow problem on a graph stored as a list of nodes *M*. The list may be a reduced list resulting from the procedure `reduce`.

limit `iterate` to one pass by setting its second argument `limit` to 1. After finding the fixed point, `iterate` sets the solved flag of the nodes in the reduced graph.

For nodes not in the reduced graph, their flow variables are computed in the procedure `propagate` shown in Procedure 3-7. It make a pass of the graph in reverse post-order to fill in the missing flow variables.

3.3 Conclusion

Sharlit lets the compiler writer use a simple model of DFA, a model in which the nodes are the individual IR instructions, and in which data-flow information is computed and available at each instruction for optimizations. But this model has two pitfalls: too many nodes and too many flow variables.

This chapter shows that path simplification can deal with the problem of too many nodes. It builds a simplified graph by replacing edges in the original flow graph. The suc-

cess of this replacement depends on completeness. This simplified graph has the property that it can be partitioned into two sets: a reduced graph that can be solved quickly (iteration), and the rest which can be solved using one traversal of the graph (propagation). In the next chapter, we show how we can eliminate many flow variables.

It is instructive to compare Sharlit’s data-flow analyzers with data-flow analyzers in existing optimizers:

1. Control-flow analysis: Sharlit’s analyzers performs as much work as other analyzers here.
2. Simplify: The closest analog to this step in traditional analyzers is local analyzers. While there is more overhead in our reduction step—which the next chapter shows can be used to simulate local analysis—it is more general and can be used to perform other styles of data-flow analysis.
3. Reduce: traditional data-flow analyses reduce the size of the graph by collapsing linear sequences of instructions into basic blocks. Sharlit’s reduction phase is more general in that it can collapse but also removes nodes in the same way that elimination algorithms do.
4. Iterate: This step is identical in Sharlit and in traditional analyzers but Sharlit’s reduced graph can be smaller.
5. Propagate: This step is identical in both Sharlit and traditional analyzers.

4

Writing Data-flow Analyzers with Sharlit

The previous chapter showed how path simplification eliminates nodes and flow equations. Path simplification lets us solve DFA problem D by solving an equivalent DFA problem D' on a reduced control-flow graph, and computing the solution of the solution of D from the solution of D' . This general DFA solution procedure subsumes not only the traditional method of basic blocks, but also extended basic blocks, interval analysis and sparse data-flow evaluation graphs. This chapter shows how these four kinds of data-flow analyzers can be programmed with Sharlit. In the process, these examples explain the following:

- *Flow variable elimination:* Although flow equations have been eliminated, every instruction still has two flow variables. Flow values could occupy large amounts of storage. We exploit the reverse post-ordering to show that many flow variables—in particular those associated with instructions in what would have been basic blocks—can be computed on-the-fly and does not need to be stored. We also eliminate redundant flow variables that arise when nodes are connected by identity paths (See Section 2.2.3)
- *Path simplification rules:* Path simplification combines flow functions as it constructs path expressions. However, the last chapter does not show how the compiler-writer can affect the simplification process. This chapter demonstrate how simplification is controlled with path simplification rules.
- *Incomplete path simplification:* As explained in the previous chapter, irreducibility results in some paths not being simplified. Another way in which incomplete simplification can arise is with *incomplete* set of rules. With a complete set of rules, all possible combinations of flow functions can be represented as another flow function

during simplification. An incomplete set may mean that a path may failed to be simplified because it contains a combination of flow functions which cannot be combined by any rule. We demonstrate incomplete rule sets by showing how to compute SDFEG as described in Chapter 2 and in Choi[19].

- *Extensibility*: Sharlit generates analyzers that are extensible. We demonstrate Sharlit with three examples, all solving the *available expression problem* (AVP)[2]. An expression is available at a node u if every path from the source to u has evaluated the expression and the value of the expression remains the same if reevaluated at u . Determining available expressions is important in global common subexpression elimination and code motion [20, 50, 51, 65].

We built each example from the previous by adding new flow functions and new path simplification rules. The first example is a prototype solver that uses iteration to solve the AVP. In this first example, each IR instruction is a node in the flow graph, making the prototype inefficient. By adding some simplification rules, we extend the prototype to do local analysis, reducing the work needed during iteration. By adding yet more rules—meet and the star rules—we enhance the solver to use interval analysis.

In the following we assume the reader has some familiarity with C++[79]. Sharlit is implemented in C++ and uses C++ in the same way that YACC uses C.

4.1 Preliminaries

An optimizer built with Sharlit consists of one or more data-flow analyzers, as shown in Figure 4-1. The analyzers may execute in series, running one after another on the input program. Or they can be combined as described in the Chapter 1 so that they cooperate over the input program.

Figure 4-2 shows that each automatically generated analyzer is generated by Sharlit and has four parts: control-flow analyzer, path simplifier, iterator, and propagator. The control-flow analyzer, which is done once for several DFAs, discovers the loop structure of the control-flow graph. The path simplifier consists of a simplifier and a reducer, both described in the last chapter. The iterator makes one or more phases over the reduced graph, calling flow functions until a solution to the data-flow problem is found. The propagator computes the solution at eliminated nodes from the solution at nodes in the reduced graph. In the process, the propagator calls user-supplied action routines for each node, routines that use the solution to perform optimizations.

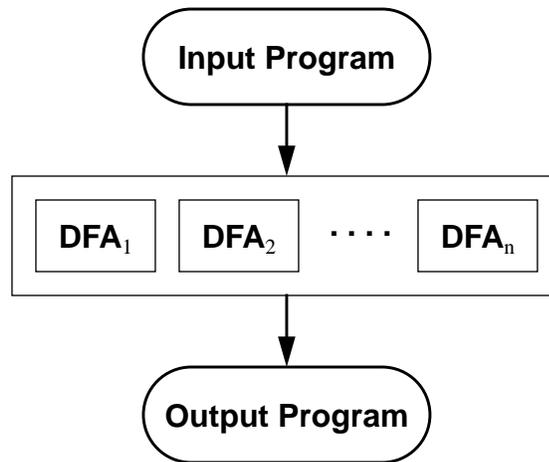


Figure 4-1 A phase of the an optimizer

Each phase of the optimizer consists of one or more data-flow analyzers generated with Sharlit.

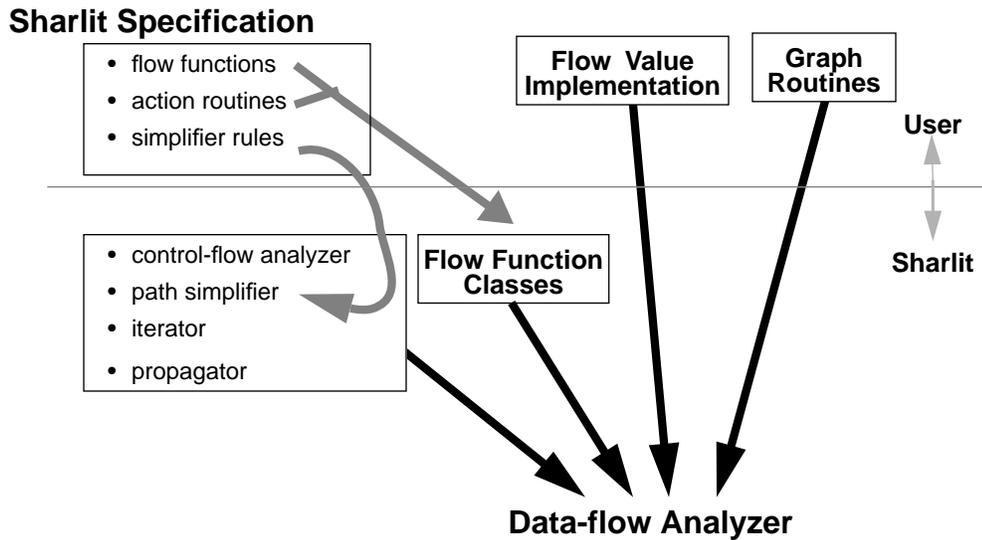


Figure 4-2 Structure of a data-flow analyzer

Data-flow analyzers are automatically generated by Sharlit. The dotted line separates what must be provided by the compiler writer, from what is generated or provided by Sharlit.

All the components are compiled and linked (black arrows) into an internal phase of the optimizer. The flow value is implemented as a C++ class by the compiler writer. The compiler writer can implement the flow graph in any way, but must provide graph routines that let the generated data-flow analyzer *view* the flow graph as a doubly-linked directed graph with edges stored as adjacency vectors.

In our examples, the IR is quad based and each IR instruction is implemented using the following structure:

```
class IR_instruction {
    ...
public:
    int kind;
    IR_variable dst,src1,src2;
    Expression expr_no;
    ...
};
```

Only those fields relevant to our examples are shown.

The IR has several types of instructions, distinguished by the field `kind`. Since expressions are built recursively from arithmetic operations and variables, we are interested in the following kinds:

- `IR_op`: performs an arithmetic operation on `src1` and `src2`, storing the result into the variable `dst`.
- `IR_ldc`: loads a constant into the variable `dst`.
- `IR_kill`: modifies the variable `dst`.
- `IR_initial`: is the instruction type used to mark the source node of the graph.
- `IR_others`: doesn't affect the contents of any of the variables.

We assume that value numbering [2, 20] has been performed—possibly in a previous phase—on the instructions. For each IR instruction, value numbering has set the field `expr_no` to identify the symbolic expression that is being computed by the instruction. For each variable, value numbering has computed the set of expressions *killed* by a write to the variable (See `kill_sets` below).

4.2 Example of Iterative Analysis

This example shows a Sharlit specification that uses iteration to solve AVP. It is the simplest kind of specification possible, and contains the minimal ingredients necessary for a working DFA solver.

The braces in the specification delimit C++ code, which Sharlit inserts into a C++ code framework to build a solver, just as YACC [48] inserts user-supplied C code into a C code framework to build a parser. Within the C++ parts, variables that begin with an underscore

```

iterate(M,limit)
{
  iter=0;
  changed=0;
  _V = new_value();
  while(!changed && iter<limit){
    for(u in M){
      if(ancestor[u])
        copy_value(_V, O[ancestor[u]]);
      else if(has_meet[u])
        copy_value(_V, meet(O[v1],O[v2],...,O[vn]))
          where v1, ..., vn are predecessors of u;
      if(ancestor[u])
        (nf[u])(_V);
      else
        (flow[u])(_V);
      if(has_var[u])
        changed=changed|copy_value(O[u],_V);
    }
    iter++;
  }
  solve[u]=1 for all u in M
}

```

Procedure 4-1 Solution by iteration

This procedure is a refinement of Procedure 3-6 in Chapter 3. The previous procedure assumes that each node u has an output flow variable $O[u]$, while this procedure handles eliminated flow variables. A node u has a variable if and only if $has_var[u]$ is set. A node u requires a meet operation to compute its input flow value if and only if $has_meet[u]$ is set. The later flag helps iterate avoid unnecessary meets.

The procedure below calls three procedures given in the Sharlit specification. The procedure `new_value` creates a new flow value and returns a pointer to it. The procedure `copy_value` copies its second argument (`_SRC` in the specification) to its first argument (`_DST`). The procedure `meet` assigns to its first argument (`_DST`) the meet of its second argument, a list of flow values (`_SRCS`).

To compute data-flow effects, the procedure below calls either `nf[u]` or `flow[u]`. They expect that `_V` points to the input flow value. They modify this flow value to become the output flow value.

are special. Playing the same role as the $\$$ variables of YACC, they refer to variables with which the Sharlit generated parts of the solver communicate with the user-supplied code.

Figure 4-3 shows a Sharlit specification. The major parts of the specifications are described below:

- Data specific to the DFA problem

In Figure 4-3(A), the keyword `solver` begins a declaration for variables that are specific to DFA problem. These variables will be used by user-supplied code in

```

solver: {
    Expr_set *kill_sets[];
};
node_base = IR_instruction;
value_base = Expr_set;
new_value: {
    Expr_set *v=new Expr_set(etable);
    v->mk_universal();
    return v;
};
copy_value: {
    if(*_SRC == *_DST)
        return 0;
    *_DST = *_SRC;
    return 1;
};
meet: {
    int i;
    _DST->mk_universal();
    for(i=0;i<_NSRCS;i++)
        *_DST &= *_SRCS[i];
};
AV_expression: flow {
    *_V += _N->expr_no;
    *_V -= _P->kill_sets[_N->dst];
};
AV_initial: flow {
    _V->mk_empty();
};
AV_kill: flow {
    *_V -= _P->kill_sets[_N->dst];
};
flow_map: {
    switch(*_N->kind){
    case IR_op:
    case IR_ldc:
        return new AV_expression;
    case IR_kill:
        return new AV_kill;
    case IR_initial:
        return new AV_initial;
    case IR_others:
        return IDENTITY;
    }
};
end;

```

Figure 4-3 Using iteration to solve for available expressions

other parts of the Sharlit specification. For example, the flow functions refer to the kill information with the expression `_P->kill_sets`.

- Flow graph nodes

In Figure 4-3(A), the `node_base` line tells Sharlit that each node in the flow graph is an object of type `IR_instruction`. Sharlit doesn't need to know the internal details of the type to generate an analyzer but it does need the name of the type to declare variables.

- Flow values:

The other keywords in Figure 4-3(A)—`value_base`, `new_value`, `copy_value`, and `meet`—tell Sharlit about the type of flow values, and the operations on flow values, operations called in the process of solving the DFA problem. The `value_base` line indicate to Sharlit the name of the type with flow values should be declared. The other keywords define operations. To allocate a new flow value, the DFA solver calls the C++ code following `new_value`. To copy a flow value (`*_SRC`) into another (`*_DST`), the solver calls the code following `copy_value`. To take the meet of a set of flow values—pointed to by an array of pointers `_SRCS`—the solver calls the code following `meet`.

The `copy_value` operation has another important duty—it tells the solver whether the source flow value (`*_SRC`) was equal to the destination flow value (`*_DST`) before copying. Returning an indication of whether the destination changed with the copy is used as a termination test within the solver.

- Flow functions:

Figure 4-3(B) shows the flow functions. In each of the flow functions, the incoming flow value is pointed to by the variable `_V`. The code following the keyword `flow` computes an output flow value, leaving a pointer to it in `_V`. In available expressions, and many other DFAs, the code can modify the incoming value directly into the outgoing value. For large, complex values, modification-in-place avoids unnecessary copies.

There is a flow function corresponding to each instruction type discussed above.

- `AV_expression` corresponds to `IR_ldc` and `IR_op` instructions. This function inserts `expr_no`, the expression number being generated by the current node `_N`, into the incoming flow value, then removes the killed expressions, those whose value may change when the variable `_N->dst` is modified.
- `AV_kill` corresponds to the instruction `IR_kill`. This function removes all the killed expressions from the flow value.

```

reduce(G)
{
  Let M be an empty list of nodes;
  /* Traverse G in reverse order */
  for(u in reverse(reverse-post-order(G))) {
    if(nf[u]!="identity"){
      if(has_meet[u])
        has_var[v]=1 for all v, predecessor of u;
      if(u is a header)
        has_var[u]=1;
    }
    if(u is a header || keep[u]){
      M->prepend(u);
      if(ancestor[u]!=nil)
        keep[ancestor[u]]=u;
      else
        keep[v]=1 for all v, predecessor of u;
    }
  }
  return M;
}

```

Procedure 4-2 Reduce revisited

In this version of reduce, we have added code that sets `has_var`. It also eliminated nodes that are connected to other by identity path functions.

- `AV_initial` corresponds to the initial node `IR_initial`. This function initializes its output value to the empty set.
- `AV_identity` corresponds to the `IR_others` instruction. This function uses its incoming flow value as its outgoing value.
- Flow map:
 - We must provide the above mapping between instructions and flow functions.
 - Part C is a flow map function called from control-flow analysis to establish the correspondence between flow functions and instructions.

Because the specification in Figure 4-3 has no simplification rules, Sharlit will generate data-flow analyzer that uses iteration for it. Procedure 4-1 on page 44 shows the iteration procedure, which differs somewhat from the one in the previous chapter because it uses fewer flow variables.

4.3 Eliminating Flow Variables

The interface between the generated analyzer and the above user code hides the flow variables ($I(u)$ and $O(u)$ of Section 2.1). In the example above, each flow function sees only

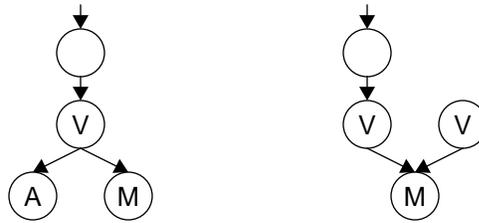


Figure 4-4 Has_meet and has_var nodes

A meet must be performed to compute the input flow value of nodes that have a predecessor which do not precede them immediately in the reverse post-order. Such nodes are marked with an M above. It is important to note that nodes such as the one labelled A do not need has_meet set because their predecessor does precede them in the reverse post-order.

Nodes that have at least one successor which is has_meet must be has_var—they must have an output flow variable. Also, because the output flow variable of an ancestor can be referenced repeated by many nodes, ancestor are also has_var.

Both has_meet and has_var is computed during control-flow analysis.

one flow value $_V$. Suppose a flow function is associated with a node u , then at the beginning of the flow function, $_V$ equals the input flow value $I(u)$. The flow function modifies $_V$ directly to reflect the data-flow effect of u . After the flow function returns, either $_V$ is passed directly to the successor node of u or $_V$ is stored in away in $O(u)$. Sharlit hides both activities from the compiler writer.

Because compiler writers have no direct access to flow variables, Sharlit can eliminate many of the flow variables to save space. This elimination is transparent to the compiler writer. During propagation, the values of the eliminated flow variables are generated and passed to the action routines, as we show later in Procedure 4-3.

Eliminating variables is demonstrated in Procedure 4-1 and Procedure 4-2, updated versions of `iterate` and `reduce` of the last chapter. Procedure 4-1 eliminates all input variables as they are computed on-the-fly while traversing the graph. To eliminate some of the output flow variables, Procedure 4-1 or `iterate` uses two flags —`has_var[u]` tells it that u has an output flow variable, and `has_meet[u]` tells it to perform a meet to compute the input flow value for u . The flag `has_meet[u]` is set during control-flow analysis at those nodes u that has at least one successor which does not immediately precede it in the reverse post-order. When `has_meet[u]` is set then u 's predecessors must be `has_var`, unless the flow function relating u to its ancestor by the identity flow function. When that happens, we can determine that

$$I[u] = O[u] = O[\text{ancestor}[u]]$$

This fact is used to generate efficient solvers for sparse data-flow evaluation graphs. Another reason that a node may be `has_var` is that it is an ancestor of another node. See Figure 4-4 for examples of node with this flags turned on.

An interesting consequence of using these flags in `iterate` is that flow variables are not necessary for nodes within linear sequences of nodes or basic blocks. Consider two nodes u and w such that u 's unique successor is w and w 's unique predecessor is u . After calling the flow function for u , `_V` contains the value of $O[u]$. But since $I[w]$ equals $O[u]$, `_V` can be passed to the flow function of w as its input, thus bypassing a copy operation from `_V` to $O[w]$, and a meet operation that copies $O[u]$ to $I[w]$. As no other flow equations need the value of $O[u]$ and $I[w]$, both variables can be eliminated.

Nodes that are ancestors of others also require output flow variables. As it is difficult to determine when and how many times the flow variable of an ancestor are used in evaluating other flow functions, we simplify matters by insisting that all ancestors have output flow variables.

4.4 Path Simplification Rules

Path simplification builds path expressions and in the process computes flow functions using composition, join, and Kleene-star of other flow functions. These operations are directed by the compiler writer with path simplification rules. The next examples use these rules to do local analysis, and to do interval analysis.

An obvious way to tell the analyzer how to combine flow functions is to use the *times table* approach. For example, when the analyzer needs to compose two flow functions, it would index a special table with the two flow functions to find what actions are necessary to do the composition. Although tables are simple, specifying them directly is tedious: A complete specification of the composition table for M flow functions would require M^2 entries. Instead Sharlit can generate a complete table using only $M + 1$ rules and a new kind of flow function: *a path function*.

During iteration, a path function behaves just like any other flow function, except that it represents the data-flow effect of a path. During path simplification, a path function is treated differently for efficiency reasons—they are modified directly or reused.

Below we show the five kinds of rules and describe how they work conceptually. Each rule has a code part that when executed computes a new path function by modifying the path function in the left-hand side of the rule.

1. `p create` Create a new path p whose data-flow effect is the identity. This rule is used in the situations depicted in Figure 4-5. The analyzer inserts identify flow functions with the intention of extending that path

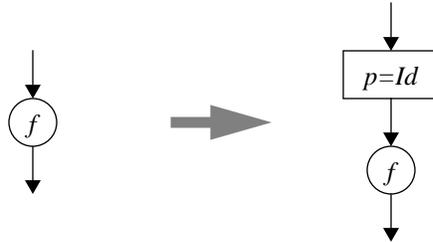


Figure 4-5 Applying the “ p create” rule

An identity path denoted by the square box is inserted into the flow graph before the node with flow function f .

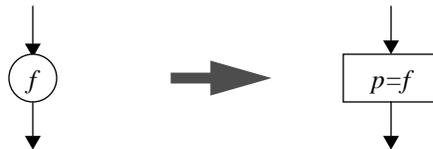


Figure 4-6 Applying the “ p create f ” rule

The second kind of create rule converts a flow function f into a path function p .

(see below). Inserting identity data-flow effects into a graph does not change the solution of the data-flow problem.

1. p create f Create a new path p whose data-flow effect is the same as the flow function f (see Figure 4-6). Again this replacement will not change the solution of the data-flow problem. This replacement is used by the analyzer to create paths in the graph that begins with nodes with flow function f .
1. p absorbs f Extend a path p so that it has the same data-flow effect as the old p composed with f , that is modify p so that it represents the data-flow effect $p \cdot f$ (Figure 4-7).
2. p star Replace the path p with a path that represents the data-flow effect of p^* . This rule, together with other rules, is used to remove back-edges (See Figure 4-8).
3. p_1 join p_2 Replace the path p_1 with a path that represents the data-flow effect of the path $p_1 + p_2$ (Figure 4-9).

These rules resemble those of T_1 - T_2 graph reductions of Ullman[86]. And Sharlit uses them in a similar fashion. For example, in Figure 4-10, we apply a sequence of rules to com-

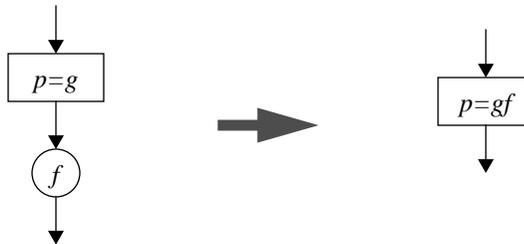


Figure 4-7 Applying the “ p absorb f ” rule

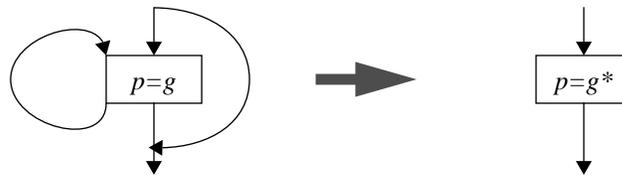


Figure 4-8 Applying the “ p star” rule

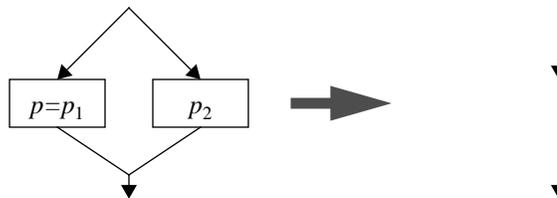


Figure 4-9 Applying the “ p_1 join p_2 ” rule

This rule only applies when the two paths share common endpoints. This condition is enforced in the path simplifier because the ancestors of the nodes must be the same (See Procedure 3-3 of Chapter 3).

pute the path expression through a loop. The path p is created once and updated directly as it absorbs and joins with other nodes. This update is possible because some of the nodes are subsequently removed by `reduce`. For example, in Figure 4-10, `reduce` eliminates the node which has flow function g , thus the path simplifier can update p in (C) directly to compute gf .

Sometimes absorption cannot be applied directly as in Figure 4-11 where the node u has two successors. To absorb g and h , the analyzer must first make a copy of p . This copy is performed with a rule of the form “ p absorbs p ”.

Relating this back to Procedure 3-3 of Chapter 3, we can see that the path computation steps are application of these rules. So computing a path expression, such as

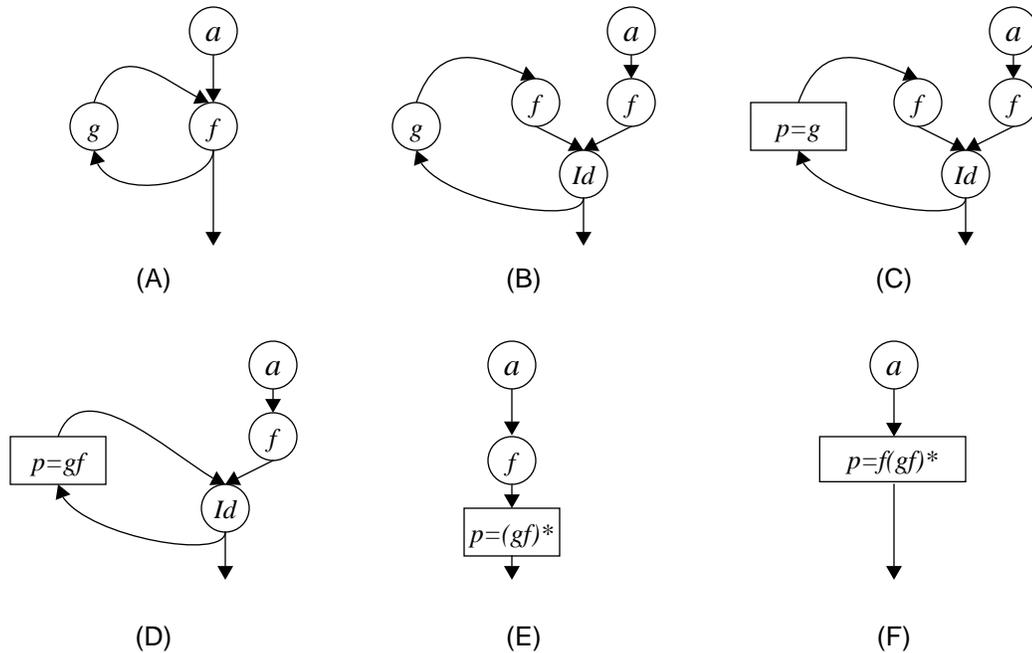


Figure 4-10 Example of applying several path simplification rules

We show conceptually how paths are computed with the path simplification rules— p create g ; p absorb f ; and p star. Part (A) above shows a subgraph with a loop. For the purposes of data-flow analysis, this subgraph is equivalent to the one in (B)—introducing an Identity (Id) doesn't change the data-flow problem. Applying p create g to (B) gives graph (C); apply p absorb f to form the composition of gf in (D); apply p star to get the fixed point of gf in (E). Finally, apply p absorb f to get the data-flow effect of the entire loop as $f(gf)^*$. This is the flow function that will be stored in the header forest for the node originally associated with f , the node whose ancestor is a .

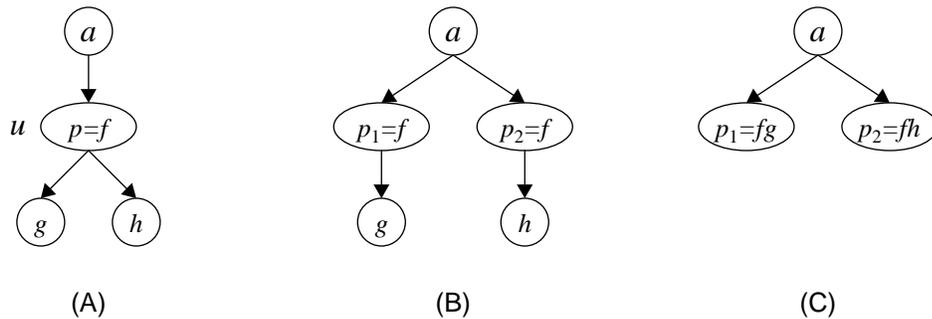


Figure 4-11 Copying paths

The node u is in the header tree and has a as its ancestor. In (B), because u has two successor, it necessary to copy u 's path function p before proceeding. After the copy, absorption can be applied to g and h as in (C).

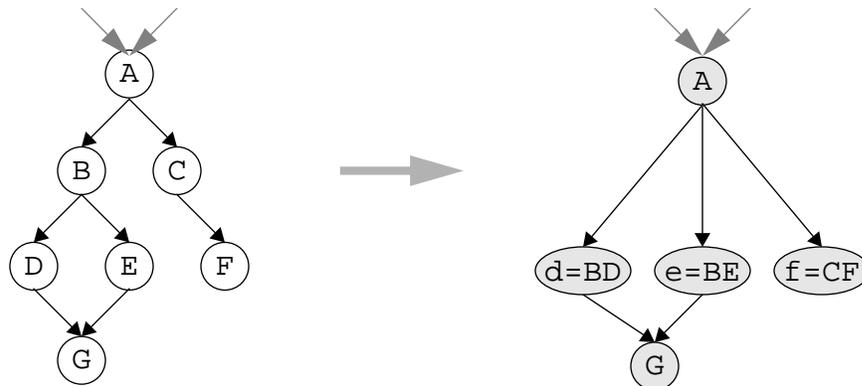


Figure 4-12 Simplifying extended basic blocks

The generated solver computes path expressions for nodes B, C, D, E, and F. But since B and C are no longer referenced they can be eliminated from the graph that is fed to the iterate procedure. Also the lack of a join rule means the path from A to G cannot be summarized, thus G will start another extended basic block.

$q \leftarrow ((q_1 + q_2 + \dots + q_a)f)$ means applying a sequence of path simplification rules. Conceptually applying the rules is the same as reducing the graph as in Figure 4-10. Of course, the graph isn't really reduced as that would be too expensive—Only the final path expression is saved in the header forest.

4.5 Local Analysis: Summarizing Extended Basic Blocks

This section shows how to make the data-flow analyzer of Section 4.2 more efficient, by adding path simplification rules. The analyzer in Section 4.2 is inefficient because during iteration it visits every IR instruction. Still, such simple analyzers can serve as prototypes for experimenting with new optimizations. When the need for efficiency arises, we can add rules to the specification, rules that reduce the number of nodes during iteration.

Existing optimizers group instructions into basic blocks to reduce the size of the flow graph, therefor reducing the amount of work during iteration. The optimizers must perform local analysis, the act of computing the data-flow effect of each basic block from its instructions. Sharlit does not explicitly represent basic blocks but achieves a similar reduction by using path simplification.

Figure 4-12 shows the computation of data-flow effects within an *extended basic blocks* or EBB [2]. An extended basic block is a tree of flow-graph nodes in which the root is the only node that may have more than one predecessor; all internal nodes of the EBB must have one predecessor. Applying `create` and `absorbs` rules, the simplifier computes the path functions that represent the data-flow effect from A to the value at the exit of the leaves: D, E, and F. Once the entrance-to-exit flow functions of an EBB have been computed, its internal nodes are no longer needed during iteration, and `reduce` removes them.

```

....
AV_bb: local {
    Expr_set defs;
    Expr_set kills;
}
path {
    _V -= kills;
    _V += defs;
};
....
simplifier
(:AV_bb) create = {};
(b:AV_bb) absorbs (l:AV_expression,N) = {
    Expr_set *kill_set= &_P->kill_sets[N->dst];
    b->kills += *kill_set;
    b->defs.add(N->expr_no);
    b->defs -= *kill_set;
};
(b:AV_bb) absorbs (l:AV_kill,N) = {
    Expr_set *kill_set= &_P->kill_sets[N->dst];
    b->kills += *kill_set;
    b->defs -= *kill_set;
};
(b:AV_bb) absorbs (l:AV_bb,S) = {
    b->defs -= l->kills;
    b->defs += l->defs;
    b->kills += l->kills;
};

```

Figure 4-13 Using local analysis to solve for available expressions

We extend the example in Section 4.2 with the code in Figure 4-13 to compute paths through EBBs. Part A of Figure 4-13 declares a path function `AV_bb`. The local part of `AV_bb` declares `defs` and `kills`, two variables computed by the simplification rules and used in the computation:

$$_V \leftarrow (_V - \text{kills}) \cup \text{defs}$$

to compute the data-flow effect of a path during iteration. Each simplified path in the flow graph has an associated `AV_bb` with its own private `defs` and `kills`.

Rules define local variables for use within the code part of rules. The absorb rule `(b:AV_bb) absorbs (l:AV_expression,N)` defines three pointer variables `b`, `l`, and `N`. When the rule is applied, the variable `b` points to the flow function of type `AV_bb`; the variable `l` to the flow function of type `AV_expression`; and the variable `N` to the node which is associated with `AV_expression`. The code part of the rule updates the

```

name: local {
    local variables
}
flow {
    ....
}
in {
    code that uses flow value at entry of node
}
out {
    code that uses flow value at exit of node
};

```

Figure 4-14 Flow function declaration with in and out functions

`defs` and `kills` fields of the path function pointed to by `b` directly to represent the composition of the path function pointed to by `b` and `l`.

Figure 4-13 has only `create` and `absorbs` rules, but not `join` rules. This is why the path simplifier only finds paths internal to EBB. When the path simplifier attempts to compute the path expression for a node with more than one predecessor, it will fail because a join rule is necessary to combine the incoming paths. The nodes with more than one predecessor are exactly the root nodes of the EBB. In Figure 4-12, for example, the path simplifier will fail to find a path expression for the node `G`. Therefore, `G` will remain in the graph and start another EBB.

After simplification, the procedure `reduce` removes the nodes internal to the EBB. The simplified graph, represented as a list of nodes, is then solved by `iterate` (Procedure 4-1). The solution consists of flow values at the output of the nodes which form the boundaries of EBB. For the solution at the internal nodes, we apply the procedure `propagate` (Procedure 4-3) which walks each EBB in topological order, computing the solution of the internal nodes from their predecessor.

As part of propagation, calls are made to `in[u]` and `out[u]`, action routines that perform optimizations based on the flow value at the input and output of a node. Action routines are provided with flow functions as shown in Figure 4-14.

4.6 Elimination: Summarizing Loops

The previous example has an incomplete set of path simplification rules. Although the rules can express the composition of any two flow function as a single path function, it cannot express the join of paths nor the fixed point of a path. This incompleteness resulted in the path simplifier breaking the flow graph into EBB and eliminating nodes within the EBBs.

```

propagate(L)
{
  _V=new_value();
  for(u in L){
    if(has_meet[u]){
      if(nf[u]== "identity")
        copy_value(_V,O[ancestor[u]]);
      else
        I=meet(O[v1],O[v2],...,O[vn])
           where v1, ..., vn are predecessors of u;
    }
    (in[u])(_V);
    if(solved[u])
      copy_value(_V,O[u]);
    else{
      (flow[u])(_V);
      if(has_var[u])
        copy_value(O[u],_V);
    }
    (out[u])(_V);
  }
}

```

Procedure 4-3 Propagation: finding solutions at eliminated nodes

The procedure *iterate* (Procedure 4-1) solves the data-flow problem for nodes retained after *reduce*. In the procedure below computes the solution—input and output flow values—for those nodes removed by *reduce*. As the flow values are computed, they are handed to special optimizer routines *in[u]* and the *out[u]*, making them available but without having to store them.

Another optimization is to detect that *nf[u]* is the identity. In that case, the *meet* can be avoided (and the flow variables at *u*'s predecessors).

We can make the rules in Figure 4-13 complete by adding the rules in Figure 4-15. These rules form the joins and the fixed points of flow functions. By completing the set of rules, we make it possible for the path simplifier to summarize loops and to remove back-edges. In that situation, the path simplifier implements TFP. In a reducible flow graph, TFP removes all back-edges and makes it possible to solve the data-flow problem without iterating. This can potentially make it quicker to solve the DFA problem, and more importantly, can make it possible to solve DFA problems which may not converge with iteration—they do not have the descending-chain condition.

4.6.1 Irreducible Graphs

Even when confronted with an irreducible graph, our example can still simplify many paths, albeit it cannot simplify cycles that include the header of an irreducible region. In Figure 4-

```

(b1: AV_bb) joins (b2: AV_bb) = {
  Expression_set sum1, sum2;
  sum1.universal(_P->etable);
  sum2.universal(_P->etable);
  sum1 -= b2->defs;
  sum1 *= b2->kills;
  sum2 -= b1->defs;
  sum2 *= b2->kills;
  sum2 += b1->kills;
  b1->kills = sum1;
  b1->kills *= sum2;
  b1->defs *= b2->defs;
};
(b:AV_bb) star = {
  b->kills -= b->defs;
  b->defs.clear();
};

```

Figure 4-15 Enhancing solver to use interval analysis

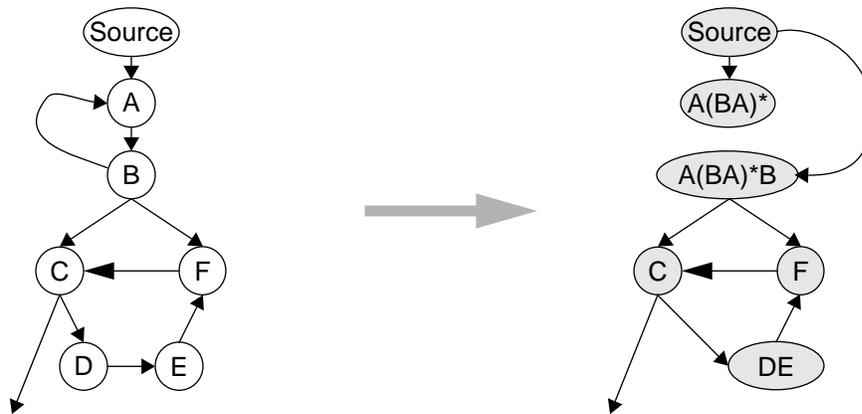


Figure 4-16 Path simplifying irreducibilities

16, for example, our path simplifier simplifies around and within the irreducible region formed by the nodes C, D, E and F, and find path expressions for nodes A, B, D, and E. The simplifier will also find path expressions for nodes following C, although those paths cannot begin at A or any other node above C.

```

...
Defs: flow { ... }
Id_path: path Identity;
...
simplifier
Id_path create = {};
Id_path absorb Identity = {};
Id_path join Id_path = {};
Id_path star = {}
end

```

Figure 4-17 Rules for doing SDFEG

Above is part of a Sharlit specification for the reaching definitions problems (preferably for a single variable). There are three flow functions `Defs`, `Id_path` and the predefined `Identity`. The flow function `Defs` corresponds to those nodes that generate definitions, while `Identity` are all other types of nodes. The flow function `Id_path` represents paths in which all the nodes are `Identity`, indicated by the keyword `identity`. We use this indication in later versions of `propagate` and `reduce` to eliminate nodes and flow variables.

4.7 Sparse Data-flow Evaluation Graphs

Path simplification is versatile. Besides being able to do local analysis and interval analysis, path simplification can build SDFEG to solve sparse data-flow problems efficiently.

To compute an SDFEG, it is necessary to find paths consisting solely of identity flow functions, and to prune away all those flow-graph nodes which are connected to nodes in the SDFEG nodes through identity paths. Sharlit identifies such paths with the path simplification rules in Figure 4-17. Figure 4-18 compares the SDFEG computed using Sharlit and those computed by Choi's [19].

Sharlit's SDFEG differs in that we use nodes to represent some of the edges in Choi's SDFEG—We indicate those nodes with dashed circles. These nodes are required because the only way Sharlit can modify the edges of a node u in the graph is to replace all of u 's edges with one edge from u 's ancestor to u . Therefore, at the node `F`, the only way to connect `F` to its SDFEG predecessor is indirectly, as ancestors of `C` and `E`.

The key point that makes SDFEG efficient is it can eliminate many flow variables by using identity path functions. Our procedures `reduce` and `propagate` takes advantage of this property.

4.8 How Rules Compute Paths

Section 4.4 introduces path simplification rules and described informally how they work. This section goes into more details, discussing how `absorb` rules can save space and how

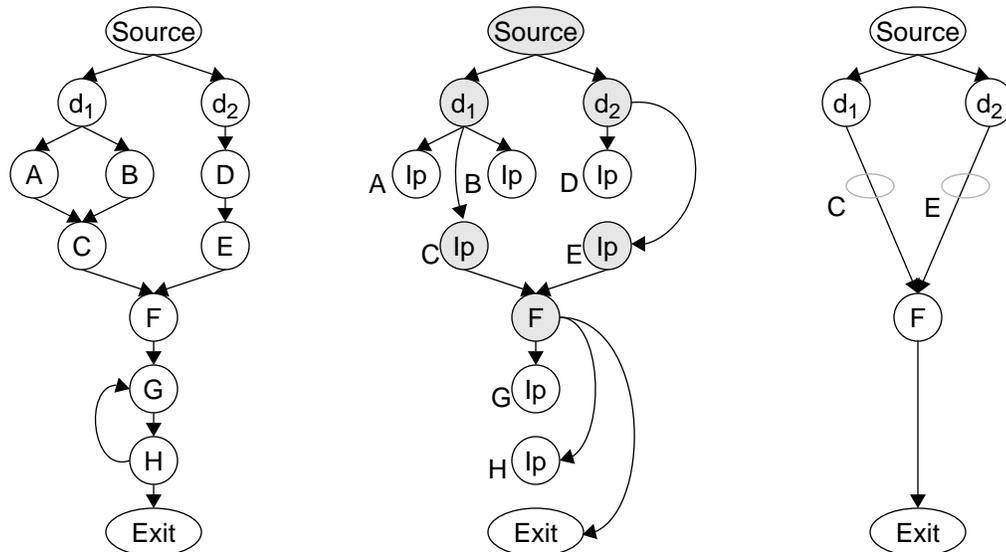


Figure 4-18 Example of simplification SDFEG

We show three control flow graphs: on the left, the original flow graph; in the middle, the graph after path simplification with eliminated nodes unshaded; on the right, the SDFEG obtained by applying Choi's algorithm [19]. In all graphs the abbreviations Ip stands for the path function `Id_path` defined in Figure 4-17. All nodes but d_1 and d_2 are identity flow function.

In the middle graph, the path simplifier has simplified the paths that are identity into `Id_paths`. We have shaded those nodes that are retained by reduce. The unshaded, eliminated nodes also have Ip paths from their ancestors. Of special note is that the path from d_1 to C can be found, while the path from Source to F cannot. This is because C can be reached from its unique ancestor d_1 with an `Id_path`, while F cannot be reached in that manner from its ancestor Source.

Our SDFEG differs in that we use nodes to represent some of the edges in the Choi's SDFEG—We indicate those nodes with dashed circles. We do so because we wish to avoid modification of the edge information in the flow graph, an operation that may be expensive. The edges from d_1 to D and d_2 to D can be represented more efficiently by setting the ancestor of C to d_1 and of E to d_2 .

procedures—`reduce_step`, `eval`, and `path_compress`—that do path computations use the rules to compute paths.

4.8.1 Absorb Rules Save Space

Path functions computed by the analyzers can potentially take a lot of storage. For example, the path function `AV_bb` in Figure 4-13 has two sets of expressions. Thus, just as we must be careful about which flow variables to retain, we must be careful about which path functions to retain. The traditional way of dealing with this problem in traditional optimizers is to group instructions into basic block. Therefore only one path function is required for each basic block.

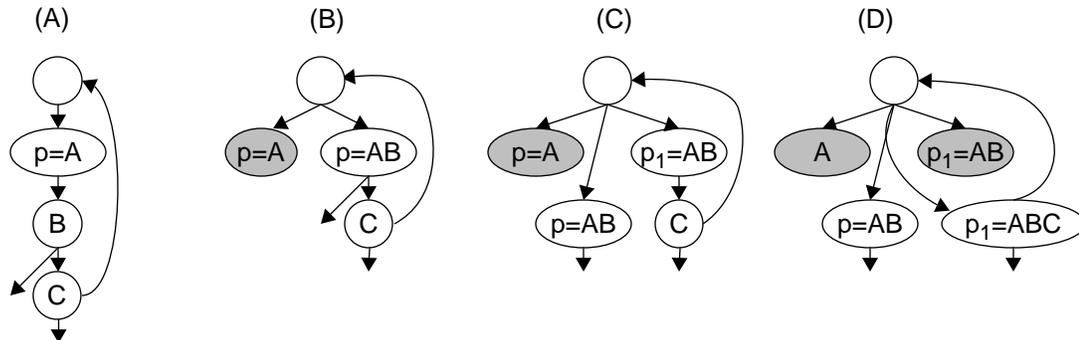


Figure 4-19 Reusing flow functions in sequences of node

In (A), when computing the path expressions for B, we know that A will be eliminated by `reduce`, hence we can reuse the data structure used to represent A's flow function. This is absorption. The result of the absorption appears in (B). We indicate that A will be eliminated by hashing it. In `reduce_step`, we set `new_flow(A)` to `nil` to indicate that its path function has been reused.

In (B), however, `p` (now equal to the path expression AB) cannot be reused because it has two successors. In fact it has to be copied, which is shown in (C).

In (C) the path function `p1` can now be used to absorb C, giving the graph in (D).

In Sharlit, the primary mechanism to save space is absorb rules together with reference counting of flow functions. To see how absorption saves on space, see Figure 4-19. Assume that we have the rules: `p create`, `p absorb p`, `p absorb A`, `p absorb B`, and `p absorb C`. In Figure 4-19(A) we apply the rules `p create`, followed by `p absorb A` to compute the path function corresponding to the path expression A. In (B), we apply `p absorb B` directly giving the path expression AB. We can reuse `p` because we know that the node A won't need its path function because `reduce` will remove it. However, in Figure 4-19(B) the node B cannot be removed because the node B—now corresponding to path expression AB—cannot be reused because AB may be used to compute the path expression for B's other successor. The simplifier makes a copy of `p` before continuing with the node C.

To decide when to reuse a path function, we keep a reference count—a count of the successors—for each path function. When composing a path function `p` with its successor, and `p`'s count is one, we can use it directly in an absorb rule; if its count is not one, then it must be copied, its count decremented, and its copy is used directly in absorption. When we add a successor to a path function (as when path simplification connect nodes to their ancestors), its count must be incremented.

There are two exceptions to the reference counting. Path functions associated with loop headers are treated as though they have infinite reference counts. This is necessary because they are never eliminated by `reduce`. Identity flow functions are treated similarly. But because identity flow functions have no local variables they can be allocated statically and shared.

4.8.2 Using Rules in Path Computation

The procedures `simplify`, `eval`, and `path_compress` builds path expressions abstractly using operators `.`, `+`, and `*`. We show how Sharlit turns this process of computing path expressions into invocation of rules. We define three functions corresponding to the path operators:

1. `compose(f, g)`:
 - a) If $f = f_{\perp}$ or $g = f_{\perp}$ then return f_{\perp} to indicate failure.
 - b) If $f = f_{\text{id}}$ then create a new path function representing g , that is, with the same data-flow effect as g . This creation can occur if by either applying the rules— p **create** g ; or the sequence— p **create** g followed by p **create** g . If exactly one of these possibilities are possible by the rules given in the specification then return p ; otherwise return
 - c) If $f = p$ for some path function p . Then apply p absorb g if possible; otherwise go to the next case.
 - d) Otherwise, return `compose(compose(f_{id} , f), g)` but only if the rule sequence to do it exists.
2. `join(p_1, p_2, \dots, p_a)`:
 - a) If $a = 1$ return p_1 .
 - b) Make a copy of the first path by doing $p \leftarrow \text{compose}(f_{\text{id}}, p_1)$, then apply p join p_2, \dots, p join p_a . Return p if successful, otherwise return f_{id} .
3. `star(p)`:

Invoke the rule p join, if the rule is present in the specification, otherwise return f_{\perp}

With these functions, we can interpret the path computation $(p_1 + p_2 + \dots + p_a)f$ as `compose(join(p_1, p_2, \dots, p_a), f)`, and pq^* as `compose(p , star(q))`.

The above functions are generated as table lookups. For example, `compose` would index into a composition table to determine what action to take. Because rule sets are small, Sharlit generates these tables by exhaustive search of the rule sequences.

4.9 Summary

This chapter and the last have shown how to solve DFA problems with Sharlit, and how Sharlit can generate solvers for these problems. The chief benefit of using Sharlit is that the compiler writer can use a simple model of data-flow analysis—a model in which each instruction is a node, and in which data-flow information is computed and made available at each node. But Sharlit does this in an space- and time-efficient manner.

Sharlit DFA specifications are extensible. It is easy to add flow functions and simplification rules. Sharlit takes care of incorporating them into the analyzer. This extensibility makes it easy to prototype data-flow analyzers. We can start with a straightforward, slow analyzer which does no local analysis at all. Then we can add rules to make the analyzer faster. Furthermore, we can add new rules to reflect changes to the intermediate representation.

Sharlit integrates many powerful data-flow analysis techniques. It can generate data-flow analyzers which can perform loop analysis of programs, and it can generate sparse data-flow evaluation graphs.

5

An Architecture for Modular and Flexible Optimizers

When writing large, complicated software, programmers commonly decompose the software into smaller, more manageable modules. Similarly, we decompose a large, complicated optimizer into a set of smaller, manageable data-flow analyzers. Previous chapters show how we can write specifications for data-flow analyzers—also called solvers—and how Sharlit generates them from the descriptions. This chapter describes the software architecture that Sharlit offers for building optimizers. The architecture defines abstract interfaces for control-flow graphs and solvers. It also includes a proposed mechanism to compose generated solvers into more complex ones. Together, the abstractions and the ability to combine solvers offer a new method to write optimizers that are more modular and flexible.

Many optimizers consists of distinct phases that correspond roughly to individual optimizations. Our architecture promotes modularity at a level finer than the phase level: we can build optimizations as chains or products of solvers. In a chain, its constituent solvers operate sequentially on the flow graph. In a product, its constituent solvers cooperate to solve a more complex DFA problem. We can chain and *multiply* solvers because Sharlit's architecture standardizes the external interface of generated solvers and the flow graph, and that interface supports chaining and multiplying. Standardizing interfaces constrains how solvers can depend on one another, contributing to further modularity. And these interfaces do not restrict the power of Sharlit. On the contrary, we show that products are powerful by showing how we can build the static-single assignment graph of a program as a product of solvers.

Sharlit's architecture lends flexibility to an optimizer: reordering and adding solvers is easy even if the solvers modify the control-flow graphs (CFGs). Solvers can delete nodes.

They can add new information to each node. They can add new nodes and new kinds of instructions. But as long as such changes are made using the facilities in the architecture, it is easy for another solver D_2 to operate on the modified flow graph. All we do is augment D_2 with flow functions corresponding to the new type of instructions inserted into the graph by D_1 .

The rest of this chapter has two major sections. The first discusses the two major abstractions that form our optimizer architecture. The second shows how to combine solvers together.

5.1 Data-flow Analyzers and Control-Flow Graphs

Sharlit provides two main abstractions for building optimizers: solvers and control-flow graph. The following two subsections define the abstractions by describing their external interfaces, which we have implemented as C++ classes. The interfaces tell us what functions the abstractions provide, and what functions the abstractions (and hence Sharlit) expect from the compiler writer.

5.1.1 Data-Flow Analyzers or Solvers

A solver consists of C++ classes that Sharlit builds by subclassing two predefined classes, one (DF) to form solvers and one (NC) to form classes that hold information about nodes. For example, from the available expressions problem in Figure 4-3, Sharlit generates the classes `Avail`, `AV_expr`, `AV_identity`, and `AV_bb`. The class `Avail`, a subclass of DF, would contain functions responsible for solving the DFA problem. Each of the other classes, a subclass of NC, wraps up into one object the related information and routines of one kind of node, information such as the local part of a node specification, and routines such as the flow function and the action routines.

The class DF implements a generic solver template that contains functions basic to all DFA problems. The functions are summarized in Table 5-1. The class DF also contains *slots*—or virtual functions in C++ lingo—which are used to customize the solver. To build `Avail`, Sharlit fills in the slots with functions specific to the availability expressions problem. The slots in the template are described in Table 5-2. Other functions in the generic solver help to combine solvers. We describe those functions in Section 5.2.

Once we have described a DFA problem (`Avail`, for example), turned the description into a solver object, and installed it as part of an optimizer, we can solve for available expressions by calling the function `Avail::solve` on a flow graph `g`. Within `solve`, several functions are called. The member function `init` initializes the analyzer object, invokes control-flow analysis, then invokes member functions that correspond to each step

Member function	Description
<code>DF::solve(g, lim)</code>	Apply the next four procedures below. The following code outlines the steps that <code>solve</code> invokes: <pre> D::solve(CFG *g, int lim) { init(g); simplify(g); rg = reduce(g); iterate(rg, lim); propagate(rg, g); } </pre>
<code>DF::init(g)</code>	Initialize the solver to operate on the graph <code>g</code> — invokes control-flow analysis on the graph <code>g</code> , and initialize data structures used to solve the DFA problem on <code>g</code> and calls <code>DF::flow_map</code> to assign a flow function to each node.
<code>DF::simplify(g)</code>	Simplify paths in the graph <code>g</code> . If the Sharlit specification did not have simplification rules, <code>reduced_graph</code> equals the graph <code>g</code> .
<code>rg=DF::reduce(g)</code>	Use the information computed in <code>DF::simplify</code> , build the reduced graph <code>rg</code> in the form of a list of nodes. If the Sharlit specification has no path simplification rules, the graph <code>rg</code> equals the graph <code>g</code> .
<code>DF::iterate(rg, lim)</code>	Evaluate nodes (i.e. their flow functions) until reaching a fix-point on the graph <code>rg</code> . The argument <code>lim</code> bounds the number of iteration.
<code>DF::propagate(rg, g)</code>	Make a pass over the graph to compute the data-flow solution at nodes not in the graph <code>rg</code> . As the solutions are computed, pass them to action routines that perform optimization.
<code>DF::in_value(u)</code>	Return a flow value, the solution of the DFA problem at <i>entry</i> to the node <code>u</code> .
<code>DF::out_value(u)</code>	Return the flow value at <i>exit</i> to the node <code>u</code> .
<code>DF::alloc_in_var(u)</code>	Indicate that a flow variable be stored to hold the flow value at the entry of <code>u</code> .
<code>DF::alloc_out_var(u)</code>	Indicate that a flow variable be stored to hold the flow value at the exit of <code>u</code> .

Table 5-1 Functions provided by Sharlit for solving DFA problems

This table describes the member functions of the analyzer class that perform data-flow analysis. The symbol `DF` denotes the name of the analyzer class.

Member function	Description
<code>DF::create(t)</code>	Create an empty path with label <code>t</code> .
<code>DF::times(p, f)</code>	Extend the path <code>p</code> with the flow function <code>f</code> . The second argument <code>f</code> can be either a flow function or a path function.
<code>DF::join(p, f)</code>	Join two paths <code>p</code> and <code>f</code> . If <code>f</code> is a flow function, <code>join</code> converts it to a path first.
<code>NC::flow(u, v)</code>	Evaluate a flow function, passing it the node <code>u</code> and the input flow value <code>v</code> . The flow function modifies <code>v</code> to reflect the data-flow effect of node <code>u</code> .
<code>NC::in(u, v)</code>	Perform optimization at node <code>u</code> based on the input flow value <code>v</code> .
<code>NC::out(u, v)</code>	Perform optimization at node <code>u</code> based on the output flow value <code>v</code> .
<code>DF::flow_map(u)</code>	Returns a flow function—represented as some class <code>FF</code> . Sharlit associates the returned function with the node <code>u</code> .

Table 5-2 Functions generated by Sharlit

The above functions differ from those of Table 5-1 in that Sharlit generates them from the parts of the analyzer specification. The functions in Table 5-1 calls these functions to solve the DFA problems. The class `NC` represent one of the flow function classes. Sharlit generates one such class for each flow function in the analyzer specification.

of DFA—`simplify` (path simplification), `reduce` (graph reduction), `iterate` (iteration), and `propagate` (propagation). Chapter 3 describes these steps in more detail.

The generic solver and the virtual functions interact in the following way. During initialization, `DF::init` calls `NC::flow_map` to assign a node type—hence a flow function—to each node. During path simplification, `simplify` will make calls to `DF::create` to create paths; `DF::times` to extend paths; and `DF::join` to join paths. These path manipulation functions are constructed from the path simplification rules in the solver specification. During iteration and propagation, the data-flow effect of a node `u` is computed by calling `NC::flow` of the node-type class associated with `u`. During propagation, `propagate` computes a flow value—that represents the solution to the DFA problem at a node `u`—and passes it to `u`'s action routines (`NC::in` and `NC::out`).

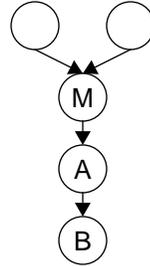


Figure 5-1 Computing the data-flow solution at a node

This figure shows that, in general, some computation such as meets and flow function evaluations are necessary to compute the solution at a node in the flow graph. In the above, to compute the flow value at entry to the node B, it is necessary to take the meet at the node M, and evaluate the flow functions at both M and A.

The technique of generating the solution, then immediately using it in the action routines serves as the primary means by which optimization occurs. Implementing optimizations in this manner frees us from the task of storing flow values as part of the flow graph, and lets Sharlit decide on how to store the flow values, which flow values to omit, and what evaluation strategy to compute the missing flow values from the stored ones. Section 4.3 describes the elimination of flow variables.

There are two other ways to access the flow values. First, the action routines of a solver can access the flow values of another solver when both are combined together. We describe this in Section 5.2. Second, a solver provides functions called `in_value` and `out_value` to access the solution of the DFA problem outside the context of propagation. Since Sharlit only stores flow values at some of the nodes in the graph, calling `in_value` and `out_value` requires, in general, meet operations and evaluations of flow functions to compute the desired flow value from flow variables stored in the CFG, as depicted in Figure 5-1. If an access function is called repeatedly at a node `u`, then it would be prudent to call `alloc_in_var(u)` or `alloc_out_var(u)`, which tell Sharlit to maintain a stored variable at the entry or exit of `u`. Subsequent calls to `in_value(u)` and `out_value(u)` fetch directly from the stored flow variable, thus avoiding unnecessary repetition of meets and flow function evaluations.

5.1.2 Control-Flow Graphs

A key goal of Sharlit's flow graph abstraction is to not restrict the choices of internal representation available to compiler writers. They could use any internal representation, as long as the representations offer the interface described below. Sharlit provides a default

flow graph implementation, the class `FG`. The class serves as a template whose functions can be replaced by class derivation. A compiler, for example, can represent programs as abstract syntax trees if it supplies a subclass of `FG` to be used with the solvers.

Thus the flow graph interface delineates the border between two views of the internal representation. On one side, the generic solver views the CFG as an abstract graph structure, without considering the internal structure of each node. This view is provided by the class `FG`. The class contains edge information and functions that compute information required to do path simplification and data-flow analysis. On the other side, code supplied by the compiler writer to customize the generic solver—flow functions, flow maps, and path simplification rules—must necessarily manipulate the internal structure of the nodes because that code depends on program operations. For the code that customizes the solver, we supply a C++ class called `FGnode`.

The class `FGnode` has only one field that holds a unique non-negative number called `unique`. When we add a node to the graph, the class `FG` assigns a unique number to the node. Solvers use this unique number as an index into tables of information such as data-flow information and node type objects. For example, a solver would store `ancestor(u)` (See Chapter 3) as an array of node pointers indexed with the unique identifier of `u`. Using the unique number instead of storing pointers in `FGnode` to associate information with a node has two advantages. It does not clutter the nodes with fields. Several solvers can operate on the same flow graph and the same nodes without knowing about each other, increasing modularity and flexibility.

Compiler writers can also use the unique number to associate information with each node. Moreover, we are free to add fields to the flow graph nodes by deriving the base class `FGnode`. For example, we can build a flow graph of quads by making the quad data structure a subclass of `FGnode`. Thus in Figure 4-3, which declares nodes as having type `instruction`, Sharlit assumes that we have previously defined `instruction` as a subclass of `FGnode`.

Before we can solve a DFA problems, we must build the flow graph with the functions described in Table 5-3. We register a node `u` into the flow graph with the `enter(u)`, and connect a directed edge from the node `u` to the node `v` with the call `link(u, v)`. We must indicate the source and sink of the flow graph by calling the member function `set`. Then we solve DFA problems on the graph by giving it as argument to a solver. During the propagation step of the solver, the action functions can add and remove nodes with `push` and `remove`. The changes go into effect when `FG::update` is called. This delay is necessary because it may be unsafe to change the flow graph when the solver is operating on it.

Table 5-4 shows the functions of `FG` that the generic solver calls. It finds out the suc-

Function	Description
<code>FG::enter(u)</code>	Add a node to the control-flow graph. Upon entering <code>u</code> , we say that <code>u</code> is registered.
<code>FG::link(u,v)</code>	Add an edge <i>from</i> the node <code>u</code> to the node <code>v</code> .
<code>FG::set(source,sink)</code>	Identify the source and sink of the flow graph as two previous registered nodes: <code>source</code> and <code>sink</code> .
<code>FG::update()</code>	Update the flow graph according to changes logged by the following two functions.
<code>FG::push(u,v)</code>	Log the insertion of a node <code>v</code> in front of a node <code>u</code> in the graph. The predecessors of <code>u</code> become the predecessors of <code>v</code> , and <code>u</code> becomes the unique successor of <code>v</code> .
<code>FG::remove(u)</code>	Log the removal of a node <code>u</code> from the graph.

Table 5-3 Functions for building and modifying the control-flow graph

cessors and predecessors of a node `u` with the call `next_nodes(u)` and `prev_nodes(u)`. The other functions in the table has do with control-flow analysis.

5.1.2.1 Control-Flow Analysis

Before we can apply a solver to a flow graph, we must apply control-flow analysis (CFA) to the graph. CFA derives information about the flow graph, information used in the operation of all solvers. During path simplification, a solver examines the most deeply nested loops first, then the next most deeply nested, and so on. Therefore CFA must discover the loop structure of the graph. During reduction, iteration and propagation, a solver examines nodes in reverse post-order. Therefore CFA must discover this order. A graph's loop structure and reverse post-ordering depends only on how nodes are connected, and does not depend on any particular solver nor any particular DFA problem. Thus it is natural to perform CFA within the flow graph class `FG`, instead of replicating the analysis in every solver.

CFA is invoked by calling the function `FG::analyze`. The result of the analysis can be fetched by calling three functions: `FG::headers` that returns a list of loop headers; `FG::body` that returns, given a loop header as argument, the body of a loop as a list of nodes in reverse post-order; and `FG::rpostorder` that returns a reverse post-ordering

Function	Description
<code>FG::next_nodes(u)</code>	Returns a vector of nodes that follow the node <code>u</code> .
<code>FG::prev_nodes(u)</code>	Returns a vector of nodes that precede the node <code>u</code> .
<code>FG::analyze()</code>	Performs or updates control-flow analysis. The first call to <code>analyze</code> performs the control-flow analysis as described in Procedure 3-1. Subsequent calls updates the control-flow information in respond to modifications of the graph.
<code>FG::headers(dir)</code>	If <code>dir</code> is 0, returns a list of headers for the forward flow graph, ordered with inner loops appearing before outer loops. If <code>dir</code> is 1, returns headers for the backward flow graph
<code>FG::body(dir, h)</code>	If <code>dir</code> is 0, returns the loop with header <code>h</code> for the forward flow graph. If <code>dir</code> is 1, returns the same information for the backward flow graph

Table 5-4 Functions used by generic solver to access flow graph

The above routines gives the view of the flow graph used by the solver. The first two routines facilitate access to edges. The third performs control-flow analysis. The last two routines return information computed by that analysis.

of the entire flow graph.

In a reducible flow graph, every loop has a unique entry node called the header which dominates every node in that loop. The call `body(h)` returns a list of nodes in the loop headed by `h` (except the header `h` itself). Within this list, inner loops are represented only by their headers. For an example of what `headers` and `body` returns, see Section 3.2.1. In an irreducible flow graph, some loops—called improper—will have more than one entry points. One of those entry points will be designated as the loop header, which does not dominate all the nodes in the loop. Section 3.2.2 shows how the solver handles irreducible flow graphs.

As the solvers can operate on the forward and backward flow graph, our CFA technique discussed below is applicable in both direction and returns two sets of information. A flag argument `dir` tells `FG::headers` and `FG::body` which set to return.

The default CFA algorithm provided by FG is based on Tarjan’s algorithm for testing reducibility of a flow graph [81] and on a closely related algorithm by Graham and Wegman [39]. In addition to finding loop headers, Tarjan’s algorithm computes an ordering of the nodes with which to apply Ullman’s T_1 and T_2 transformations [86] on a graph. The appli-

cation order will turn a reducible graph—but not an irreducible graph—into a single node, thus providing a convenient test for reducibility.

We modified Tarjan’s algorithm so that it computes loop bodies instead of an application ordering. Procedure 5-1 shows the main analysis function `FG::cfa`, which is called from `FG::analyze`. Within `FG::cfa` are three steps. This following will outline the steps in the context of reducible flow graphs. Readers interested in other aspect of control-flow analysis should refer to the literature [39, 59, 78, 80, 81]. To see how the path simplifier operates on irreducibilities see Section 3.2.2.

The first step `DF::dfs` performs a depth-first search and assigns two numbers to each node—`preorder` and `postorder`—that correspond respectively to orderings of the nodes in a pre-order and post-order traversal of the depth-first spanning tree (DFST) of the graph. These numbers are used to detect back-edges and to compute the reverse post-ordering.

Every loop has a back-edge, an edge (u, h) in the flow graph in which the target h *dominates* the source u [2]. For any node u , its dominator must lie on the path from the root of the DFST to the node u . Therefore, a back-edge from u must connect u with one of the nodes on that path. We can detect such edges by observing that when DFS visits a node u , all the nodes on the path from the root to the node u are *stacked*: every node on the path is represented by an activation of DFS on the call stack. Because DFS assigns `preorder` on entry and assigns `postorder` on exit, an edge (u, h) is an back-edge if the node h has non-zero pre-order, but zero postorder.

The second step sorts the back-edges so that in a nesting of loops the back-edges of inner loops appear before back-edges of outer loop. We sort according to the following condition: consider two back-edges $e_1=(u_1, h_1)$ and $e_2=(u_2, h_2)$. Then e_1 appears before e_2 if `preorder[h1] > preorder[h2]`. The condition gives use the desired loop order because if loop e_2 —the loop formed by e_2 —contains loop e_1 , then the header h_2 must dominate the header h_1 . Thus `dfs` must visit h_2 before h_1 .

The third step use the sorted back-edges to compute the reach-under set, the set of nodes dominated by a loop header and that can reach the loop header through the loop back-edge. We can visualize this step as generating a set of derived graphs. After computing the reach-under set of the innermost loop, it replaces the loop with its header. This replacement is accomplished by setting `highpt[u]` which `reach_under` uses to skip over replaced loop bodies. The reach-under computation is repeated until the loop has no loops. At which point, all remaining nodes become the loop body of a loop headed by the source node. See Figure 5-2 for an example of an application of `FG::cfa`.

It is important to recognize that `FG::reach_under` resembles a depth-first traversal

```

FG::cfa(){
    int pre=1,post=number_of_nodes();
    /* compute depth-first spanning tree */
    dfs(source_node,pre,post);
    be_sort(back_edges);
    /* perform reach-under computation */
    for( (u,h) in back_edges ){
        append h to the headers list;
        reach_under(h,u);
    }
    append source to the headers list;
    put all remaining nodes not in a loop-body
    in loop_body(source);
}

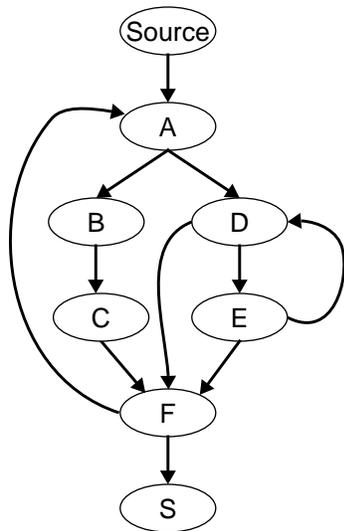
FG::dfs(FGnode *u,int &pre, int &post){
    preorder[u]=pre;
    pre=pre+1;
    for (v in successors[u]){
        if (preorder[v]==0)
            /* not visited yet */
            dfs(v,pre,post);
        else if(postorder[v]==0)
            /* (u,v) back-edge */
            add (u,v) to list of back_edges;
    }
    postorder[u]= post; post = post-1;
    append u to the rpostorder list;
}

FG::reach_under(FGnode *h,FGnode *u)
{
    if(u!=h){
        if(highpt[u])
            reach_under(h,highpt[u]);
        else if(postorder[u]<postorder[h])
            mark loop_body[h] as improper;
        else {
            highpt[u] = h;
            for(v in predecessors[u])
                reach_under(h,v);
            add u to loop_body[h];
        }
    }
}

```

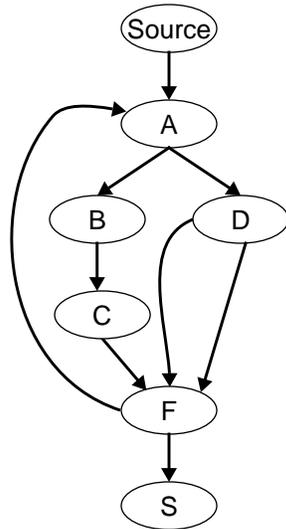
Procedure 5-1 Control-flow analysis in detail

The pseudo-code above expands on the control-flow analysis given in Procedure 3-1. The first step computes the depth-first spanning tree, represented by the two sets of numbers `preorder[u]` and `postorder[u]`. As part of that computation, all the back-edges are found. The back-edges are sorted to direct the reach under computation (see text).

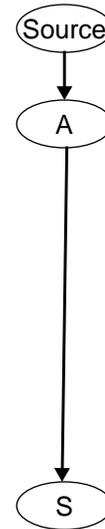


Original Flow Graph

	pre	post	loop bodies
Source	1	1	A, S
A	2	2	B, C, D, F
B	3	5	
C	4	6	
D	7	3	E
E	8	4	
F	5	7	
S	6	8	



First Derived Graph



Second Derived Graph

Figure 5-2 Example of control-flow analysis

This figure shows the original flow graph and its two derived graphs after `FG::reach_under` has processed the loops headed by D and A. The table shows the information computed by control-flow analysis.

and a topological sort that follows predecessors instead of successors. This resemblance is why `FG::reach_under` inserts nodes into `loop_body` in topological order as desired for path simplification.

5.2 Combining Data-flow Analyzers

Besides generating a solver directly from a Sharlit specification, we can also build a solver by chaining or multiplying other solvers.

A chain of two solvers D_1 and D_2 applies D_1 on the flow graph \mathcal{G} , then updates the control-flow information in \mathcal{G} to accommodate for changes made by D_1 , then applies the solver D_2 on the updated graph. The main benefit of chains is that they allow us to build optimizations that are composition of simpler optimizations, represented in our example as the solvers D_1 and D_2 .

A product of two solvers D_1 and D_2 interleaves the steps of the solver—path simplification, reduction, iteration, propagation. Flow functions of the solvers can depend on the flow values from both solvers. The action routines can use the solutions computed by both solvers to change the program. The main benefits of products are that they allow us to build more complicated solvers in which several simpler solvers cooperate to solve a DFA problem, and to build optimizations that use the solution from more than one DFA problem to drive code transformation.

5.2.1 Chains

Chains have been implemented as part of Sharlit and have been used to build optimizations. If we think of D_1 and D_2 as optimizations, then a chain of the two solvers would be a composition of two optimizations. Changes to the flow graph—such as adding or deleting nodes—invalidate the information computed by CFA. Sharlit can update the CFA information, as long as the changes to the graph are performed via the functions in Table 5-3.

Suppose D_2 is a solver that is not compatible with the solver D_1 — D_1 introduces new kinds of instructions into the flow graph which D_2 does not understand. We can make the output of D_1 palatable to D_2 by adding new flow functions for these new kinds of instructions.

Figure 5-3 shows how to create a chain and outlines its solve routine. In between calls to the `solve` of its constituents solvers, calls are made to `FG::update`, a function that updates the CFA information—the header list, loop bodies and the reverse post-ordered list. This can be done in one traversal of the reverse post-ordered list and each loop bodies. Nodes inserted in front of a node u are inserted in front of u in the loop bodies and also in reverse post-ordering. If a node v is inserted in front of a header h , then v replaces h in the

```

DFchain C(D1, D2, ..., Dn); /* creating a chain */

C::solve(FG *g, int lim)
{
    D1->solve(g, lim);
    g->update();
    D2->solve(g, lim);
    g->update();
    ...
    Dn->solve(g, lim);
}

```

Figure 5-3 Creating a chain

The declaration declares a chain of n solvers. The procedure `C::solve` outlines what occurs inside a chain.

headers list and `h` is prepended to the loop body.

In general, changes to the flow graph invalidate the solution to the DFA problems. That is, the solution computed was for the original flow graph and does not account for the modifications. Therefore, a chained solver needs to be careful in using data-flow information computed by previous solvers.

5.2.2 Products

Products let us build solvers for more complex DFA problem by combining simple solvers. Products introduce several new concepts: *product flow values*, and *interacting flow functions*. The following sections give an example of why multiplying solvers is desirable, explain the new concepts, and show how the phase of the product solver operate. In particular, we show how the iteration and propagation interleaves evaluations of flow functions from the solvers that make up the product.

5.2.2.1 An Example: Constant Propagation with SDFEGs

Consider constant propagation (Section 2.1.3) within a procedure. For simplicity, we formulate our DFA problem on two variables, `x` and `y`. In this DFA problem a flow value—a set of two bindings $\{(x, c_1), (y, c_2)\}$ —defines the values which `x` and `y` have at entrance or exit of a node. To create an output flow value from an input flow value, the flow function of an instruction such as `y=x+1` would extract the value of `x`, add one to the value, and store the result back into `y`.

The disadvantage of such a DFA problem is that every flow value stored must carry both

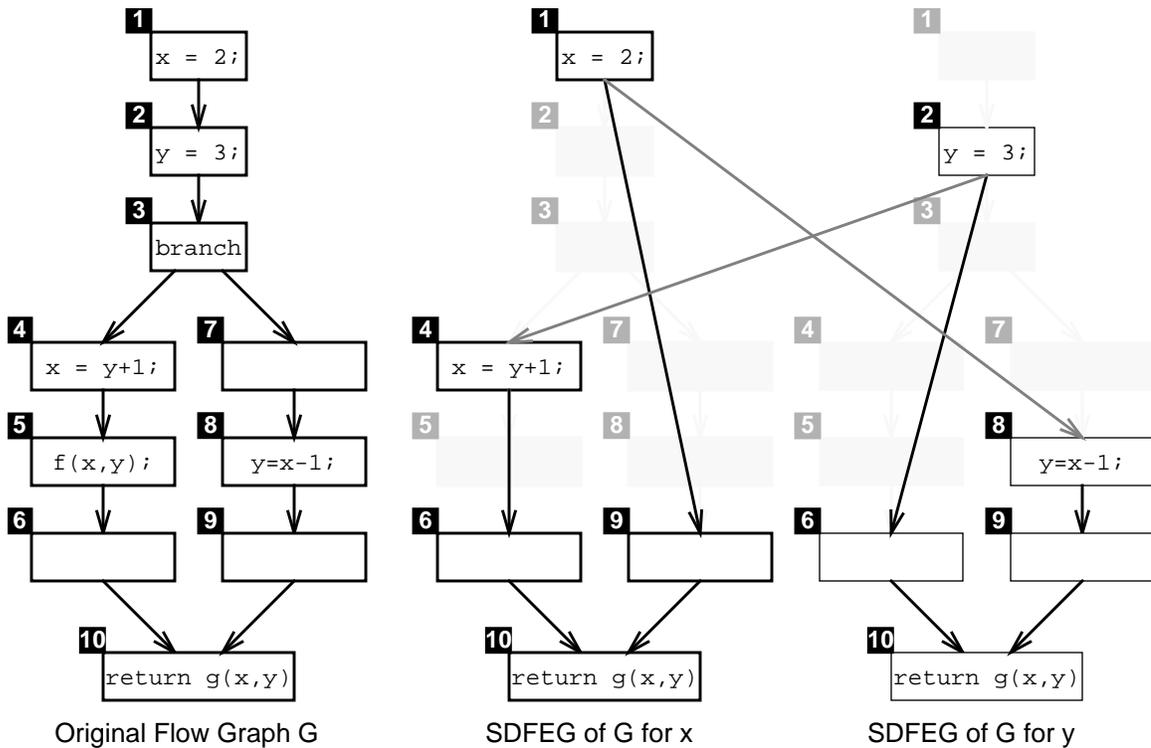


Figure 5-4 Multiplying two SDFEGs to do constant propagation

The original flow graph is shown on the left; the SDFEG for the variable x , in the middle; the SDFEG for the variable y , on the left. For a variable, its SDFEG, as described in Section 4.7, contains only those nodes many generate a new value for the variable, and some meets. The dotted arrows communicate values between SDFEGs. The solid arrows of the SDFEGs communicate values within the graph

x and y 's value. We can reduce space and time by using sparse data-flow evaluation graphs, one for x and one for y . As pointed out in Section 4.7, for sparse data-flow problems (a DFA problem in which many nodes have identity the flow functions) a SDFEG stores fewer flow variables than the traditional formulation. Furthermore the storage of x and y bindings are decoupled.

Figure 5-4 shows a flow graph, and its two SDFEGs. The SDFEG for the variable x contains two kinds of nodes. The first kind are nodes (1, 4, and 10) that modify flow values: node 1 and 4 binds new values to x , and node 10 performs a meet which binds a new value to x . The second kind are the nodes (6 and 9) required by the first kind: node 6 and 9 hold flow values for the meet operation in node 10. Similarly, the SDFEG for the variable y contains the same two kinds of nodes but corresponding to y . Edges within an SDFEG represent identity paths in the flow graph. The SDFEGs store bindings for x only at nodes 1 and 4, and for y only at nodes 2 and 8.

The key advantage of a product is that it lets solvers communicate during iteration. In

Action of the Iterator	Input Flow Value		Output Flow Value	
	x	y	x	y
Eval SDFEG x at node 1	?	?	2	-
Eval SDFEG y at node 2	2	?	-	3
Reconstruct input product flow value, then eval SDFEG x at node 4	2	3	4	-
Eval SDFEG x at node 6	4	-	4	-
Eval SDFEG y at node 6	-	3	-	3
Reconstruct input product flow value, then eval SDFEG y at node 8	2	3	-	1
Eval SDFEG x at node 9	2	-	2	-
Eval SDFEG y at node 9	-	1	-	1
Eval SDFEG x at node 10	?	-	?	-
Eval SDFEG y at node 10	-	?	-	?

Table 5-5 Iteration step of a product solver

The above table shows the actions taken by the iterator of the solver formed by multiplying the SDFEGs of x and y in Figure 5-4. A dash in a column indicates that the variable is not referenced by the action. A question mark indicates the value is unknown. For interacting flow functions such as those assigned to node 4 and node 8, the iterator reconstructs the input flow product flow value.

our example, this communication occurs at node 4 in x 's SDFEG and node 8 in y 's SDFEG. To determine the new values of x and y , the flow functions of the two nodes must obtain values from the other SDFEG. Figure 5-4 shows this flow of information with dashed arrows. Communication occurs through product flow values—cartesian products of the SDFEG's flow values—and interleaved evaluation of flow functions from different SDFEGs. Nodes that require cross communication are assigned *interacting* flow functions by the SDFEGs. Table 5-5 shows the actions taken by the iterator steps of the product solver. The iterator visits nodes in reverse post-order but at each node, it can evaluate, not one, but several flow functions. And at interacting nodes, the product flow value is reconstructed. The flow functions can then obtain the flow value of another SDFEG by extracting it from the product flow value. Reconstruction also occurs during propagation so that action routines have access to all the flow values.

5.2.2.2 Product Flow Values and product specifications

One can imagine a product flow value as the *cartesian product* of flow values of a product's component solvers. The components solvers are called *slices*. Sharlit expects the compiler writer to supply the declaration for the product flow values, so as not to restrict how they are represented. For example, they can be represented as a list or as a lookup table in the above constant propagator.

The product solver extracts components of the product flow value, and passes the components to its slices. As extraction functions depend on the type of the product flow value, Sharlit expects the compiler writer to provide extraction functions. Each slice has its own extraction function that we add by deriving the solver class that defines the other parts of the slice. By adding different extractors to a solver, we can reuse the solver in different products.

Product flow values and associated functions to allocate, copy and take their meet are provided to Sharlit using a *product specification*. This specification is just like a normal solver specification, containing `new_value`, `copy_value`, and `meet`, but excluding a `flow_map`, flow functions, action routines, and simplifier rules. A product specification creates an empty product object which we turn into a working solver by adding slices to it.

5.2.2.3 Modifying and Non-Modifying Solvers

A product can have many slices, but only one slice is permitted to modify the flow graph. Thus, we need to distinguish between *modifying* and *non-modifying* solvers. A modifying solver is one that has action routines, even if they don't actually modify the flow graph. A non-modifying solvers is one that does not have action routines.

5.2.2.4 Functions of Generic Solver for Supporting Products

As products interleave the phases of its slices, we must provide functions which give access to finer details of each phase. Instead of providing just a single iteration function that operates on the whole graph, we refine iteration into functions that operate on individual nodes. We refine propagation in a similar manner.

As described in Procedure 3-6 and Procedure 3-7, the function `iterate` and `propagate` consists of loops nests in which each iteration of the inner loop operates on an single node. We expose the inner loop body as two functions—`iterate_in`, `iterate_out`, `propagate_in`, and `propagate_out`. Table 5-6 shows these functions. Intuitively, we can think of the “_in” functions as computing an input flow value. They are used to reconstruct the product flow value. We can think of the “_out” functions as computing the data-flow effects. The above functions permit us to paraphrase `DF::iterate` and `DF::propagate` as shown in Procedure 5-2. The corresponding functions for the product solver have the same structure except that calls to “_in” are replaced with

Member function	Description
DF::extract(Fvalue *_W)	Returns the component of the product flow value _W that corresponds to this slice. It is important to note that extract returns a pointer to the component and any changes made to the component will be reflected in the product flow value.
DF::iterate_in(FGnode *a, FGnode *u, Fvalue *_V)	Compute the input flow value for a path that starts at the exit of node a (the ancestor) to the exit of the node u. If a is nil, the path starts at the entrance of u and ends at the exit of u. The flow value is returned by copying into the flow variable pointed to by _V.
DF::iterate_out(FGnode *u, Fvalue *_V, Fvalue *_W)	Compute the output flow value of the node u. The argument _V points to the input flow value to u; the argument _W points to the product flow value for use by the flow function of u. When the solver is used outside a product, _W is the null pointer.
DF::propagate_in(FGnode *u, Fvalue *_V)	Similar to DF::iterate_in, except it doesn't consider ancestors and the computed input flow value is fed to the in action routine of u.
DF::propagate_out(FGnode *u, Fvalue *_V, Fvalue *_W)	Same as DF::iterate_out except that it calls the out action routine of u. Propagation always use the original flow function assigned to u, and not the flow function assigned to the node by the path simplifier.

Table 5-6 Functions of generic solver for supporting products

a series of calls to “_in” of the slices; “_out” with calls to “_out” of the slices. We give more details about the phases of the product solver below.

5.2.2.5 Initialization

Suppose we have a product solver made up of the solvers D_1, D_2, \dots, D_n . Then the first phase is initialization. This is the most straightforward and consists of calls to the initialization phase of the component solvers, as show in Procedure 5-3 on page 80.

Initialization must check that each node is not assigned more than one interacting flow function. We'll explain this restriction when we discuss the iterator.

```

DF::iterate(list l, lim)
{
    int changed=0;
    int iter=0;
    while(!changed && i<lim){
        for(u in l){
            iterate_in(ancestor[u],u,_V);
            iterate_out(u,_V,0);
        }
    }
}

DF::propagate()
{
    for(u in g->rpo()){
        propagate_in(u,_V);
        propagate_out(u,_V,0);
    }
}

```

Procedure 5-2 Paraphrase of `DF::iterate` and `DF::propagate`.

```

P::init(g)
{
    D1->init(g);
    ...
    Dn->init(g);
    Check that each node has, at most, only one
        interacting flow function
}

```

Procedure 5-3 Initialization phase of product solver

```

DF::simplify_node(h,u,int needs_input)
{
    if(needs_input){
        keep[p]=1 for all predecessors p of u;
    }
    /* the same code as the old DF::simplify_node
       (See Procedure 3-3) */
}

CP::simplify_region(h)
{
    for(u in g->body(h))
    {
        keep[u] =D1->simplify_node(h,u,interacting(u));
        keep[u] |=D2->simplify_node(h,u,interacting(u));
    }
}

```

Procedure 5-4 Sketch of new `Simplify_node` and `simplify_region`

5.2.2.6 Path Simplification

If all the solvers of a product did not interact, neither would their simplifiers. But when solvers interact, their simplifiers must adjust to ensure that the iterator can reconstruct the input flow value for interacting flow and path functions. The danger lies in that if a solver D_i deems a node u as interacting, and another solver D_j eliminates a predecessor v of an interacting node when v is necessary for computing the D_j 's input flow value for u . The elimination can occur because D_j does not know that u is required by D_i .

We add to `simplify_node` a flag to indicate when the iterator needs to reconstructs the input flow value when it encounters the node code u . The flag tells the simplifier that it must retain all predecessors of u for use by interacting flow functions¹. Procedure 5-4 sketches out what the simplifier will look like with this new flag which we named `needs_input`. It is important to see that this flag will not hamper simplification—although a node is retained, its flow function can still participate in simplification.

Interacting flow and path functions cannot be absorbed by other paths. Sharlit enforces this restriction by disallowing rules from absorbing interacting flow functions. The restriction is necessary because paths are represented by noting the ancestor node (where the

1. Unless the predecessor is connected to its ancestor by a identity path.

```

CP::reduce()
{
    Make M an empty list.
    for(u in reverse(g->rpo()))
    {
        if(g->loop_header(u)){
            M->prepend(u,[D1,D2,...,Dn]);
            if(ancestor[u])
                keep[ancestor[u]]=1;
        } else if (keep[u]){
            M->prepend(u,[Di's])
                such that Di->keep[u]==1 and order
                the solver list so that interacting
                flow funtions come first;
            if(ancestor[u])
                keep[ancestor[u]]=1;
        }
    }
    return M;
}

```

Procedure 5-5 The reducer of a product solver

information flows in) at the final node (where the information flows out). Admitting interacting flow functions inside a path would mean that information flows into the path other than at the ends. That would complicate our path representation.

Procedure 5-4 shows the product simplifier for a region. As is shown in Chapter 3, `simplify_region` is called for each loop in the graph. The product simplifier invokes the simplifiers of the slices on each node, passing them a flag to indicate when an interacting node. If any of the solvers keeps a node (indicated by the return value), then the product solver also keeps the node.

5.2.2.7 Reduction

Reduction takes the result of simplification and produces a smaller graph with which to apply iteration. It takes the nodes marked as *kept* and back propagates keep flag in reverse topological ordering (ignoring back-edges). Propagating the keep flag through back-edges has been taken care of by the simplifier when it finds headers that are kept. A node that is kept *may* be in the reduced graph, subject to elimination due to identity path optimization. Procedure 5-4 shows the reducer of the product solver—I have omitted optimization for

identity paths so as to simplify this exposition. For details on identity path optimization, see Chapter 3.

Like the generic reducer, the reducer generates a list of nodes—ordered in reverse post-order—which iteration will solve the DFA problem on. Unlike the generic solver, the reducer associates a list of slices with the nodes on the list M . The list of slices indicate that the node is in those solvers' reduced graph. Referring to Figure 5-4, both slices (the SDFEG for x and y) would be on the list of slices for node 6 and 9, while only x 's SDFEG would be on the list for node 4.

5.2.2.8 Iteration

Like the iterator shown in Procedure 3-6 on page 38, the product iterator solves the DFA problem on a reduced graph by evaluating flow functions as dictated by the list generated by the reducer. Unlike the generic iterator, at any particular node the product iterator may execute more than one flow function.

Ostensibly, the product solver uses product flow values—but the iterator uses product flow values only when it encounters interacting flow functions. At non-interacting nodes, the components of the product values are distributed throughout the slices. A particular node may have a flow variable for one slice, but not for another. Since interacting flow functions need the entire flow value, the iterator must gather all the information together. This process is depicted in the function `reconstruct` of Procedure 5-6. Previously, the simplifier has ensured that every solver maintains enough information to generate its component of the product flow value. This usually means that predecessors of the interacting node will remain in reduced graph.

A product iterator, when it encounters an interacting node, must evaluate interacting flow functions before the others. This requirement is an artifact of how flow functions compute their output from their input. They transform their input flow value directly into an output flow value. Transformation saves space (and time) because Sharlit doesn't need to create a new flow value on every flow function evaluation. If the product iterator evaluates non-interacting flow functions first, they may modify some of the product flow value components. Thus the reconstructed flow value will no longer represent a valid input to interacting flow functions. Furthermore, validity is guaranteed if every node have at most one interacting flow function.

5.2.2.9 Propagation

Procedure 5-7 shows the product propagator. The propagator makes a pass of the graph, computing a flow value for every node in the flow graph. In the case of a product, the flow values are product flow values where each component represents the solution of the DFA problem of each slice.

```

CP::reconstruct(FGnode *a, FGnode *u, Fvalue *_W)
{
    D1->iterate_in(a,u,D1->extract(_W));
    D2->iterate_in(a,u,D2->extract(_W));
    ...
}

CP::iterate(list l,lim)
{
    int changed=0;
    int iter=0;
    iterate_init();
    while(!changed && i<lim)
    {
        changed=0;
        for(u in l){
            if(!interacting(u)){
                let (u,d1),(u,d2)...be found in l such
                that di is one of D1, D2, ...
                changed |= d1->iterate_node(u,
                    d1->extract(_W),0);
                changed |= d2->iterate_node(u,
                    d2->extract(_W),0);
                ...
            }else {
                let (u,d1),(u,d2)...be found in l such
                that di is one of D1, D2, ... and such
                that d1 has assigned the interacting
                function to the node.
                reconstruct(d1->ancestor[u],u,_W);
                changed |= d1->iterate_node(u,
                    d1->extract(_W),_W);
                changed |= d2->iterate_node(u,
                    d2->extract(_W),0);
                ...
            }
        }
        iter++;
    }
}

```

Procedure 5-6 Reconstruction of product flow value and the iterator

```

CP::propagate()
{
    for(u in g->rpo()){
        D1->propagate_in(u,D1->extract(_W));
        D2->propagate_in(u,D2->extract(_W));
        ...
        Dk(u)->propagate_out(u,
            Dk(u)->extract(_W),_W);
        Call Propagate_out of all other solvers
        except for Dk(u).
    }
}

```

Procedure 5-7 The product propagator, `CP::propagate`

In the above, $D_{k(u)}$ refers to the unique solver that assigned u an interacting flow function.

Like iteration, the propagator must handle interacting nodes specially, and at such nodes, the propagator also evaluate interacting flow functions first. But unlike iteration, the propagator builds the product flow value at every node. It passes the product flow value to the action routines with which they can perform optimizations.

5.3 Summary

This chapter has shown how Sharlit’s architecture for writing optimizers promote modularity and flexibility. Modularity is achieved by writing the optimizers as a series of data-flow analyzers or solvers. The current version of our optimizer architecture can chain solvers, but multiplying solvers is not fully implemented. Flexibility is achieved because the dependences between the solvers are controlled. We can reorder them much more easily than we can reorder optimizations in conventionally-written optimizers.

Another key feature of the architecture is that many of the objects defined in it have slots or virtual functions. Their defaults can be overridden to customize solvers and flow graphs. All the phases—initializer, simplifier, reducer, iterator, and propagator—of the solvers can be overridden. We can, for example, replace the simplifier or reducer with ones that are more efficient for a particular DFA problem.

6

A Prototype Optimizer

This chapter gives an example of an optimizer organized using the software architecture espoused in Chapter 5. This optimizer operates on and outputs SUIF programs. We describe several scalar optimizations, demonstrating that Sharlit is applicable to a wide variety of DFA problems and code transformations. We show that by using the architecture we have made the optimizer modular in design, flexible and extensible in how the optimizations connect, and effective in generating good code. We delve into the details of two transformations: code motion and induction variable rewrite.

Code motion moves computations from more-frequently executed regions of the program to earlier, less-frequently executed regions. It consists of several data-flow analysis steps—all use traditional bit-vector techniques. It shows how data-flow analysis can be used to compute directly what code to move and where to move it. To demonstrate flexibility, we turn the code motion algorithm into a strength reducer.

Induction variables rewrite (IVR) is an example of a code transformation performed via symbolic analysis and extensive use of path simplification. IVR demonstrate the use of Sharlit in implementing a code transformation that is often not thought of as a DFA problem.

Although we are describing the SUIF optimizer, we will try to keep the explanations of the optimizations independent of the specifics of SUIF. The only exception is Section 6.2.1, but the SUIF used there is easy to comprehend. SUIF will be explained in the next chapter.

6.1 Structure of the Prototype Optimizer

Figure 6-1 shows the structure of the SUIF optimizer. The unshaded square boxes represent sets of DFA solvers that correspond to an optimization. To encourage prototyping and

experimentation, we make it easy to reorder and add data-flow analyzers. All solvers take as input and modify a control-flow graph (CFG). If a solver adds new kinds of instructions to the flow graph, we can make these new instructions acceptable to other solvers by adding new flow functions to them, as suggested in the previous chapter. The shaded box in Figure 6-1 represent special phases. The graph constructor builds the CFG by calling functions in

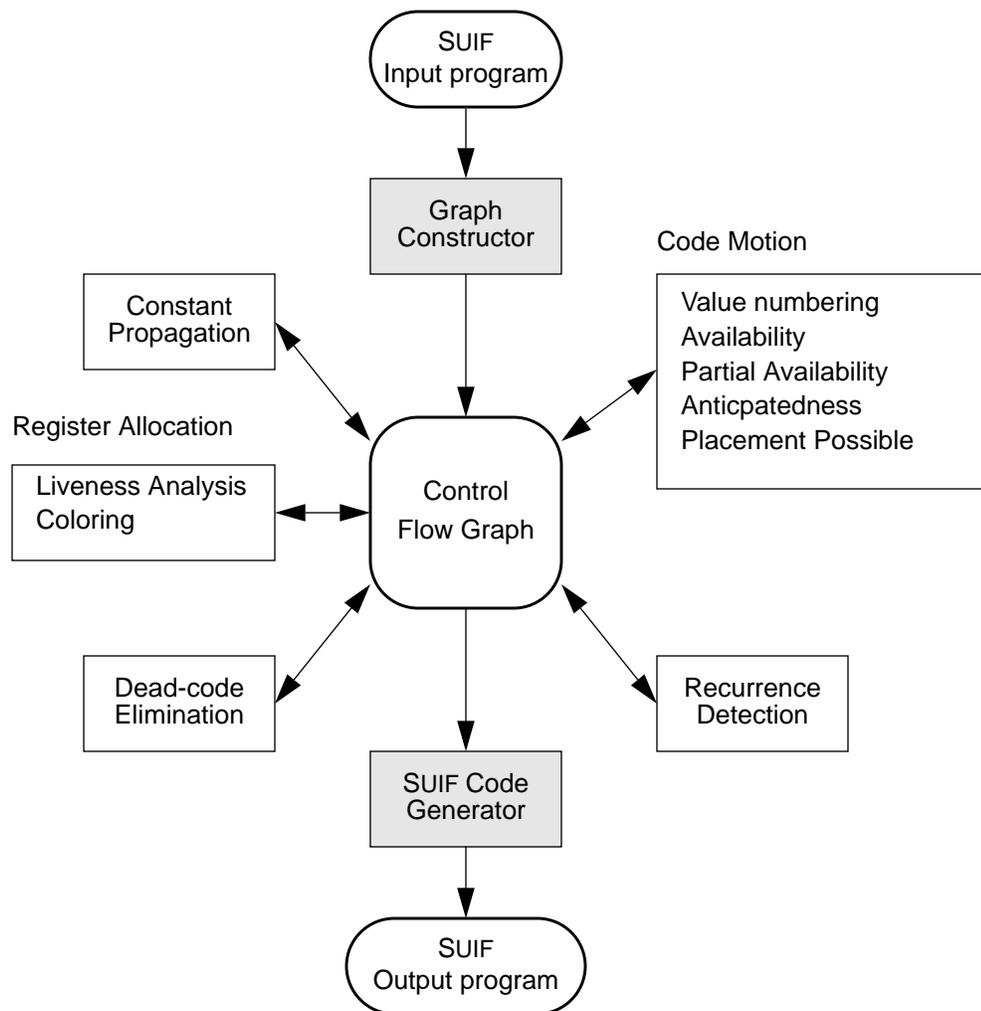


Figure 6-1 Structure of the SUIF optimizer

We envisage an optimizer as a set of data-flow analyzers—generated by Sharlit—that operate on central program representation, a control-flow graph. The diagram above groups analyzers together such that each group corresponds to an *optimization*, such as register allocation or code motion. The shaded boxes represent special phases. The graph constructor uses the facilities provided by Sharlit to build the control-flow graph. The code generator—implemented as a Sharlit analyzer—performs some cleanup and writes out the code.

the flow graph interface. Another special phase, the code generator, cleans up the CFG, and writes it out as a SUIF program. The clean-up routine takes the form of a Sharlit solver.

The optimizer reads a SUIF program, one procedure at a time, and translates the procedure into a CFG. Then a sequence of phases is applied to the CFG. The phases implement the following analysis and transformations:

- *Code Motion:* The SUIF optimizer moves code by eliminating partial redundancies, a technique first proposed by Morel and Renvoise[65], and later refined by Joshi and Dhamdhere [50, 51], and by Chow [20]. Approaches based on eliminating partial redundancy offer an important advantage: they perform global common-subexpression elimination, loop-invariant code motion, and strength reduction simultaneously.

Code motion consists of phases that solve for available, partially available, and anticipated expressions [20]. This information serves as input to a *placement* phase, a data-flow analysis problem that solves for which expressions to delete, which expressions to insert, and where to insert them. We have implemented all the analyses with Sharlit.

- *Register Allocation:* We use the priority-based coloring approach first implemented by Chow [20]. This optimization consists of two data-flow analysis steps that compute live ranges, followed by a coloring routine that builds the interference graph and colors it. The coloring routine does not fit the data-flow model, thus we did not write it with Sharlit.
- *Reaching Definitions:* The optimizer applies the standard bit-vector based reaching definitions problem and annotates each use of a variable with the set of definitions that reach it. Some SUIF parallelization phases use this information.
- *Constant propagation and folding:* This phase uses Kildall's technique [54] which determines, for every instruction in the flow graph, a set of variables with constant values. The phase replaces each instruction that computes a constant value with an instruction that loads the constant.
- *Copy Propagation:* This phase, similar to constant propagation and folding, tracks assignments of the form $x \leftarrow y$ where x and y are program variables. We call the assignment $x \leftarrow y$ available at an instruction, if the procedure executes the assignment on every path leading to the instruction. When an assignment $x \leftarrow y$ is available at an instruction u , then the copy propagator substitutes every use of x in u with y .
- *Dead store elimination:* This phase performs liveness analysis, and eliminates instructions that computes values that are never used.
- *Induction variable rewrite:* Section 6.4 explains this optimization in more detail.

Field	Description
<code>kind</code>	A tag used within <code>flow_map</code> , in conjunction with the instruction opcode, to determine which flow function to assign the node.
<code>instruction</code>	A pointer to an <i>instruction</i> , which can be a SUIF instruction, or internal instruction defined by other solvers.
<code>expr_no</code>	The expression number used in code motion, and computed by value-numbering.

Table 6-1 Fields of a flow graph node (class `IRnode`)

6.1.1 Flow Graph Nodes

As discussed in Chapter 5, flow graph nodes must be a subclass of the class `FGnode`. For the SUIF optimizer, we add the fields show in Table 6-1. Data-flow information is not stored in the node because that information is stored as flow variables by the solvers. The program code associated with the node is held in the field `instruction`. This field points to subclasses of the instruction base class, provided by the SUIF libraries. For example, several subclasses hold SUIF instructions. One way to modify the graph is to define new subclasses, instantiations of which can be inserted into the flow graph.

Why not insert regular SUIF instructions? Because SUIF instructions may not suit some optimizations, and extension using subclasses gives us flexibility. Strength reduction, for example, manipulates linear combination of variables. Therefore, it seems appropriate to replace instructions with equivalent linear combinations. We can place machine instructions into the internal representation if we wish to experiment with using machine dependencies in our optimizations.

For code motion, we store an expression number in the node. The expression number represents the *value numbers* computed by those instructions that participate in code motion. A later section discusses how value numbers are defined and used.

6.2 Code Motion

Our code motion algorithm—based on the elimination of partial redundancies (EPR)—operates by solving several DFA problems, and thus is well-suited for implementation with Sharlit. The following sections discuss the steps involved in eliminating partial redundancies and sketch out how we implement them with Sharlit.

Suppose that the node u evaluates an expression e . Then the expression e is *redundant* at u if every execution path up to u contains an execution of e and none of the variables on which e depends are modified after that execution of e . The expression e is *partially redundant* if there is one path meeting the above condition. We illustrate how EPR works with an example. In Figure 6-2(A), the expression $x+y$ in node B is redundant. To remove this redundancy, at node A save in a temporary t the value of $x+y$, and at node B replace $x+y$ with t . In Figure 6-2(B), the expression $x+y$ in node B is partially redundant because it is available on one incoming path but not on all paths. To remove the partial redundancy, insert an instance of $x+y$ in the path in which $x+y$ is not available. This insertion renders $x+y$ redundant in node B, which can then be removed. It is especially interesting to see that eliminating partial redundancy can move loop invariant expressions out of loops. In Figure 6-2(C), the expression $x+y$ is partially redundant—it is available through the back-edge but not from outside of the loop. To eliminate this partial redundancy, insert $t \leftarrow x+y$ just before the loop, and replace $x+y$ inside the loop with t . The effect is to move the loop-invariant expression out of the loop.

6.2.1 Value Numbering

The first step of code motion is *value numbering* [2], a process that identifies movable expressions. Value numbering gives to each syntactically equivalent expression—modulo commutativity and different node register numbers—a unique *expression number*, which is used to index bit vectors of expressions. The bit vectors are the flow values of the EPR solvers. The SUIF instructions on the left below would be given the same expression number as the corresponding SUIF instructions on the right,

```
ldc (s.32) nr#1,2          ldc (s.32) nr#2,2
add (s.32) nr#3,pr#1,pr#2  add (s.32) nr#4,pr#2,pr#1
```

since the instructions in each row compute the same expression: 2 and $pr\#1+pr\#2$.

To give a unique number to syntactically equivalent expressions, value numbering must determine whether two arbitrary expressions are equivalent. To speed up this time-consuming process, hashing is used. We compute a hash value—not to be confused with the unique expression number—for each expression from the hash values of its subexpressions. If two expressions have different hash values they must be different. If two expressions have the same hash numbers, then we must recursively test the equivalence of its subexpressions. Thus hashing reduces the number of explicit expression comparisons.

Value numbering resembles the constant propagator described in Section 2.1.3 because it resembles an interpretation of the instructions. In this case, flow values associate each

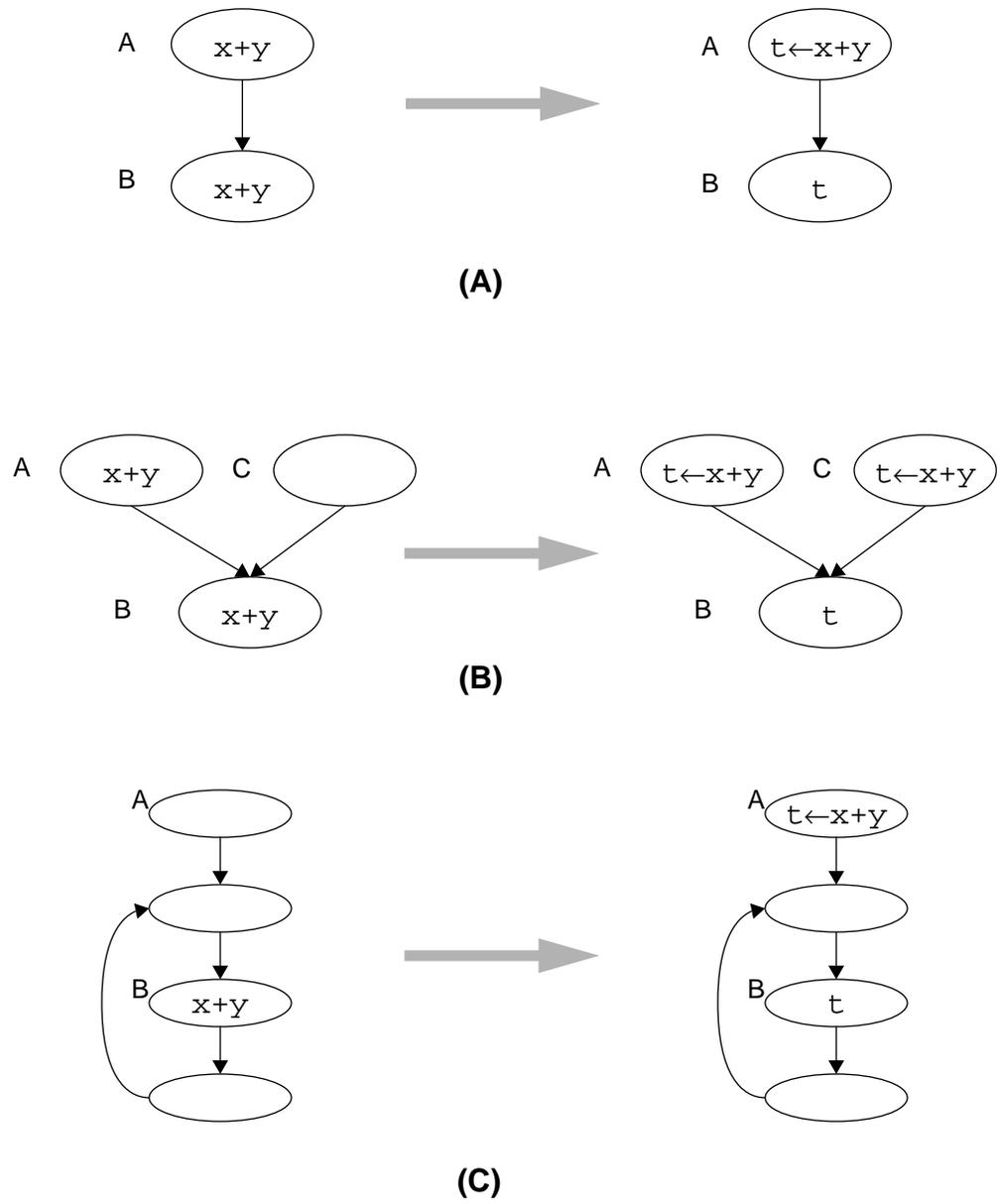


Figure 6-2 Redundancy and partial redundancy

node register with of the expression computed into the register. It associates each pseudo register with a unique atomic expression. Value numbering maps each SUIF instruction into operations on hash values; commutative SUIF instructions are mapped into commutative operations on hash values. Value numbering differs from constant propagation in that its meet operator resets flow values—that is, the meet operator initialize the flow value to the null binding that contains no associations for node registers. Thus value numbering takes only one pass of the flow graph.

As expressions are found, kill vectors are also computed. Each pseudo-register has a kill vector, the set of expressions whose value depends on the value of the pseudo-register. A write to a pseudo-register *kills* each expression e in this set, in the sense that re-evaluating e after the write may yield a different value than an evaluation of e before the write.

6.2.2 Availability and Partial Availability

The next two steps of EPR solve two classical DFA problems iteratively. The solutions to these DFA problems serve as input to the next step, the placement phase that decides where expressions are to be moved.

We define availability in Chapter 4 and we define partial availability as follows. An expression e is *partially available* at a node u if at least one path from the source to u has evaluated e and the value of e remains the same if re-evaluated at u . We can express the two problems as solving the set of bit vector equations:

Partial Availability:

$$\begin{aligned} O_{\text{pav}}(\text{source}) &= \emptyset \\ I_{\text{pav}}(u) &= \sum_{v \in \text{succ}(u)} O_{\text{pav}}(v) \\ O_{\text{pav}}(u) &= \text{en}(u) + \overline{\text{killed}(u)} \cdot I_{\text{pav}}(u) \end{aligned}$$

Availability:

$$\begin{aligned} O_{\text{av}}(\text{source}) &= \emptyset \\ I_{\text{av}}(u) &= \prod_{v \in \text{succ}(u)} O_{\text{av}}(v) \\ O_{\text{av}}(u) &= \text{en}(u) + \overline{\text{killed}(u)} \cdot I_{\text{av}}(u) \end{aligned}$$

Here $\text{en}(u)$ represents the expression—the field `expr_no`—of the node u , and $\text{killed}(u)$ represents the set of expressions killed by assignments within the node u .

The Sharlit specification for both problems are very similar to the one given for available expressions in Chapter 4. The flow functions implement essentially the equations for the output flow variable $O_{\text{av}}(u)$ and $O_{\text{pav}}(u)$. Each SUIF instruction gets turned into one of

these flow functions. The meet operators compute the product and the sum for availability and partial availability respectively. We use path simplification to do local analysis. The flow function for a basic block has the form expressed in the above equation, except “en” is a set instead of a single expressions. The solvers are modular, self-contained except for references to SUIF data structures, and the expression table.

6.2.3 Placement: Where to Insert Expressions

This section gives an informal explanation of the PRE algorithm. We explain how we solve a DFA problem to determine what computations to move and where to compute them. The next section discusses the details of implementation of this DFA problem with Sharlit. We’ll omit the proofs because other authors have covered them [20, 65, 55].

Morel and Renvoise, the originators of PRE, defined a DFA problem called *anticipated expressions* that leads us to the following formulation of code motion. Suppose we have a node v which computes the expression e . Then e is anticipated at a node u , if every execution path that contains u has the form

$$[\text{source}, \dots, u, w_1, w_2, \dots, w_n, v, \dots]$$

and computing e at u yields the same result as computing it at v . A sufficient condition for anticipatedness is that the nodes w_1, \dots, w_n do not modify the variables on which e depends. Incidentally, e is also anticipated at the nodes w_1, \dots, w_n . This definition leads naturally to the following set of equations to solve anticipated expressions:

$$\begin{aligned} I_{\text{ant}}(u) &= \text{en}(u) + \overline{\text{killed}(u)} \cdot O_{\text{ant}}(u) \\ O_{\text{ant}}(u) &= \prod_{v \in \text{succ}(u)} I_{\text{ant}}(v) \\ O_{\text{ant}}(\text{sink}) &= \emptyset \end{aligned}$$

Imagine copies of the expression e floating from v up—in reverse of control-flow—towards the source along paths in which e is anticipated. Where two or more paths that anticipate e join, a floating e replicates and a copy floats up each adjoining path. When two floating expressions meet, they merge before continuing. Eventually the floating e ’s reach the source or reach points—either exits or entrances of nodes—in which further progress is blocked by nodes that do not anticipate e . If e had reached the entrance of a node w then e is not anticipated at the exit of *at least one* predecessor of w , while if e had reached the exit of w then e is not anticipated at the entrance of w ($e \notin I_{\text{ant}}(w)$). The points at which floating stops marks the *frontier* of anticipation.

Now insert the expression e at this frontier. The insertions are *safe*, because anticipation guarantees that they do not introduce e at nodes in which evaluating e can cause errors.

Moreover, the insertions render the original computation of e redundant at the node v , and therefore the redundant e can be deleted.

However, the above method doesn't guarantee that inserted expressions are not themselves partially redundant, a situation depicted in Figure 6-3(A) in which shading indicates those nodes that anticipate e . There are two insertions: one at the exit of node A, and a partially redundant one at the entrance of node D. This deficiency in code motion is caused by *critical edges*, edges that connect a node with more than one successor and a node with more than one predecessor—such as the edges (B,D) and (F,D). Converting the critical edge, as illustrated in Figure 6-3(B), to a node moves the insertion from D to the node G, averting the partial redundancy.

The SUIF optimizer doesn't convert all critical edges into nodes but it tries to do so for the most important case: loops. The optimizer always inserts a loop preheader as shown in Figure 6-3(C), converting the critical edge that connects the loop pretest to the loop header, thus providing a place to hold code that moves out of the loop. Other critical edges are handled during the placement DFA problem.

Once all critical edges are converted to nodes, the above method of finding insertion points is optimal, in the sense that every execution path contains no more than the original number of computations of the expression e , and that any other set of safe insertions requires just as many evaluations as the optimal set of insertions. In particular, the insertions followed by deletions of newly redundant expressions will move code out of loops.

Optimality as defined above does not guarantee that optimized programs run faster. In fact, EPR is too eager and tends to place expressions as far up paths of anticipation as possible, lengthening live ranges, causing the register allocator to insert spills, and slowing the program. The following reformulation of anticipated expression, which we call placement or placement possible, limits unnecessary expansion of live ranges. The placement problem is shown below. The most striking differences between placement and anticipation is the presence of two factors $B(u)$ and $I_{\text{pav}}(u)$, which we discuss below.

$$\begin{aligned}
 I_{\text{pp}}(\text{source}) &= \emptyset \\
 I_{\text{pp}}(u) &= B(u) \cdot I_{\text{pav}}(u) \cdot (\text{en}(u) + \overline{\text{killed}(u)}) \cdot O_{\text{pp}}(u) \\
 O_{\text{pp}}(u) &= \prod_{v \in \text{succ}(u)} I_{\text{pp}}(v) \\
 O_{\text{pp}}(\text{sink}) &= \emptyset \\
 B(u) &= \prod_{v \in \text{pred}(u)} (O_{\text{pp}}(v) + O_{\text{av}}(v))
 \end{aligned}$$

We compute two sets, called insert and delete, that are computed from the placement information as follows:

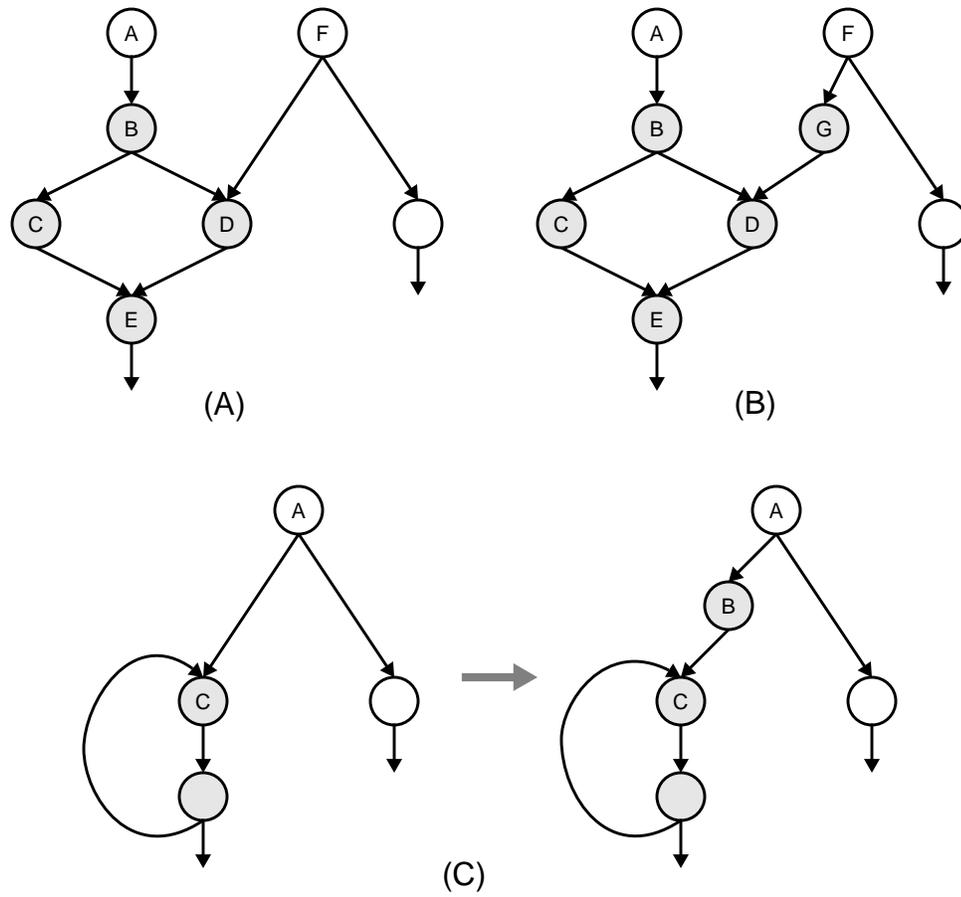


Figure 6-3 How partially redundant insertions occur

Consider only one expressions e . The nodes are shaded to indicate that e is anticipated at their exits.

$$\begin{aligned}\text{delete}(u) &= \text{en}(u)I_{\text{pp}}(u) \\ \text{insert}(u) &= O_{\text{pp}}(u)\overline{I_{\text{pp}}(u)O_{\text{av}}(u)}\end{aligned}$$

Deletion occurs if the expression floats away—an expression e at the node u is deleted if $e \in I_{\text{pp}}(u)$, a condition that indicates e will be inserted somewhere to make e redundant at u . The factor $\text{en}(u)$ ensures that in fact node u computes e , otherwise the deletion is moot.

As before, an expression e is inserted at its frontier, not of anticipation, but of placement. The factor $B(u)$ ensures that frontiers exist only between the entrance and the exit of a node, not between the entrance of a node and the exits of its predecessors. In other words the situation $e \notin I_{\text{pp}}(u)$ and $e \in O_{\text{pp}}(u)$ can happen at a node u . But the situation $e \in I_{\text{pp}}(u)$ and $e \notin O_{\text{pp}}(v)$ can never happen, where v is u 's predecessor (unless e is available at v). Thus, insertions occur only at the exits of nodes in which placement is possible at their exits but not possible at their entrance. If an expression is already available ($e \in O_{\text{av}}(u)$) then we can skip the insertion and instead use the already computed value. An advantage of this restriction of frontiers is that we can tell whether a node u is at the frontier of expression by checking $I_{\text{pp}}(u)$ and $O_{\text{pp}}(u)$ without needing to check the data-flow information at u 's predecessors or successors.

In Figure 6-3(A) if we had shaded nodes in which $e \in O_{\text{pp}}(u)$ instead of $e \in I_{\text{ant}}(u)$ then only nodes c, d and e would be shaded. Thus insertions can only occur at the exits of nodes c and d. This example illustrates the other advantage of the factor $B(u)$, its role in suppressing partial redundant insertions at unconverted critical edges.

The factor $I_{\text{pav}}(u)$ in the equation for $I_{\text{pp}}(u)$ stops unnecessary code motion and expansion of live ranges. If e is not partially available at u then no paths leading to u computes e and thus moving will not improve run-time.

6.2.4 Implementation of Placement with Sharlit

Placement problems, like availability and partial availability, are solved iteratively, using path simplification to perform local analysis. However, placement has two unique peculiarities that Sharlit must handle specially. First, although placement is mostly a backwards DFA problem, it contains some *bidirectional* computations. Second, placement uses the solutions of two forward DFA problems—available and partially available expressions—in reverse.

Most data-flow problems are unidirectional. Their equations either show a forward (backward) information flow: the input (output) flow value of a node u depends on the output (input) flow values of u 's predecessors (successors), and the output (input) flow value depends on u 's input (output) flow value. Though predominantly backwards, placement exhibits *bidirectionality* because of the factor $B(u)$. The factor has forward information

flow—the input I_{pp} of a node u depends on the output O_{pp} of u 's predecessors. While Sharlit automatically performs the products that compute O_{pp} , Sharlit does not do so for the product in $B(u)$. The flow functions must compute $B(u)$ explicitly.

Since Sharlit doesn't know about the information flow within $B(u)$, it may delete some of the flow variables required in computing $B(u)$. Treating placement as a backwards problem, Sharlit eliminates all output flow variables. We must tell Sharlit to keep those variables necessary to compute $B(u)$. At first, it seems that we need output flow variables O_{pp} at all nodes. However, that's not quite true. Suppose u has no other successor but the node v . Then we can show that in fact $I_{pp}(v) = O_{pp}(u)$ which lets us remove the term $B(v)$ from the equation of $I_{pp}(u)$ for all v that are successors of fork points. We indicate that a flow variable is needed at fork points by calling `DF::alloc_out_var` in `flow_map` as we assign flow functions to fork points.

A more vexing peculiarity of placement is its references to availability and partial availability in a direction opposite to the direction in which the availability flow values are generated. Using `DF::in_value` and `DF::out_value` would be expensive because of repeated meets and flow-function evaluations needed to compute the flow values for those problems. To forestall this expense, we trade time for space and tell Sharlit to allocate a flow variable at every node, for availability and partial availability.

6.2.5 Code Generation

The sets `insert` and `delete` are interpreted by action routines when inserting and deleting expressions. Each expression in the set `insert` is added into the graph as a new node with a zeroed instruction field. Later, a code generation phase turns the expressions into SUIF instructions. Deletions occur within action routines with calls to the function `FG::remove`.

6.3 Strength Reduction

Strength reduction turns affine expressions into induction variables, replacing multiplication operations with less expensive additions. Suppose a loop has an affine expression $4 * i$, and i is an induction variable with an initial value of 1 and a step of s . Strength reduction turns $4 * i$ into an induction variable t with initial value $4 * 1$ and step $4 * s$. Strength reduction, in general, can handle more complex expressions. It can deal with affine expressions where the coefficients are variables that are loop constants. It can replace other expensive operations—exponentiations with multiplications, for example. For a comprehensive discussion of strength reduction, see Allen and Cocke [6].

The SUIF optimizer uses a restricted form of strength reduction that operates only on affine expressions whose coefficients are constants. This restricted form is sufficient for

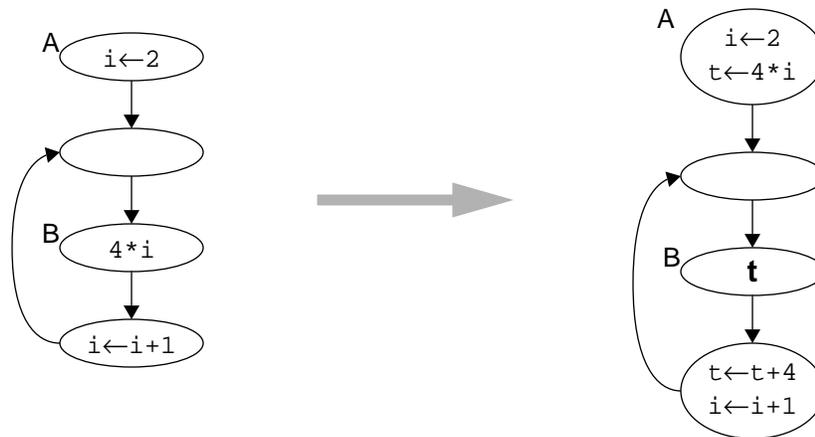


Figure 6-4 Using partial-availability elimination for strength reduction

improving most programs, as the major opportunities for strength reduction are address computations arising from array references. Such address computations are usually affine expressions.

The reducer is closely related to the EPR code motion algorithm discussed above. Joshi and Dhamdhere [50, 51] first pointed out that EPR can be used to perform strength reduction, if candidates—expressions eligible for strength reduction—are defined and increments of induction variables are interpreted as having no effect on candidates. This is illustrated in Figure 6-4 in which a loop has a candidate, $4 * i$. Because the increment $i \leftarrow i + 1$ has no effect on candidates, the candidate $4 * i$ can be considered as partially redundant in the loop. Therefore EPR moves $4 * i$ out of the loop. A later liveness phase determines which candidates are live and dependent on i , at the node containing $i \leftarrow i + 1$. The temporaries holding such candidate—in our example, the temporary is t —are then incremented.

In the SUIF optimizer, the strength reducer is constructed by grafting a new value numbering phase, and by adding new flow functions, for increments, to the DFA problems of the EPR algorithm.

6.3.1 Value Numbering

The SUIF optimizer restricts candidates to be affine combinations of the form:

$$c_0 + c_1r_1 + c_2r_2 + \dots + c_nr_n$$

and increments to be of the form

$$r \leftarrow r + s$$

where the r, r_1, r_2, \dots, r_n are pseudo-registers, and s, c_0, c_1, \dots, c_n are integer constants, and $n \geq 0$.

Candidates and increments are determined by using a modified version of the value numbering phase in Section 6.2.1. Each expression number, if it is a candidate, has an associated linear combination. As it scans instructions, value numbering combines these linear combinations. Suppose it encounters the SUIF instruction:

```
add (s.32) nr#3, nr#1, nr#2
```

when the node registers `nr#1` and `nr#2` are bound to candidates. Then the two candidates are added to form the candidate for `nr#3`. The new candidate is associated with the instruction's expression number. Suppose value numbering encounters the SUIF instruction:

```
cpy (s.32) pr#1, nr#4
```

when `nr#4` is bound to a candidate. Then the instruction is regarded as an increment if the linear combination associated with `nr#4` has the form `pr#1+c`.

Besides the information collected described in Section 6.2.1, value numbering also computes a set called *candidates* of all candidates detected in the procedure. We use this set to compute data-flow effects of increments.

6.3.2 Changes to the EPR Algorithm

To change the EPR algorithm so that it performs strength reduction, we must do the following:

- For increments, we add flow functions to all three DFA problems of the EPR algorithm. The flow functions compute the same data-flow equation except that the killed set is defined as:

$$\text{killed}_{\text{sr}}(u) = \text{killed}(u) - \text{candidates}$$

```

      K0=1
      K1=6
      DO 10 I=1,10
        DO 20 J=1,5
(A)   1          A(K1)=2*A(K0)
          K0=K0+1
          K1=K1+2
        20      CONTINUE
          K0=K0+15
          K1=K1+10
      10      CONTINUE

      DO 10 I=1,10
        DO 20 J=1,5
(B)   1          A(20*I+2*J-16)=2*A(20*I+J-20)
        20      CONTINUE
      10      CONTINUE

```

Figure 6-5 Induction variable rewrite

In statement 1, the compiler can rewrite the induction variables in terms of the loop indices: $K0$ as $20*I+2*J-20$, and $K1$ as $20*I+2*J-16$. After the substitution, the two induction variables are now redundant and are removed. Although this *strength-increasing* substitution could slow the execution of each iteration, it enables the compiler to parallelize the loop.

- The optimizer must insert increments to strength reduction candidates immediately after any variables on which they depend are incremented. We modify the dead register elimination algorithm to do this. Suppose a node increments a variable i , then any candidates that depend on i that is live at the exit of the node will require an increment.
- We modify the code generator algorithm so that it knows how to generate SUIF instructions from a linear expression.

6.4 Induction Variable Rewrite

Previous optimizations use path simplification in a relatively simple way such in local analysis. This section shows a program transformation, induction variable rewrite (IVR), that uses path simplification rules extensively.

Within each nesting of loops, IVR detects *synchronized induction variables*. IVR then replaces uses of synchronized induction variables with equivalent affine combination of loop indices. In Figure 6-5, for example, IVR detects that the variables $K0$ and $K1$ are synchronized with the loop indices i and j . On every iteration of the inner loop j increases by

```

      K0=1
      K1=6
      DO 10 I=1,10
          K0=K0+5
          K1=K1+10
          K0=K0+15
          K1=K1+10
10     CONTINUE

      DO 10 I=1,10
          K0=K0+20
          K1=K1+20
10     CONTINUE

```

Figure 6-6 Loop summarization

Path simplification computes the effect of loops on induction variable. Part (A) shows the overall effect of the inner loop summarized by the italicized statements. Part (B) shows the overall effect of the loop body of the outer loop.

1, while K_0 and K_1 increases by 1 and 2. On every iteration of the outer loop i increases by 1, while K_0 and K_1 increases by 10. After detection IVR substitutes uses of K_0 and K_1 with expressions that compute the value of K_0 and K_1 from the value of i and j . A subsequent pass of dead-store elimination removes the increments to K_0 and K_1 , made redundant by the substitutions.

On the face of it, IVR could slow down programs because it replaces simple references with expressions that may include multiplications. But IVR improves the effectiveness of dependency analysis [64], enhancing the effectiveness of parallelization and other optimizations that relies on dependence information. Moreover, IVR usually runs before parallelization. After parallelization, we run scalar optimizations to remove introduced multiplications.

Our implementation of IVR consists of two solvers. The first solver has only a path simplification step. It does not iterate nor propagate. The second solver performs only iteration and propagation.

The path simplification step of the first solver discover synchronized induction variables and their step sizes by summarizing the effects loops have on induction variables. The effect of the innermost loop in Figure 6-5 is to increment K_0 by 5 and K_1 by 10. This is shown in Figure 6-5(A) by replacing the loop with italicized statements. After summarizing the inner loop, path simplification will discover that the effect of the outer loop is to increment both K_0 and K_1 by 20 (Figure 6-5(B)).

The iteration step of the second solver—using the initial value and step size of the induction variables—computes the relationships between the induction variables and the loop indexes. In Figure 6-5, for example, at the top of the outer loop the solver brings together the following information: initial values of $K0$ and $K1$ (1 and 6), together with the loop effects ($K0=K0+20$ and $K1=K1+20$), and the initial value of the loop index (1). This information is used to determine that at the top of loop $K0=20*(I-1)+1$ and $K1=20*(I-1)+6$. The two relationships are also the initial values of $K0$ and $K1$ for the inner loop. Thus at the top of the inner loop

$$\begin{aligned} K0 &= 20*(I-1) + (J-1) + 1 = 20*I + J - 20 \\ K1 &= 20*(I-1) + 2*(J-1) + 6 = 20*I + 2*J - 16. \end{aligned}$$

The propagation step of the second solver replaces references of the induction variables with the relationship. In our example, the references to $K0$ and $K1$ are replaced with their corresponding expressions within the loops.

6.4.1 Representing Relationships between Induction Variables

Suppose a loop L has the variable i as index such that i has bounds l_i and u_i , and step size s_i . Every iteration increments i by s_i . During the last iteration i has the value u_i . Suppose the loop also has an induction variable j . If the compiler can determine bound l_j and step size s_j , then within the loop j can be expressed as:

$$l_j + s_j \frac{(i - l_i)}{s_i}$$

The bound u_i , as it does not appear in the expression, isn't necessary for computing the relationship between j and i . However, u_i is needed to determine that when the loop exits j has been incremented by

$$s_j \left(\left\lfloor \frac{(u_i - l_i)}{s_i} \right\rfloor + 1 \right)$$

If the loop L is nested within another loop, this expression is needed in order to relate j to the loop index of the outer loop.

The compiler, in general, must represent the expressions above symbolically—an expensive proposition. Fortunately, for the purposes of improving dependency analysis, it suffices for the compiler to consider induction variables that are affinely related to the loop index. Moreover, it suffices for the compiler to find affine relationships in which the coefficients are integers, the same relationships used during strength reduction.

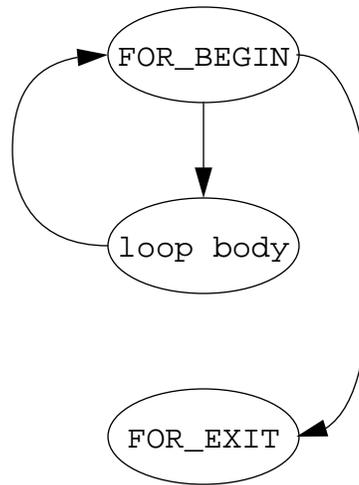


Figure 6-7 Control-flow graph for a loop

For IVR, only FOR-loops interest the compiler. A FOR-loop is delimited by two special nodes—a FOR_BEGIN node, and a FOR_EXIT node.

6.4.2 Representing Loop Structures

To apply IVR to a loop, the compiler must determine its index variable, and the lower bound and step size of the index variable. The information is most easily discovered for FOR-loops, which coincidentally are the ones of interest for dependence analysis. Therefore, it suffices for the compiler to consider only FOR-loops. If a FOR-loop never exits early—that is the index always reaches the upper bound—then, for any induction variables defined by the loop, the compiler can compute both expressions described in the previous section.

Thus, the loops of interest for IVR can be represented in the flow graph as in Figure 6-5. Two nodes demarcate the loop. The first, the FOR_BEGIN node, generates information that is propagated within the loop body. The second, the FOR_EXIT node, generates information that pertains to the overall effect of the loop. This information will become evident in the following sections.

6.4.3 Computing Induction Variables and Their Step Size

The flow functions used in the first solver of IVR represent the effects of CFG paths on integer variables. For a node that computes $z = x + y$, its flow function is the set containing the one *assignment*:

$$\{ z \leftarrow x + y \} .$$

This set tells us that the value of z at the *exit* of the node can be computed by taking the sum of the values of x and y at the *entrance* of the node.

Suppose we have two flow functions $f = \{z \leftarrow x+y\}$ and $g = \{x \leftarrow 2-z\}$ that represent respectively a node u and its unique successor v . We can find the flow function from the entrance of u to the exit of v by composing f and g —in this case, by substituting $x+y$ for z in g to obtain the flow function $f \bullet g$ or

$$\{z \leftarrow x+y, x \leftarrow 2-x-y\}.$$

In interpreting the flow functions, it is important to remember that variables on the right of the arrow denote values at the *entrance* of a path, while variables on the left denote values at the *exit* of a path. Two paths are joined by taking the intersection of their flow functions or sets.

We apply Kleene-star to a flow function to expose induction variables. The induction variables are exactly those variables x that appear in both sides of an assignment of the form $x \leftarrow x+e$ in which e is an expression that isn't modified in the loop. We call e the step expressions. For example, in the flow function

$$\{K0 \leftarrow K0+1, K1 \leftarrow K1+2\}$$

Kleene-star exposes the variables $K0$ and $K1$ as induction variables with step sizes 1 and 2 respectively.

In FOR-loops, Kleene-star is applied to the flow function of a loop body when the path simplifier examines the loop header or the FOR_BEGIN node. At that time, exposed induction variables and their step sizes are attached to the header. We represent the Kleene-star of the above flow function as

$$\{K0 \leftarrow IV(K0, 1), K1 \leftarrow IV(K1, 2)\}$$

where the term $IV(1, s)$ represents an induction variables with initial value 1, and step size s , all of which are expressed in terms of values from outside of the loop.

Composing the flow function of a loop with its FOR_EXIT node gives the function that represents the overall effect of the loop on its induction variables. In Figure 6-5, the flow function from the beginning of the outer loop's body to the exit of the FOR_EXIT for the inner loop would be $\{K0 \leftarrow K0+5, K1 \leftarrow K1+10\}$.

Whenever path simplification cannot determine an expression for a variable, it sets the variable to *bottom* or \perp , as constant propagation does in Chapter 2.

6.4.4 Propagating Induction Variable Relationships

The second solver resembles the constant propagator described in Chapter 2. But unlike constant propagation in which flow values bind constants to variables, the flow values of IVR binds expressions to variables. If the input flow value of a node u contains the binding (v, e) , then a use of v in u can be safely replaced by the expression e .

Like constant propagation, the second solver symbolically interprets instructions. However, besides adding new bindings, the second solver must also remove bindings that are no longer valid, a removal analogous to killing expressions in the available expression problem. For example, the flow function for an instruction $z=x+y$ must replace all bindings (v, e) in the input flow value with bindings (v, \perp) in the output flow value, if e contains the variable z .

In the second solver, the `FOR_BEGIN` and `FOR_EXIT` nodes play special roles. The `FOR_BEGIN` node computes the relationships between induction variables and the loop index, and binds that relationship to the induction variable for propagation within the loop. The `FOR_EXIT` node binds the induction variable to their final values for propagation outside of the loop.

6.5 Conclusions and Results

We have found Sharlit to be a useful tool. Its facilities are ideal for describing DFAs, from simple iterative ones, to ones that use symbolic analysis (recurrence detection) to ones that use interval analysis. Because of its extensibility, Sharlit is well-suited for experimenting with different internal representations and for implementing optimizations by first building a simple prototype which can be made more sophisticated when necessary.

Moreover, the SUIF optimizer is real. Our results (Table 0-2) show that this optimizer can optimize large, real programs—such as the PERFECT club [15]—as effectively as commercial compilers.

To my knowledge, this is the first reported use of a tool to build a practical optimizer. This thesis demonstrates that there is a modular way—modular below the level of individual optimizations—to build optimizers. The modularity makes it significantly easier to extend and modify the optimizer. Moreover, the tool makes it easier to bind these modules together.

A particularly important aspect of Sharlit is that it imposes a structuring discipline that constrains compiler writers in the way they can implement an optimizer. But as with other tool like YACC and LEX, this kind of constraint makes it much easier to understand and ultimately makes the compiler easier to modify.

	Description	Lines	SUIF -O (seconds)	MIPS -O2 (seconds)	SUIF/MIPS
LG	Lattice guage, Quantum chromodynamics	2327	97	106	0.82
LW	Liquid water simulation, Molecular dynamics	1237	1437	1405	1.02
NA	Nucleic acid simulation, Molecular dynamics	3976	421	377	1.12
SD	Structural dynamics, Engineering design	7607	88	105	0.84
SR	Supersonic reentry, 2-D fluid dynamics	3958	1270	1253	1.01
TF	Transonic flow, 2-D fluid dynamics	1985	243	232	1.05
TI	2-electron transform integrals, Molecular dynamics	484	259	244	1.06

Table 0-2 Comparison of SUIF optimizer with a commercial optimizer

PERFECT club execution results in seconds from runs on a Decstation 5000 with version 1.31 of the Mips compiler

7

SUIF

So far, this thesis has addressed the problem of structuring the internals of an optimizer. Now we turn our attention to the problem of integrating that optimizer into a compiler system that incorporates both scalar and parallelizing transformations.

7.1 Overview of the SUIF IL

It is difficult to support scalar optimization and parallelization in one IL because each desires program representations at different levels of abstraction. Effective scalar optimization prefers a low-level IL, to expose as many operations as possible. Array indexing, for instance, should be expressed as address calculations to expose common subexpressions and strength reduction candidates. Loops should be expressed in terms of labels and jumps, with landing pads where an optimizer can insert code and still preserve control dependences. Parallelizing transformations, however, prefer the loop structure to have a unique induction variable, and an explicit representation of loop bounds. Some dependence tests (for example, *extended GCD*[13]) require that each index expression in a multidimensional array reference be identifiable. Linearizing these array references by combining the index expressions into one lengthy computation would limit our choice of dependence tests.

SUIF has a core of small set of simple instructions with register operands. This set is called low-SUIF, and is similar to quad-based ILs used in traditional scalar optimizers.

To support parallelization, SUIF uses a register-naming convention that helps to extract expression trees from the SUIF instructions, and SUIF has explicit representation for high-level control flow and array indexing. When these features are present in a SUIF program, we say that the program is represented in high-SUIF.

Having high-SUIF is not the same as having a separate high-level IL. Except for high-level control flow and array-indexing operators, all other instructions, in both high- and

low-SUIF, are identical. Thus low-SUIF is a proper subset of high-SUIF. High-SUIF features do not exclude low-SUIF features: branches and labels (low-level control-flow), address arithmetic, high-level control-flow and array-indexing operators can all coexist together.

Low-SUIF instructions are chosen to minimize case analysis during optimization and code generation. We reduce the number of cases by having a small set of operations and by not hiding operations.¹ All computational instructions require a type argument. All but one instruction² require their operands be in virtual registers, even branch targets and memory addresses, and all results are returned in registers. Load and store instructions are the only way to move values between registers and memory. Load constant instructions or `ldcs` (lines 16--18 in Figure 7-1) are the only way to put constants into registers.

The SUIF type system is similar to the Ucode type system [20, 67]. SUIF types represent data types and sizes that are naturally representable on the target machine. In Figure 7-1, the types `(s . 32)` and `(a . 32)` stand for a 32-bit signed integer and a 32-bit address respectively. Even though SUIF types are machine dependent, past experiences [20, 67] have shown that we can easily control this dependency by parameterizing the compiler.

7.1.1 Registers

Normally, registers and low-level instructions like SUIF's are good for scalar optimizations, but undesirable for high-level transformations. The latter transformations prefer expression trees to be represented explicitly. Using expression trees reduces the work for high-level transformations because the large number of instructions are grouped into a smaller number of trees. Trees bound the computation and use of intermediate values, reducing the number of variables to be considered. On the other hand, scalar optimizations treat all values, intermediate and otherwise, equally. These optimizations want to find redundant computations and strength reduction candidates regardless of whether they compute intermediate values or not.

SUIF resolves this conflict by providing two different views of its instructions: expression trees and lists of quads. This is done by using a register naming convention. SUIF has three kinds of registers:

- *Hard Registers:* SUIF uses hard registers to refer to real machine registers.

1. Our call instruction `cal` does hide operations, those that load argument registers or stack and those that copy the returned result. These operations, however, depend on the target machines. We hide this machine dependence, to simplify our inliner.

2. In the case of the multi-way branch `mbr` which occurs rarely, we opted to have a format for easy code generation, with a list of branch targets.

```

1   n = 49;
2   k = 0;
3   for(i = 0;i<=n;i++){
4       A[k] = A[200];
5       k += 2;
6   }

1   ldc (s.32) nr#0,49; // n = 49;
2   cpy (s.32) pr#0,nr#0;// k = 0;
3   ldc (s.32) nr#1,0;
4   cpy (s.32) pr#2,nr#1;
5   *["for" "begin" pr#1 pr#3 pr#4 pr#5 "slte"
6       <function.f.label.L14>
7       <function.f.label.L15>];
8   ldc (s.32) nr#2,0;
9   cpy (s.32) pr#3,nr#2;
10  *["for" "ub"];
11  cpy (s.32) pr#4,pr#0;
12  *["for" "step"];
13  ldc (s.32) nr#3,1;
14  cpy (s.32) pr#5,nr#3;
15  *["for" "body"]; // for(i = 0;i<=n;i++){
16  ldc (s.32) nr#5,300;
17  ldc (s.32) nr#6,200;
18  ldc (a.32) nr#7,<global.A>;
19  array (a.32) nr#7,nr#8,32,(nr#6),(nr#5);
20  lod (s.32) nr#9,nr#8;
21  ldc (s.32) nr#10,300;
22  ldc (a.32) nr#11,<global.A>;
23  array (a.32) nr#11,nr#12,32,(pr#2),(nr#10); //
A[k] = A[200];
24  str (s.32) nr#12,nr#9;
25  ldc (s.32) nr#13,2;
26  add (s.32) pr#2,pr#2,nr#13;// k += 2;
27  *["for" "end"];

```

Figure 7-1 Program fragment and associated SUIF code

- *Node Registers*: Intermediate values—data with only one point of definition and one point of use, and both points are limited to a basic block—are held in *node registers*. These values are *single-def-single-use* and correspond to values generated and used inside an expression tree. In Figure 7-1, node registers are those instruction operands with *nr#* prefixes.

Node registers make it easy to reconstruct expression trees. To build trees, make a tree node out of each SUIF instruction and connect the instruction defining a node register to the instruction using the node register with a tree edge.

Besides being easy to build trees, bounding the lifetimes of node registers is useful to those parts of the compiler that don't use trees. For example, node registers provide clues to interpretive code generators [35]. These generators associate information, machine resources, and machine instructions with each encountered node register as it scans through a SUIF program. When the last and only use of a node register is encountered, they can forget the associations.

Although node registers are distinguished from other registers, the form of instruction is the same and independent of the kind of registers it uses. Therefore, even when the internal representation is an instruction list, the lifetime information provided by node registers are still present. Reaching definitions analysis takes advantage of this situation. Since it knows definitions of node registers are limited in extent, it ignores them, thus reducing the number of cases it must consider.

- *Pseudo Registers*: Pseudo registers hold values that are not single-def-single-use and values that flow between basic blocks. In Figure 7-1, register names with the prefix *pr#* denote pseudo registers. The compiler has assigned *pr#0*, *pr#1*, and *pr#2* to *n*, *i*, and *k* respectively.

Pseudo registers form an infinite register file within which we can allocate user and temporary variables.

The SUIF compiler identifies, in a procedure, those local variables with no aliases, and assigns each permanently to a pseudo register for the duration of the procedure. To find these local variables, the front-end scans a procedure and makes a list of those locals whose addresses are never taken. These variables are not the only candidates for permanent residence in pseudo registers. If a variable—including a global or compiler-generated temporary variable—has no aliases in a region of a procedure, we can promote that variable, throughout the region, to a pseudo register. Finding such candidates is more difficult, requiring the tracking of pointer values and alias analysis.

Splitting up register allocation into two phases—first allocating to pseudo registers, then allocating pseudo registers to machine registers—has several benefits. Because they are unaliased, pseudo registers are ideal candidates for allocation to machine registers. In

addition, assigning a pseudo register to a machine register involves only substituting the pseudo register number with the assigned hard (physical) register number.

Global data-flow analysis, in SUIF, concentrates on pseudo registers. Because pseudo registers are not aliased, gen/kill[2] information for them becomes very simple. When the analysis encounters an assignment of a pseudo register, only the data-flow information about that register has to be changed. The assignment cannot affect information collected about other registers. Similarly, this assurance of nonaliasing simplifies other passes by restricting what is changed by an assignment to a pseudo register.

SUIF's virtual registers cannot mediate all data-flow. Memory is required for arrays and those variables that are referenced indirectly. To simplify the representation of and operations on memory addresses, SUIF organizes memory with *paths*.

7.1.2 Paths

Addressing details and relationships are expressed using paths. Paths hide many addressing details from the high-level phases, details such as accesses to a function's activation record, accesses within a record, and up-level accesses. Paths encode aliasing relationships in a language-independent way, making it easy to pass aliasing information to the low-level phases.

The path system organizes the names of program variables (including temporary ones) and labels into a tree structure. At the leaves are variables, temporary variables, or fields of aggregates, all of which we will refer to as storage areas. An internal node groups a set of children nodes—related, for example, by scope or by being components of the same aggregate—to form a larger storage area. Each node is labeled with an identifier that is not necessarily unique. Its unique name is derived by concatenating all the identifiers along the path from the root to the node itself. For example, in the following code

```
int f ()
{
    int a;
    struct {
        int a;
        int b;
    } c;
}
```

all variables declared in function `f` can be specified by the path `<function.f>`. Individually, their names are `<function.f.a>` and `<function.f.c>`. The two fields of the variable `<function.f.c>` are `<function.f.c.a>` and `<function.f.c.b>`. In Fig-

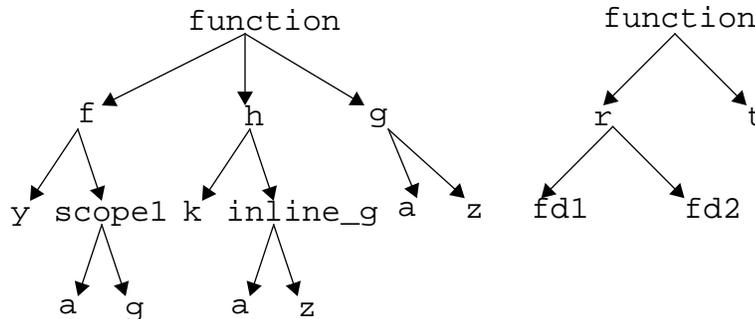


Figure 7-2 A Path Forest

The trees above represent the relationships between paths in a program. Notice that we can represent the functions within an inlined function by replicating subtrees.

ure 7-1, the path `<function.f.label.L14>` is a label and the path `<global.A>` refers to the array `A` in Figure 7-1.

A path serves not only as a unique identifier, but also succinctly encodes how the address for the variable is generated. Stored in the symbol table is the addressing relationship between an internal node of a path and each child of the node, and the code to be generated that derives the child’s address from the parent’s. For high-level phases that are not concerned with address computations, the address is represented by a path, which is treated as a constant. When necessary, the path can be expanded into a series of instructions by looking up the code corresponding to each edge in the path. This technique can be used for accesses into data structures, variables off the stack, or even up-level accesses in nested functions.

More importantly, SUIF’s hierarchical name-space design simplifies many optimizations. One example is inlining. Suppose function `g` is to be inlined at a call site in function `h`. To get the new name space, we duplicate the subtree rooted at `<function.g>`, give the root of the new subtree a unique name, and place the copy as a new subtree under `<function.h>`. The result is shown in Figure 7-2. Name clashes are automatically avoided and we do not have to give all the variables of inlined `g` obscure names. Contrast this to Ucode, which provides one level of grouping with memory blocks. Each function can only have one memory block; thus a variable would be represented as a region within such a block. Ucode instructions would reference the variables using an index and a length. For inlining, Ucode must merge the variables of the callee and the caller into one flat area, thus losing high level information.

Paths simplify alias analysis. Our approach with alias analysis is modeled after the

resource IDs of Coutant [23]. Alias analysis[2] tracks each variable's *reference set*, the set of addresses it can point to. A variable is aliased if its address is in a reference set of a pointer used as a memory address of a load or store. Computing these sets exactly is undecidable, so the reference sets will be conservative.

Reference sets can be succinctly represented as sets of paths in SUIF. If a register, for example, points to a record $\langle \text{global} . r \rangle$, then it is not necessary to enumerate all the fields of the record in r . The reference set is simply $\{\langle \text{global} . r \rangle\}$. Paths are particularly convenient for large reference sets: The set of all global variables is $\{\langle \text{global} . r \rangle\}$. Enumerating all the global variables can be very inefficient. An example of a need to describe all the global variables is the reference set of a procedure call. When no interprocedural analysis is performed, the conservative reference set for a function is often all the globals.

7.1.3 High-SUIF

The previous sections described low-SUIF, a set of features for both the high- and low-level phases of the SUIF compiler. High-SUIF is SUIF when it has the following high-level constructs:

- An `ARRAY` instruction explicitly indicates each index expressions of array reference. An array instruction appears on Line 19 in Figure 7-1. The `ldc` instruction on line 18 loads into node register `nr#7` the base address of the array. Node register `nr#6` holds the index value, the constant 200; `nr#5` holds the upper bound 300; and `nr#8` holds the computed address of the indexed element.
- A `FOR` construct represents a FORTRAN DO-loop. In Figure 7-1, the lines beginning with the character `*` are the parts of the `FOR` construct. The `begin` line tells us that `pr#1`, `pr#3`, `pr#4` and `pr#5` are the loop variable, lower bound, upper bound, and step respectively. The bounds and step registers are compiler generated, hence invisible to the programmer of the source program and cannot be modified within the loop body, as required under FORTRAN DO-loop semantics. The string `slte` indicates the loop test is `pr#1 ≤ pr#4`. The two labels are the *break* and *continue* labels. Break and continue statements inside the loop would be translated by the front-end into jumps to these labels. The lower bound computation follows the loop `begin` marker; the `ub` marker precedes the upper bound calculation; and the `step` maker precedes the step initialization code. Finally, the `body` and `end` lines mark the boundaries of the loop body.
- An `IF` construct has three lists of instructions corresponding to the test, the *then* part and the optional *else* part.

- A `LOOP` construct contains two lists of instructions, one representing the body and one representing the test; the test implicitly occurs at the bottom of the loop body. We represent C for-loops not matching FORTRAN DO-loop semantics with `LOOP` construct surrounded by an `IF` construct. The outer `IF` decides whether the loop should execute at least once, while the test at the end of the `LOOP` construct decides whether to execute the loop again.

Not all control-flow within high-SUIF is represented by our constructs; unstructured control-flow is represented with the usual low-level jumps, branches and labels.

Our front end translates source programs to high-SUIF directly. (We also use a preprocessor that restructures Fortran into structured code). We have found that our high-SUIF constructs, array-indexing and loop structures, are convenient and sufficient for data-dependence analysis and loop transformations, without sacrificing any functionality. The fields of the `ARRAY` instruction provide the arguments for dependence testing. The control-flow constructs delimit the code regions of interest to loop transformations. Loops can be permuted by updating the fields of the `FOR` constructs; loops can be distributed by surrounding new loop bodies with `FOR` constructs.

After analysis and optimizations on high-level constructs, the *expander* phase lowers the high-level constructs to low-level SUIF. Following the approach of Harrison [5, 41] and of Auslander [11], our expander macro replaces the high-level constructs with lower level code using a code template. Each construct can be lowered independently. The expander makes no attempt to generate tight, fast code. Instead, later optimizations tune and customize the result to fit the surrounding code. For example, when expanding an array instruction representing `a [3] [4]`, the expander does not fold any constants; it is done later by a constant-folding and propagation phase.

Because high-SUIF includes low-SUIF as a subset, we have built many compiler passes that accept high-SUIF by extending passes that accept low-SUIF. Most of our scalar optimizations fit this model and will accept high-SUIF. For instance, we obtained a high-SUIF reaching-definitions analyzer by including, in the data flow analyzer, knowledge about how the `FOR` construct generates and uses definitions. Except for those operations hidden by the high-level constructs, our scalar phases are equally effective on both high-SUIF and low-SUIF.

7.2 Annotations

Annotations in SUIF offer a flexible way of transmitting information (e.g. data dependence results) between phases of the compiler. Analyzers attach annotations to instructions, procedures, data declarations or control constructs; and program transformers look for and interpret specific annotations, leaving other annotations intact for subsequent passes. A par-

allelizing pass, for example, can annotate a FOR loop as a DOALL, indicating the loops that need to be implemented with parallel multiprocessor code. Likewise, an interprocedural analysis tool can annotate a procedure and call sites with summary information. Currently, the phases of the SUIF compiler tell each other a wide variety of information—such as ud-chains, symbol table information, and data dependences—with annotations.

An annotation consists of a name, a string, followed by a list of fields. Each field can be a string, register, integer, SUIF type or path. Any specific annotation has a known format agreed upon by the compiler phases producing and consuming the annotations. For example, if an alias analysis phase determines that a store instruction could write any global or the function *f*'s local variable *r*, it might attach the following annotation to the store instruction:

```
[ "alias set" <global> <function.f.r> ]
```

Subsequent passes look for the “alias set” annotation on loads and stores and use this information. In general, the annotation format is simple and flexible. Thus it is easy to add new information into SUIF, an important property for a research compiler.

Annotations allow us to separate analysis phases of the compiler from transformation phases, improving the modularity of the compiler. The separation permits the substitution of analysis phases, a capability that is invaluable for experimental purposes.

7.3 Overview and Status of the SUIF compiler system

This section shows how the SUIF compiler has been organized to take advantage of the integration of optimizations and parallelizations. The SUIF compiler separates its functions into distinct phases which communicate only through SUIF files. Because SUIF has only one representation, we are free in ordering the phases with respect to each other. We have found this structure to be invaluable for experimentation, permitting several implementation projects to coexist.

Figure 7-3 shows the progress of a typical FORTRAN program as it flows through the SUIF compiler. The first phase *f2c*, a FORTRAN-to-C converter [36] translates the program to an equivalent C program. Then the program *FE*, a front-end based on the portable C compiler, translates the C program to a high-SUIF program.

During parallelization, the program representations have the features of high-SUIF. The high-SUIF program is passed through a sequence of phases. This sequence includes not only parallelizing phases, but also includes scalar optimizations, each providing information or preparing the program for later phases in the sequence. Which phases are present and how they are ordered in the sequence will depend on the goal of the compilation—compiling for multiprocessors and compiling for superscalar machines require different sequences. They will also depend on the information requirements of the phases—some

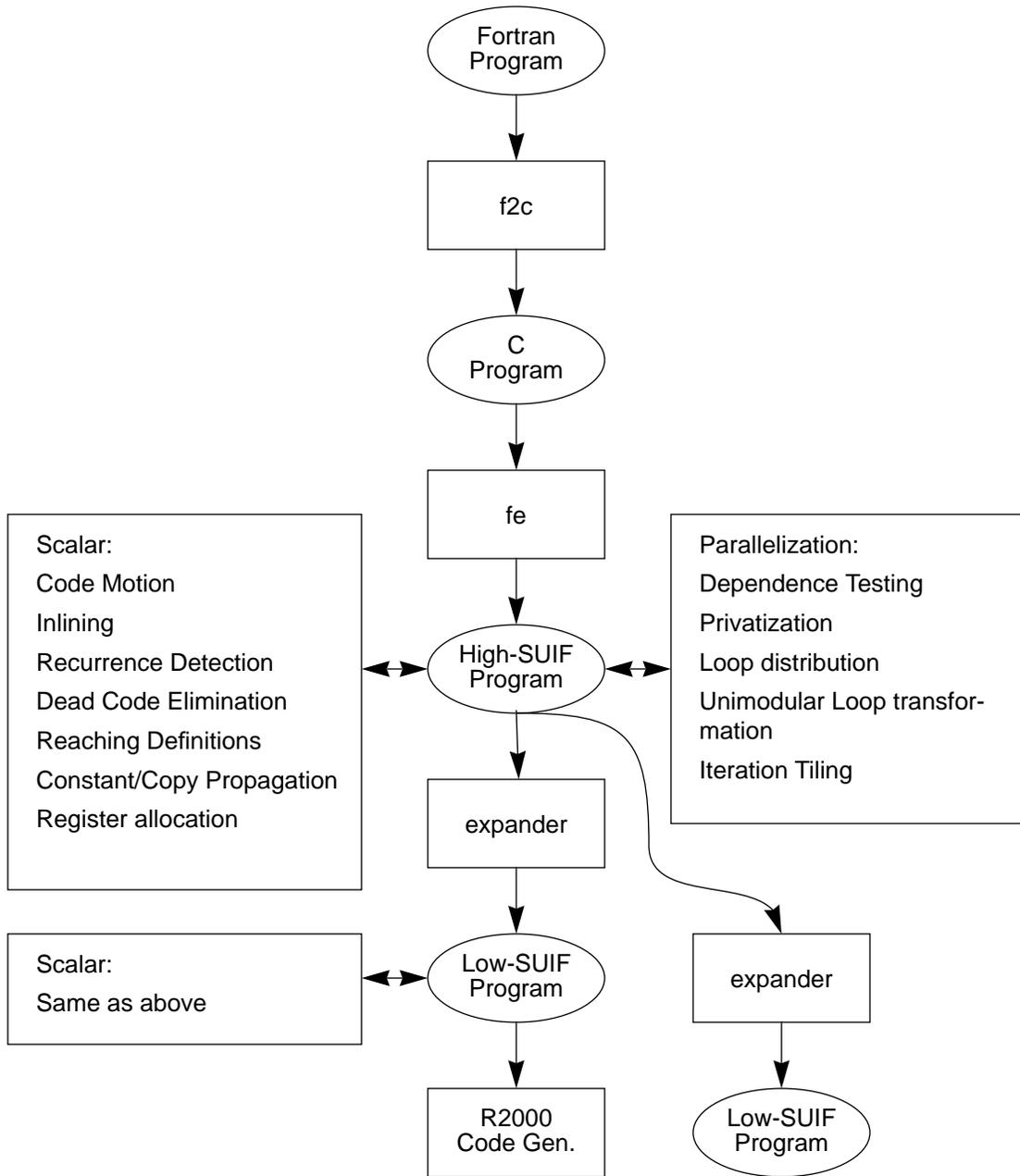


Figure 7-3 The SUIF compiler system

phases (dependence analysis) work better if others (constant propagation, induction variable expansion) precede them in the sequence. The output of the sequence is another program in high-SUIF that has been parallelized and improved by scalar optimizations.

Then the expander translates the high-level constructs of the high-SUIF program into low-SUIF (See Section 7.1.3) The low-SUIF program is optimized and customized by scalar phases, many the same as those previously applied to high-SUIF.

Eventually, low-SUIF is translated into target code. We have developed code generators for the Mips R2000 and the VAX.

Our compiler has a comprehensive repertoire of transformations and analyses, which we now discuss. The scalar optimizations have already been described in Sections 6.1.

7.3.1 High-level Transformations and Analysis

Array- and loop-level analyses and transformations require data-dependence information. By arranging dependence tests in increasing strengths and costs, and using memoization, the SUIF compiler can generate exact dependence information efficiently [64]. This dependence information can be used in low-level passes such as a superscalar/VLIW instruction scheduler, and in high-level passes that privatize scalars, distribute and transform loops.

Loop optimizations in the SUIF compiler are based on unified unimodular transformations—interchange, reversal, skewing—together with a new transformation theory [93, 94]. The theory shows how these transformations, together with tiling (or blocking), can expose parallelism and improve data locality, without introducing excessive communication or over-committing caches. This theory simplifies implementation, finding directly the optimal loop transformations and computing directly the loop bounds without searching exhaustively through the transformation space.

In addition to unimodular transformations, the SUIF compiler also contains several well-known transformations: loop distribution, scalar privatization and scalar expansion. Scalar privatization is important because it allows loops which define scalars for every iteration to be parallelized. Loop distribution and scalar expansion work in concert to allow sequential portions of the loop to be split from parallel computations in a loop.

7.3.2 Status and Experience

Over the last four years, the SUIF compiler has become an effective research compiler, a compiler that can be extended to run new experiments, a compiler capable of generating good optimized and parallelized code. We can compile and validate nine of the Perfect club benchmarks with full scalar optimizations. The high-level phases are functioning. Data-dependence tests will run on all the Perfect club [64]. Linear loop transformations can auto-

matically block matrix algorithms such as QR decomposition and LU decomposition without pivoting, whose performance we have measured on an SGI multiprocessor [94].

8

Conclusion

This dissertation has made two contributions to the building optimizing compilers. The first is a software architecture with which to write scalar optimizers that are modular and extensible. The second is how we can integrate scalar optimizations—that require a low-level program representation—with other code transformations that require a high-level program representation.

The optimizer architecture is based on Kildall’s original proposition of using data-flow analysis as a basis for describing optimizations [54]. To simplify the construction of data-flow analyzers and to encourage the use of Kildall’s model, a tool called Sharlit has been provided. The key features of Sharlit are:

- Sharlit separates the description of the DFA problem—the flow values and the flow functions—from the description of how the problem is to be solved. Thus the compiler-writer may use the same flow values and flow functions with several solution techniques. This independence promotes flexibility in choosing a solution technique.
- Sharlit provides one DFA solution procedure—path simplification—that unifies several common solution procedures used in existing optimizers. This dissertation has shown how local analysis, Tarjan interval analysis and sparse data-flow evaluation graphs can be implemented with Sharlit. Using path simplification decreases the effort required to build an optimization. Another advantage of path simplification is that it permits the compiler writer to pose DFA problems without the need to consider local analysis and basic blocks. These concepts complicate many DFA problems such as partial redundancy elimination. Instead, path simplification is used to make the data-flow analyzer more efficient.

- Sharlit provides facilities for combining data-flow analyzers. The facilities enable the create of complex analyzers from simpler ones and increase the modularity of the optimizer.

Sharlit has been tested by using it to construct a complete optimizer. To the best of our knowledge, this dissertation is the first to realize Kildall's proposition fully within a working optimizer that compiles real, large programs.

The optimizer is modular and extensible because its phases are data-flow analyzers that can be reordered and because new phases are relatively easy to add. The reordering and addition of phases is facilitated by the ability to add new flow functions and path simplification rules. The optimizer is also modular because its analyzers can built from simpler ones using chains and products. This optimizer generates efficient code, nearly as efficient as one of the best commercial optimizing compilers.

The second component of this dissertation is how we can integrate scalar optimizations (that require a low-level program representation) with other code transformations that require a high-level program representation. Optimizing compilers for modern high-performance computers must implement not just traditional scalar optimizations, but also array level analyses and loop transformations. Such an optimizing compiler is a very large system. Moreover, to keep up with the rapid pace of processor architecture advances, the compiler system must be capable of incorporating new optimizations developed for specific architectures. Thus, it is even more important that the system be modular and well-engineered to support growth and experimentation. To fulfil these needs, this thesis proposed an intermediate language SUIF with the following features:

- A register naming convention that helps high-level phases to identify expression trees.
- A technique for encoding high-level control-flow construct in a manner that makes them amenable to both scalar optimizations and high-level transformations.
- A naming convention for variables that permit it easy to do things like inlining and that encodes the high-level storage relationships between the variables.
- An annotation mechanism that allows a phases to communicate a wide variety of information.

The SUIF compiler system is based on this intermediate format. The compiler system consists of a set of tools, ranging from scalar data flow optimizations to compound loop transformations. The compiler has been under development and in use at Stanford for over four years. The SUIF experience suggests that the intermediate language design does provide a useful platform for multiple simultaneous projects on a variety of compiler research topics. Using the ability to share phases inherent in SUIF, we have, in a reasonable time frame obtained interesting results in individual topics such as data dependence analysis [64], parallelization [68] and loop transformations [92, 93]. There are still many subjects to be

explored with the SUIF platform. The SUIF project continues to use this comprehensive compiling system to gather meaningful statistics on complete programs, experiment with new architectural designs, and evaluate new compilation techniques.

8.1 Future Work

Sharlit and the SUIF optimizer has several limitations whose resolution would widen the applicability and the usefulness of Sharlit and the SUIF optimizer. The following sections address the following issues:

- facilities to build flow values.
- algorithms for more efficient iteration.
- program representations other than control-flow graphs.
- mechanisms to update data-flow information incrementally.

Furthermore, more experience should be obtained by implementing optimizations using the Sharlit architecture. This experience will give a better idea of the overhead incurred due to the increased modularity and flexibility.

8.1.1 Flow Values

Implementing an analyzer to solve a DFA problem with Sharlit requires writing flow functions and simplification rules, and implementing the flow values. Sharlit provides good facilities for the first two but has none for implementing flow values. Adding libraries to help with implementing such objects as sets, expression trees, graphs, simple symbolic manipulation routines would hasten the development of analyzers. Ideally these libraries would be independent of the internal program representation.

8.1.2 Efficient Iteration

The iteration algorithm used in Sharlit evaluates every node even when the flow values that enter a node have not changed. This is the technique used by Hecht and Ullman [43]. A more efficient algorithm, proposed by Horwitz, Demers and Teitelbaum [46], would use the header forest—hence flow graph regions—and change status of nodes to reduce the number of node evaluations. While the worst case running-time for both algorithms are identical, it is expected that Horwitz's algorithm will have better average running-time. Moreover, the algorithm would incur minimal overhead as Sharlit already computes the header forest and change status.

8.1.3 Flow Graphs and Incremental Updates

The primary focus of Sharlit has been on control-flow graphs. In such graphs, control-flow is explicitly represented as edges in the graph, while data-flow relationships, data-dependences, and control-dependences must be implicitly represented as part of the flow values. While this focus on CFG suits many optimizations, a uniform representation for all dependencies can benefit many optimizations. For example, several powerful techniques [4, 8, 12, 26, 29, 73, 89] are formulated on graphs in which control-dependence and data-flow are uniformly represented. Although Sharlit can support these techniques through flow values abstractions, it would be interesting to see how Sharlit can support these techniques more directly as Sharlit can do with sparse data-flow evaluation graphs.

High-level analyses and transformations are generally applied to abstract syntax trees rather than control-flow graph. As discussed in Chapter 7 many of these analyses and transformations require data-flow analyses and scalar optimizations to be applied to the program beforehand. The approach taken in the SUIF compiler is to run scalar optimizations before and after high-level transformations. While this approach is very flexible, it is inefficient as the program has to be written out as a SUIF intermediate file several times. To reduce intermediate file traffic, we can build control-flow graphs directly from the tree and apply data-flow analyzers directly on the graphs.

Integrating Sharlit within a system that uses parse trees introduces two interesting possibilities. The parse tree can be used to speed up control-flow analysis (CFA). For example, CFA as described in Section 5.1.2.1 essentially re-discovers the loop structure of a program. This information is already evident in a parse tree. Furthermore, other structures such as if-then-else can participate in CFA. At the limit, we can let the parse tree of a program define all regions during CFA. This leads us to the natural interpretation of viewing the data-flow analysis as attribute evaluation in an attributed parse tree.

This synergism between attribute grammars and data-flow analysis may allow us to apply efficient incremental update algorithms used in attribute evaluation [72] to data-flow analysis. This idea was first put forth by Carroll and Ryder [18, 74]. While attribute evaluation has been applied to solving DFA problems [72], that work has largely not taken advantage of the techniques developed in traditional data-flow analysis. Integrating Sharlit into an attribute evaluation system would be interesting.

8.1.4 More Experience with Sharlit

Although this dissertation has shown that it is possible to build an effective optimizer with Sharlit, the SUIF optimizer did not fully test many of Sharlit's facilities, especially those used to combine DFA solvers. More optimizations should be implemented to test these features. Moreover, we should try to use these facilities to improve the speed of the SUIF opti-

mizer. Some phases do not take advantage of Sharlit's facilities to combine optimizations. For example, constant propagation cannot be run after code motion at present without writing out a SUIF intermediate file after code motion and re-invoking the optimizer to propagate constant. The overhead in using an intermediate file is quite large and makes the SUIF optimizer about ten times slower than the MIPS optimizer. We can remove this overhead by modifying the constant propagation phase to accept the output of the code motion algorithm, thus eliminating the intermediate files. We believe that the modification can be performed within the current Sharlit architecture and should make the SUIF optimizer much faster. Additional run-time overhead will still exist because of the abstract interfaces in the architecture. However, we believe that this overhead is tolerable in light of the flexibility provided by Sharlit.

Sharlit's flexibility and extensibility has made the SUIF compiler a good vehicle to investigate different scalar optimizations and how the optimizations interact. The SUIF compiler project at Stanford is continuing these investigations.

Bibliography

1. A. V. Aho, M. Ganapathi, and S. W. Tjiang. "Code generation using tree matching and dynamic programming". *ACM Trans. on Programming Lang. and Systems* 11, 4 (October 1989), 491-516.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.
3. F. E. Allen. A basis for program optimization. In *IFIP Congress*, 1971, pp. 385-390.
4. F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. Tech. Rept. Technical Report RC 13115, IBM, Sept, 1987.
5. F. E. Allen, J. L. Carter, J. Fabri, J. Ferrante, W. H. Harrison, P. G. Loewner, and L. H. Trevillyan. "The experimental compiling system". *IBM Journal of Research and Development* 24, 6 (November 1980).
6. F. E. Allen, J. Cocke, and K. Kennedy. Reduction of Operator Strength. In S. S. Muchnick and N. D. Jones, Ed., *Program Flow Analysis: Theory and Applications*, Prentice-Hall 1981, pp. 79-101.
7. R. Allen and S. C. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1988.
8. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th ACM Symposium on Principles of Programming Languages*, 1988.
9. Z. Ammarguellat and W. L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1990.
10. P. Anklam, D. Cutler, R. Heinen Jr., and M. D. Maclaren. *Engineering a compiler: VAX-11 Code Generation and Optimization*. Digital Equipment Corporation, 1982.

11. M. Auslander and M. Hopkins. An overview of the PL.8 compiler. In *SIGPLAN Conference on Compiler Construction*, 1982.
12. R. A. Ballance, A. B. Maccabe, K. J. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1990, pp. 257-271.
13. U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.
14. M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1988.
15. M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: effective performance evaluation of supercomputers. Tech. Rept. Technical Report (UIUCSRD) 827, University of Illinois, Urbana-Champaign. Center for Supercomputing Research and Development, 1989.
16. B. M. Brosgol, J. M. Newcomer, D. A. Lamb, D. R. Levine, Mary S. Van Deusen, and William A. Wulf. TCOL_Ada: Revised Report on An Intermediate Representation for the Preliminary Ada Language. Tech. Rept. Technical Report CMU-CS-80-105, Carnegie Mellon University, February, 1980.
17. M. Burke and B. G. Ryder. Incremental Iterative Data Flow Analysis Algorithms. Tech. Rept. Technical Report LCSR-TR-96, Rutgers University, Laboratory for Computer Science Research, August, 1987.
18. M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *15th ACM Symposium on Principles of Programming Languages*, 1988.
19. J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graph. In *18th ACM Symposium on Principles of Programming Languages*, 1991.
20. F. C. Chow. *A portable Machine-Independent Global Optimizer --- Design and Measurement*. Ph.D. Th., Stanford University, 1983.
21. F. C. Chow. Register Allocation by Priority-based Coloring. In *SIGPLAN Conference on Compiler Construction*, 1984.
22. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, January, 1977, pp. 238-252.

23. D. S. Coutant. Retargetable high-level alias analysis. In *13th ACM Symposium on Principles of Programming Languages*, 1986.
24. D. S. Coutant, C. L. Hammond, and J. W. Kelly. "Compilers for the new generation of Hewlett-Packard computers". *Hewlett-Packard Journal* 37, 1 (January 1986).
25. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. "An efficient method of computing static single assignment form". *ACM Trans. on Programming Lang. and Systems* 13, 4 (October 1991), 451-490.
26. R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. In *ACM Symposium on Principles of Compiler Construction*, 1986.
27. J. W. Davidson and C. W. Fraser. "Code selection through object code optimization". *ACM Trans. on Programming Lang. and Systems* 6, 4 (October 1984), 505-526.
28. C. D. Farnum. *Pattern-Based Languages for Prototyping of Compiler Optimizers*. Ph.D. Th., University of California, Berkeley, 1990.
29. J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The program dependence graph and its use in optimization". *ACM Trans. on Programming Lang. and Systems* 9, 3 (July 1987), 319-349.
30. A. Fong, J. Kam, and J. Ullman. Application of lattice algebra to loop optimization. In *2nd ACM Symposium on Principles of Programming Languages*, January, 1975, pp. 1-9.
31. C. W. Fraser and A. L. Wendt. Integrating code generation and optimization. In *SIG-PLAN Conference on Compiler Construction*, 1986.
32. M. Ganapathi. *Retargetable Code Generation and Optimization using Attribute Grammars*. Ph.D. Th., University of Wisconsin-Madison, 1980.
33. M. Ganapathi and C. N. Fischer. Description-driven code generation using attributed grammars. In *9th ACM Symposium on Principles of Programming Languages*, 1982.
34. M. Ganapathi and C. N. Fischer. "Attributed linear intermediate representations for retargetable code generators". *Software --- Practice and Experience* 14, 4 (April 1984), 347-364.
35. M. Ganapathi, C. N. Fischer, and J. L. Hennessy. "Retargetable compiler code generation". *ACM Computing Surveys* 14, 4 (1982), 573-592.
36. D. Gay. Private Communication. ATT Software.
37. P. Gerring, P. Nye, A. Rodriguez, and A. Samuel. A Universal P-Code for the S-1 Project. Tech. Rept. CSL Technical Note 159, Stanford University, Aug, 1979.

38. R. S. Glanville. *A Machine-Independent Algorithm for Code Generation and Its Use in Retargetable Compilers*. Ph.D. Th., University of California, Berkeley, 1978.
39. S. Graham and M. Wegman. "Fast and usually linear algorithm for global flow analysis". *J. ACM* 23, 1 (January 1976), 172-102.
40. D. R. Grundman. *Graph Transformations and Program Flow Analysis*. Ph.D. Th., University of California, Berkeley, 1990.
41. W. Harrison. A new strategy for code generation --- the general purpose optimizing compiler. In *ACM Fourth Symposium on Principles of Programming Languages*, 1977, pp. 29-37.
42. M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.
43. M. S. Hecht and J. D. Ullman. "A simple algorithm for global data-flow analysis problems". *SIAM J. Comput.* 4, 4 (December 1975), 519-532.
44. R. R. Henry. *Graham-Glanville code generators*. Ph.D. Th., University of California, Berkeley, 1984.
45. R. R. Henry. The CODEGEN user's manual. Tech. Rept. Technical Report 87-08-04, University of Washington, 1987.
46. S. Horwitz, A. Demers, and T. Teitelbaum. "An efficient general iterative algorithm for dataflow analysis". *Acta Inf.* 24 (1987), 679-694.
47. S. C. Johnson. A Portable compiler: theory and practice. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, 1978, pp. 97-104.
48. S. C. Johnson. YACC--yet another compiler compiler. Tech. Rept. CSTR-32, Bell Laboratories, 1975.
49. M. S. Johnson and T. C. Miller. Effectiveness of a machine-Level, global optimizer. In *SIGPLAN Conference on Compiler Construction*, 1986.
50. S. M. Joshi and D. M. Dhamdhere. "A composite hoisting-strength reduction transformation for global program optimization: part I". *Intern. Journal of Computer Math.* 11 (1982), 21-41.
51. S. M. Joshi and D. M. Dhamdhere. "A composite hoisting-strength reduction transformation for global program optimization: part II". *Intern. Journal of Computer Math.* 11 (1982), 111-126.
52. J. B. Kam and J. D. Ullman. "Global data flow analysis and iterative algorithms". *J. ACM* 23, 1 (January 1976), 158-171.

53. J. B. Kam and J. D. Ullman. "Monotone data flow analysis frameworks". *Acta Inf.* 7 (1977), 305-317.
54. G. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principle of Programming Languages*, 1973, pp. 194-206.
55. J. Knoop, O. Rething, and B. Steffen. Lazy Code Motion. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1992, pp. 224-234.
56. W. R. Lalonde and J. des Rivieres. A flexible compiler structure that allows dynamic phase ordering. In *SIGPLAN Conference on Compiler Construction*, 1982.
57. M. S. Lam. *A Systolic Array Optimizing Compiler*. Ph.D. Th., Carnegie-Mellon University, May 1987. Also available as CMU-CS-87-187.
58. D. A. Lamb. *Sharing intermediate representations: The interface description language*. Ph.D. Th., Carnegie-Mellon University, May 1983. Also available as CMU-CS-83-129.
59. T. Lengauer and R. E. Tarjan. "A fast algorithm for finding dominators in a flow-graph". *ACM Trans. on Programming Lang. and Systems* 1, 1 (July 1979), 121-141.
60. M. E. Lesk. LEX--a lexical analyzer generator. Tech. Rept. CSTR-39, Bell Laboratories, 1975.
61. S. MacLane and G. Birkhoff. *Algebra*. MacMillan Company, 1967.
62. T. J. Marlowe and B. G. Ryder. Properties of Data Flow Frameworks: A Unified Model. Tech. Rept. Technical Report LCSR-TR-103, Laboratory for Computer Science Research, Rutgers University, April, 1988.
63. H. Z. Marshall. The linear graph package, a compiler building environment. In *SIGPLAN Conference on Compiler Construction*, 1982.
64. D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence testing. In *SIGPLAN Conference on Programming Language Design and Implementation*, June, 1991.
65. E. Morel and C. Renvoise. "Global optimization by suppression of partial redundancies". *Comm. ACM* 22, 2 (February 1979), 96-103.
66. K. J. Ottenstein. *Data-Flow Graphs as An Intermediate Program Form*. Ph.D. Th., Purdue University, 1978.
67. D. L. Perkins and R. L. Sites. Machine-independent pascal code optimization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 1979, pp. 201-207.

68. K. L. Pieper. *Parallelizing Compilers: Implementation and Effectiveness*. Ph.D. Th., Stanford University, 1993. In preparation.
69. J. R. Reif. "Code motion". *SIAM J. Comput.* 9, 2 (May 1980).
70. J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *4th ACM Symposium on Principles of Programming Languages*, January, 1977, pp. 104-118.
71. J. H. Reif and H. R. Lewis. Efficient Symbolic Analysis of Programs. Tech. Rept. Technical Report TR-37-82, Harvard University, Aiken Computation Laboratory, 1982.
72. T. W. Reps. *Generating Language-Based Environments*. The M.I.T. Press, 1984.
73. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value number graphs and redundant computations. In *15th ACM Symposium on Principles of Programming Languages*, 1988.
74. B. G. Ryder. Incremental data flow analysis. In *10th ACM Symposium on Principles of Programming Languages*, January, 1982, pp. 167-176.
75. B. G. Ryder, T. J. Marlowe, and M. C. Paull. Incremental Iteration: When Will It Work?. Tech. Rept. Technical Report LCSR-TR-89, Rutgers University, Laboratory for Computer Science Research, March, 1987.
76. B. G. Ryder and M. C. Paull. "Elimination algorithms for data flow analysis". *Comput. Surv.* 18, 3 (September 1986), 277-316.
77. J. T. Schwartz and M. Sharir. A Design for Optimizations of the Bitvectoring Class. Tech. Rept. NSO-17, Courant Institute of Mathematical Sciences, 1979.
78. M. Sharir. "Structural analysis: a new approach to flow analysis in optimizing compilers". *Computer Languages* 5 (1980), 141-153.
79. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
80. R. E. Tarjan. "Finding dominators in directed graphs". *SIAM J. Comput.* 3, 1 (March 1974).
81. R. E. Tarjan. "Testing flow graph reducibility". *J. of Comput. and Syst. Sci.* (9 1974), 355-365.
82. R. E. Tarjan. "Fast algorithm for solving path problems". *J. ACM* 28, 3 (July 1981), 594-614.

83. R. E. Tarjan. "A unified approach to path problems". *J. ACM* 28, 3 (July 1981), 577-593.
84. S. W. K. Tjiang and J. L. Hennessy. Sharlit: A tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1992, pp. 82-93.
85. S. W. K. Tjiang, M. E. Wolf, M. S. Lam, K. L. Pieper, and J. L. Hennessy. Integrating scalar optimization and parallelization. In *Fourth Workshop on Languages and Compilers for Parallel Computing*, Aug., 1991, pp. 137-151.
86. J. D. Ullman. "Fast algorithms for the elimination of common subexpressions". *Acta Inf.* 2, 3 (July 1973), 191-213.
87. G. A. Venkatesch. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1989, pp. 1-12.
88. H. S. Warren, Jr., M. A. Auslander, G. J. Chaitin, A. C. Chibib, M. E. Hopkins, and A. L. Mackay. Final Code Generation in the PL.8 Compiler. Tech. Rept. Technical Report RC 11974, IBM Thomas J. Watson Research Center, June, 1986.
89. M. Wegman and K. Zadeck. Constant propagation with conditional branches. In *12th ACM Symposium on Principles of Programming Languages*, 1985.
90. E. B. White. *Charlotte's Web*. Harper and Row, 1952.
91. D. Whitfield and M. L. Soffa. Automatic Generation of Global Optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1991, pp. 120-129.
92. M. E. Wolf. *Improving Parallelism and Data Locality in Nested Loops*. Ph.D. Th., Stanford University, June 1992.
93. M. E. Wolf and M. S. Lam. An algorithmic approach to compound loop transformations. In *Third Workshop on Languages and Compilers for Parallel Computing*, Aug., 1990.
94. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

