RICE UNIVERSITY

Interprocedural Symbolic Analysis

by

Paul Havlak

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Ken Kennedy, Noah Harding Professor
Computer Science
Chair

Keith D. Cooper, Associate Professor
Computer Science

Robert S. Cartwright, Professor
Computer Science

John Bennett, Assistant Professor
Electrical and Computer Engineering

Houston, Texas

May, 1994

Abstract

Interprocedural Symbolic Analysis

by Paul Havlak

Compiling for efficient execution on advanced computer architectures requires extensive program analysis and transformation. Most compilers limit their analysis to simple phenomena within single procedures, limiting effective optimization of modular codes and making the programmer's job harder. We present methods for analyzing array side effects and for comparing nonconstant values computed in the same and different procedures.

Regular sections, described by rectangular bounds and stride, prove as effective in describing array side effects in LINPACK as more complicated summary techniques. On a set of six programs, regular section analysis of array side effects gives 0 to 39 percent reductions in array dependences at call sites, with 10 to 25 percent increases in analysis time.

Symbolic analysis is essential to data dependence testing, array section analysis, and other high-level program manipulations. We give methods for building symbolic expressions from gated single-assignment form and simplifying them arithmetically. On an suite of 33 scientific Fortran programs, symbolic dependence testing yields reductions of 0 to 32 percent in the number of array dependences, as compared with constant propagation alone. The additional time and space requirements appear proportional to the size of the codes analyzed.

Interprocedural symbolic methods are essential in enabling array section analysis and other advanced techniques to operate on multiple procedures. Our implementation provides this support while allowing for recompilation analysis to approximate the incrementalism of separate compilation. However, direct improvement of data dependence graphs from interprocedural symbolic facts is rare in the programs studied.

Overall, the use of our symbolic techniques in a production compiler is justified by their efficiency, their direct enhancement of dependence testing within procedures,

and their indirect improvement of interprocedural dependence testing through array side effect analysis.

Acknowledgments

As the prospect of finishing my dissertation begins fully to sink in, I realize again that I would never have gotten this far without the help, good wishes and high expectations of all my family, friends, and colleagues.

My wife, Lisa Gray, has been more patient than I deserve during my graduate career, and I would be lost without her continuous love and support. I thank my parents and all the Havlaks for being there when I need them.

I owe a great debt to my teachers over the years, especially to Wilma Hill and Ronda McNew for giving me my first taste of rigorous science.

Rice and its Department of Computer Science have been my home for a long time, and I am eternally grateful for the congenial atmosphere and for the inspiring colleagues I have found there. The team efforts of the PFC and ParaScope projects and the shared trials of graduate school have solidified several cherished friendships.

Finally, I would like to thank my advisor, Ken Kennedy, and the rest of my thesis committee for seeing me through graduate school. I particularly appreciate the extensive feedback provided by John Bennett. Thanks also to the many others who have helped me improve the syntax and semantics of this document.

To all these, and my patient readers, thanks.

Contents

Illustrations

Tables

Algorithms

Chapter 1

Introduction

In the past decade, the computer power available to scientific programmers has expanded greatly. Part of this gain comes from the continuing speed improvements of single-chip microprocessors. But much of the gain, especially for the most computationally expensive programs, has come from employing many processors in parallel.

Computer architects have discovered how to connect multiple processors with high-speed memory and communication networks. Application programmers have found ways to make these processors cooperate on single problems. The challenge remains to make the same program solve these problems on different parallel architectures. As things stand, a program originally written for a mainframe computer might have been rewritten several times for novel architectures over the past two decades:

- in the mid-1970s, with vectorizable DO-loops for a Cray-style vector processor
- in the early 1980s, with multitasking for parallel vector big iron (two to four processors)
- in the mid-1980s, with DOALL loops for shared-memory multiprocessors (eight to 24 processors connected by bus or switch)
- in the late 1980s, using array section notation for a SIMD parallel computer (such as the CM-1, CM-2, or Maspar)
- today, with explicit message-passing for distributed-memory computers

These are a few of the general architectural categories. Different programming support on machines of similar architecture lead to additional reprogramming, a particular problem given the high rate of attrition among vector and parallel computer vendors.

Mere portability is no solution to this Sisyphean task. Demanding programmers will insist on hand-tuning their programs for specific machines so long as they achieve speedups. Computationally intensive applications need *portable performance*, which must be supplied through expressive languages, ingenious run-time support, and sophisticated compilers.

Compilers play many roles in this task:

- For a clearly written sequential program, relying primarily on array computations, different compilers may be able to produce good code for the most of parallel architectures listed above.
- For an explicitly parallel program, written with parallel language primitives or added libraries, compilers can be important to optimizing both the parallel synchronization and communication and the sequential computation.
- For any program, compilers must optimize the sequential fragments inside and in between the parallel components. Because of Amdahl's Law, slow execution of sequential components can greatly limit the gains of parallelism.

While language support for machine-independent programming is hard to provide, it is the preferred solution. It would allow continued flexibility in computer architectures and would simplify the programming process by hiding machine details, so that programmers can focus on the application problem.

This thesis presents work on *symbolic analysis*: compile-time methods for reasoning about program values that may not be constant. While symbolic analysis is only a piece of the language support puzzle, it is fundamental. Most useful programs have important computations governed by non-constant values, which must be analyzed to determine what the program does before it can be transformed. Symbolic analysis is essential to successful dependence analysis of realistic programs. Our techniques have proven helpful in several high-level compilation methods, including optimization of run-time support.

1.1 Program Analysis and Transformation

The heart of an optimizing compiler is *analysis* of what a program does and *transformation* of the program to do it more efficiently, using less time or storage. Recent years have brought a number of challenges.

Proliferation of Uses

Program analysis and transformation are not just for compilers any more; furthermore, compilers are asked to generate more flexible code.

- **Run-time support.** Run-time analysis involves the insertion of code, perhaps by the compiler, that answers questions about the program's meaning as it runs. Increasing use of run-time techniques, especially for parallelization, create more opportunities for optimization. Examples include data race detection and scheduling of run-time data-dependent loops [HKMC90, DSvH93].
- **Interactive tools.** The analysis used for compilation can also help the authors and maintainers of programs. Analysis and transformation must be cleanly separated; a programmer will be unhappy if the smart editor changes the program, even by just propagating constants, without being asked [KMT91a].

Larger-scale transformations

Continually improving processor technology has made it harder to build memories with high enough bandwidth and low enough latency to keep the processor supplied with data. Computer architects address this problem in several ways. For a single processor, levels of cache can be added between the processor and main memory, giving the illusion, at least for some access patterns, that the large main memory has much lower latency. In a distributed-memory system, several microprocessors, each with its own local memory, are connected through a network. The aggregate memory bandwidth to all the processors can be much higher than is possible to deliver to a single processor with current technology. (These two methods are independent and generally combined.)

Optimization for these advanced architectures requires the the hiding of memory latency and frequently the introduction of explicit parallelism. These entail extensive rewriting of loops, and often moving code across procedure boundaries as well.

Many compilers in the past have gotten by with analyzing small sequences of instruction to improve instruction selection and register allocation. Two such techniques were called *peephole* optimization, examining a small instruction window, and *local analysis*, examining straight-line instruction sequences.

Current compilers generally do some analysis of the entire procedure being compiled; this scale is generally called *global analysis*. However, since we also wish to consider the whole program, this name seems confusing. We will call this scale *sub-procedural analysis*. Once we start looking past the interface of call sites, we require new, interprocedural techniques.

Increasing procedure counts

Programs are growing larger and more modular. The former is a result of larger computer memories and more complicated applications; the latter is the attempt of overtaxed programmers to manage the complexity of their task. These trends combine to produce rapid growth of the number of procedures in a typical program.

One approach to interprocedural analysis would be to bring the whole program — complete representations of every procedure — into memory at once. This could be done while keeping the procedures separate, or some procedures could be expanded inline, replacing their call sites with the procedure body. But any approach which analyzes the whole program monolithically has disadvantages:

- Whole programs can be too large. Programs of 100,000 or one million statements are not unheard of, and they may simply not fit in any affordable amount of memory.

- Monolithic recompilation can be too slow. Even the whole program fits in memory, or we find some way that it doesn't need to, we must be careful about giving up separate compilation. In the past, the only part of rebuilding a program that required access to all the procedures was the linkage phase. We do not want every procedure to be recompiled from scratch every time a small change is made.

Advanced programming tools for today's architectures need interprocedural analysis methods that are informative, yet maintain some semblance of efficiency and separate compilation. Symbolic analysis is one area seriously in need of such methods.

1.2 Symbolic Analysis

Symbolic analysis, in its most general sense, encompasses any method capable of representing and comparing non-constant values. While methods for dealing with constants and some simple symbolics have long been used in compilers, the enlarging scope of analysis and transformation motivates a general approach.

1.2.1 Symbolic Questions

Symbolic analysis is an intermediate step in optimization; it does not improve the program directly but improves knowledge of the program's meaning to increase the power of later phases. Several components of analysis and transformation could exploit knowledge of `j`, `k`, and `n` in the following fragment.

```
subroutine foo(j,k,n)
  call init(j,k,n)
  do i = 1, n
    A(i+j) := ...
    A(i+k) := ...
  enddo
  return
end
```

Common subexpression elimination deletes redundant computations of equivalent expressions [ASU86]. If `j` and `k` can be proven equivalent, then the two subscript expressions are also the same and their value need only be computed once per iteration.

Constant propagation replaces variable references and expressions with equivalent constants [WZ91]. Interprocedural versions need ways of representing values that are not constant yet [CCKT86, GT93]. In the example, the values of `j`, `k`, and `n` passed to `init` are constant if the values passed into `foo` are constant.

Test elision removes branches and (unexecutable) dead code when comparisons and boolean expressions can be evaluated [WZ91]. If we have ($\mathtt{n} == 1$) on entry to the `do` loop, then it only executes once and can be rewritten as straight-line code. If we have ($\mathtt{n} < 1$), then the loop does not execute at all and may be deleted.

Array section analysis needs to manipulate and compare symbolic subscript expressions in order to summarize subarray accesses. The subarray modified by the loop can be described as $\mathtt{A}[(1 + \mathtt{j}) : (\mathtt{n} + \mathtt{k})]$ if we know ($\mathtt{k} \geq \mathtt{j}$); if we know nothing, the section descriptor would have to be more complicated or less precise.

Dependence testing also needs to compare symbolic subexpressions, in order to determine which array accesses overlap in memory and which are independent (and thus may run in parallel). Above, if ($|\mathtt{j}-\mathtt{k}| \geq n$), then the two assignments never hit the same location, so there are no dependences, and the loop may be parallelized.

Performance estimation can better determine which parts of a program are most computationally intensive when loop bounds can be compared to each other or to constants. If ($\mathtt{n} \gg 1$), it may be profitable to run the loop in parallel.

1.2.2 Partitioning the Problem

We divide the symbolic analysis problem into two parts to be solved separately.

Expression matching represents and compares program expressions based on their form. In a sense, this focuses on *how* expressions are computed, without caring *if* they are computed.

Predicate analysis exploits other information about values, not necessarily tied to their form of expression. Particularly important are the control decisions, error conditions, and programmer assertions that govern whether or not expressions are executed.

Why the distinction? Informally, we feel that the form of expressions captures a fundamental notion of what is computed, whereas predicate analysis deals mainly with the context in which a computation is used. Consider the following fragment:

```
read i
j = i * 2
A[j] = ...
if P then
  A[i*2] = ...
else
  A[j] = ...
endif
```

Ignoring the context of the references, the three subscript expressions clearly have the same value, so that the assignments all hit the same element of `A`. The predicate P may give us additional constraints on the subscript values, and we can see that only one of the later definitions executes. We expect to get much more benefit from analyzing the forms of expressions than from analyzing predicates. The main purpose of partitioning the problem is to keep the latter from interfering with the former.

1.3 Research Overview

1.3.1 Goals

Our goal in this work is to design and test an efficient, effective, general technique for analyzing symbolic values over whole programs.

Experimental Implementation

We have implemented the central components of our symbolic analysis method in the ParaScope programming environment for Fortran [KMT91b, KMT91a]. They form the basis for manipulation of non-constant expressions in:

- interprocedural constant propagation [GT93]
- interprocedural array section analysis [Tsa94]
- interprocedural performance estimation
- symbolic dependence testing within procedures
- optimization of run-time preprocessing [DSvH93]

The implementation is stable enough to run on most of the well-known Fortran benchmark applications listed in Appendix A. We hope it will serve as a foundation both for further enhancement of symbolic analysis and for its application in program optimization.

Effectiveness

Our main criterion for the value of symbolic analysis is its effect on dependence analysis and parallelization. ParaScope currently lacks an automatic parallel code generator, so we judge our success by comparing ParaScope dependence graphs built with symbolic information against those built without. A dependence tester using our symbolic analysis infrastructure should prove more effective in disproving dependences than the demonstrably successful method in PFC [GKT91]. However, the current implementation of dependence analysis in ParaScope is not yet mature enough for a fair comparison of the underlying symbolic analysis systems.

Efficiency

A significant contribution of this work is a useful symbolic analysis method that is efficient enough to use in a compiler. We would consider this research unsuccessful had the memory and time usage of the implementation grown disproportionately with the size of the programs analyzed.

While such overall efficiency measurements are important, they are no longer sufficient. The exceptional performance needs of an interactive programming environment compel the provision of incremental analysis techniques.

True incrementality involves the maintenance of special data structures that need only be partially modified after an edit — as opposed to the non-incremental technique of throwing everything away and starting over. While we have chosen analysis methods that we believe can be adapted to incremental updates, we are much closer to achieving weaker forms of incrementality:

Demand-driven analysis is not strictly an incremental feature, because it does not save old data structures for use after an edit. However, by building symbolic information only on demand, we reduce the need for *immediate* reanalysis after an edit. If the information demanded between edits is substantially less than the complete solution, we can save significant time.

Recompilation analysis provides a degree of incrementality at the procedure level. Interprocedural analysis is driven by initial analysis files derived from each procedure, and produces solution files for use in optimizing each procedure. When one source procedure of a compiled program is edited, we need only rebuild that procedure's initial information. The interprocedural analysis is repeated *in toto*, but recompilation of an individual procedure is only required if the previous compilation relied on interprocedural solutions that have changed [Tor85, Hal90].

None of these incremental techniques are included in our implementation, but they will be important considerations in a production system.

1.3.2 Organization

This dissertation describes the design and implementation of two symbolic analysis systems. Chapter 2 describes an implementation of interprocedural array section analysis. Experience with that system convinced us that an improved symbolic analysis framework was needed.

Chapter 3 presents the program representations which form the foundation for our symbolic analysis. Chapter 4 describes how we recognize and compare symbolic expressions using a combination of global value numbering and arithmetic simplification. Chapter 5 shows how symbolic predicate information can be combined with our symbolic expressions.

Chapter 6 presents an interprocedural symbolic propagation framework that addresses the recompilation problem. Chapter 7 discusses future research directions and considers the contributions and implications of this work. Appendix A describes the benchmarks used in the experiments of Chapters 3, 4 and 6.

Chapter 2

Array Section Analysis

2.1 Introduction

A major goal of compilation research is to generate efficient code from high-level language constructs. However, most current compilers, even for supercomputers, lack the ability to analyze multiple procedures simultaneously. In the absence of better information, compilers must assume that any procedure call reads and writes every accessible variable, hindering optimization. They discourage programmers from using modular programming style, in which large blocks of code are split into multiple procedures.

Were interprocedural analysis and optimization better supported, programmers could write in a more readable style and compilers still manipulate large regions of code. Wide-ranging compiler optimization would be particularly helpful with parallel computers, many of which still require a large granularity of work for efficiency.

Array side effects require special attention in interprocedural analysis. Consider the following loop:

```
         DO 100 I = 1, N
            CALL SOURCE(A,I,M)
            B(I) = A(INDEX(I),I)
     100 CONTINUE
```

If `SOURCE` only modifies locations in the `I`th column of `A`, then parallel execution of the loop is deterministic. Classical, *scalar* interprocedural analysis only discovers which variables are used and which are defined as side effects of procedure calls. We must determine the *subarrays* that are accessed in order to safely exploit the parallelism.

Callahan and Kennedy proposed a method called *regular section analysis* for tracking interprocedural subarray side-effects. Regular sections describe side effects to common substructures of arrays such as elements, rows, columns and diagonals [CK88a, Cal87]. This chapter describes our adaptation of regular sections and the design and implementation of array side effect analysis in the Rice Parallel Fortran

Converter (PFC) [AK84], an automatic parallelization system that can also export dependence information to the ParaScope programming environment [KMT91b]. The overriding concern in the implementation is that it be effective and efficient enough to be incorporated in a practical compilation system.

Algorithm 2.1 summarizes the steps of the analysis, which is integrated with the three-phase interprocedural analysis and optimization structure of PFC [ACK86, CCKT86]. Regular section analysis adds less than 8000 lines to PFC, a roughly 150,000-line PL/I program which runs under IBM VM/CMS.

This project successfully demonstrated that interprocedural subarray analysis is useful and inexpensive. However, the implementation has several limitations which could be improved by better support for the representation and comparison of symbolic subscripts. Most of these results were previously reported in [HK91].

The remainder of this chapter is organized as follows. Section 2.2 describes our framework for representing bounded subarrays. Sections 2.3 and 2.4 describe the construction of initial sections and their propagation, respectively. Section 2.5 examines the performance of regular section analysis on various Fortran applications. Section 2.6 discusses related work on computing array side effects. Section 2.7 discusses future work, and we conclude in Section 2.8.

2.2 Bounded Sections

A simple way to improve dependence analysis around a call site is to perform inline expansion, replacing the called procedure with its body [AC72]. This precisely represents the effects of the procedure as a sequence of ordinary statements, which are readily understood by existing dependence analyzers. However, even if the whole program becomes no larger, the loop nest which contained the call may grow dramatically, causing explosive growth in resource requirements due to the non-linearity of array dependence analysis and other single-procedure compilation algorithms [CHT91].

To gain some of the benefits of inline expansion without its drawbacks, we must find another representation for the effects of the called procedure. For dependence analysis, we are interested in the memory locations modified or used by a procedure. Given a call to procedure `p` at statement S_1 and an array parameter or global variable A, we wish to compute:

- the set $M_{S_1}^A$ of locations in A that may be modified via `p` called at S_1 and
- the set $U_{S_1}^A$ of locations in A that may be used via `p` called at S_1.

```
Initial_Analysis:
    for each procedure
        for each array (formal parameter, global, or static)
            save section describing shape
            for each reference
                build ranges for subscripts
                merge resulting section with summary MOD or USE section
            save summary sections
        for each call site
            for each array actual parameter
                save section describing passed location(s)
            for each scalar (actual parameter or global)
                save range for passed value

Interprocedural_Propagation:
    solve other interprocedural problems
        build call graph
        compute scalar MOD and USE
        propagate constants
    mark section subscripts and scalars invalidated by modifications as ⊥
    iterating over the call sites
        translate summary sections into call context
        merge translated sections into caller's summary

Dependence_Analysis:
    for each procedure
        for each call site
            for each summary section
                simulate a DO loop running through the elements of the section
    test for dependences (Banerjee's, GCD)
```

Algorithm 2.1 Overview of Regular Section Analysis

We need comparable sets for simple statements as well. We can then test for dependence by intersecting sets. For example, we assume a true dependence from a statement S_1 to a following statement S_2, based on an array A, unless we can prove that

$$M^A_{S_1} \cap U^A_{S_2} == \emptyset.$$

Bounded regular sections comprise the same set of rectangular subarrays that can be written using triplet notation in the proposed Fortran 90 standard [X3J89]. They can represent sparse regions such as stripes and grids and dense regions such as columns, rows, and blocks.

2.2.1 Vectors of Ranges

The descriptors for bounded regular sections are vectors of elements from the subscript lattice in Figure 2.1. Lattice elements include:

- invariant expressions, containing only constants and symbols representing the values of parameters and global variables on entry to the procedure;
- ranges, giving invariant expressions for the lower bound, upper bound, and stride of a variant subscript; and
- $\perp$, indicating no knowledge of the subscript value.

While ranges may be constructed from invariants through a sequence of meet operations, they may also be built directly from the bounds of a loop induction variable.

Since no constraints between subscripts are maintained, merging two regular sections for an array of rank d requires only d independent invocations of the subscript meet operation. We test for intersection of two sections with a single invocation of standard d-dimensional dependence tests. Translation of a formal parameter section to one for an actual parameter is also an $O(d)$ operation (where d is the larger of the two ranks).

2.2.2 Operations on Ranges

Ranges are typically built to represent the values of loop induction variables, such as `I` in the following loop.

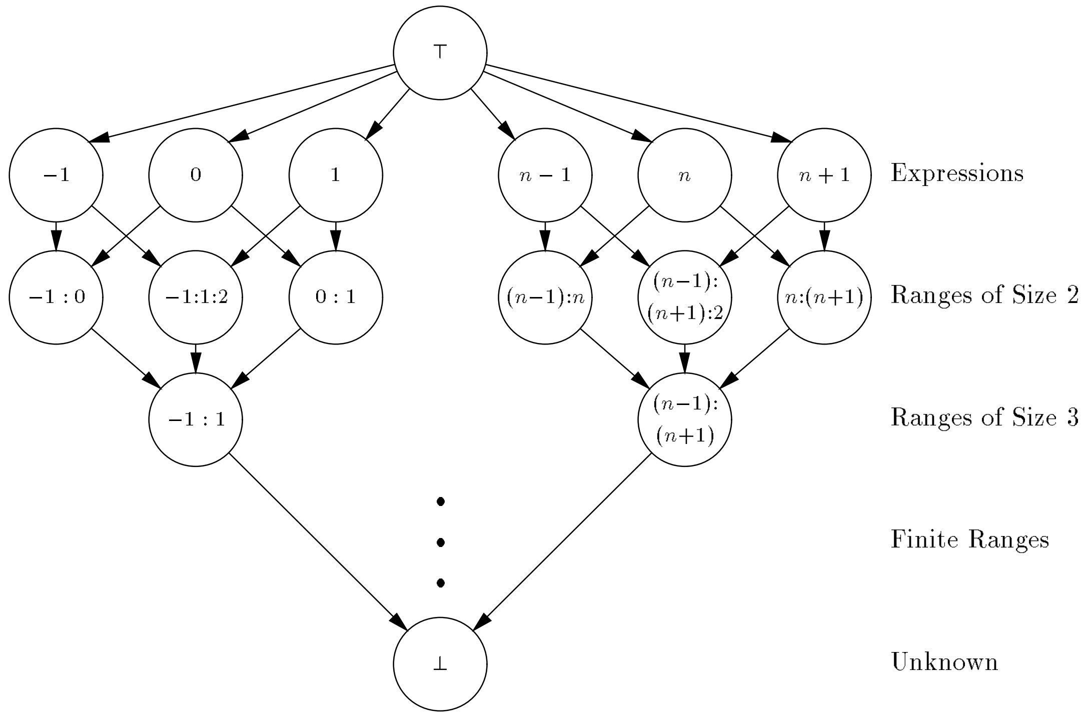


Figure 2.1 Lattice for Regular Section Subscripts

```
DO I = l, u, s
    A(I) = B(2*I+1)
ENDDO
```

We represent the value of `I` as $[l : u : s]$. While l and u are often referred to as the lower and upper bound, respectively, their roles are reversed if s is negative. We can produce a standard lower-to-upper bound form if we know $l \leq u$ or $s \geq 1$; this operation is described in detail in Algorithm 2.2. Standardization may cause loss of information; therefore, we postpone standardization until it is required by some operation, such as merging two sections.

Expressions in ranges are converted to ranges; for example, `2*I+1` in the above loop is represented as $[(2 * l + 1) : (2 * u + 1) : (2 * s)]$. Only invariant expressions are accurately added to or multiplied with a range; Algorithm 2.3 constructs approximations for sums of ranges.

Ranges are merged by finding the lowest lower bound and the highest upper bound, then correcting the stride. An expression (without embedded ranges) is merged with a range or another expression by treating it as a range with a lower bound equal to

```
function standardize([l : u : s])
begin
    diff := u − l
    perfect := false
    if diff and s are both constant then
        if sign(diff) ≠ sign(s) then return(⊤) /* empty range */
        direction := sign(diff)
        u := u - direction * (abs(diff) mod abs(s))
        perfect := true

    else if diff is constant then direction := sign(diff)
    else if s is constant    then direction := sign(s)
    else return(⊥)

    select direction
        when == 0 return(l)
        when > 0 return([l : u : s])
        when < 0 if perfect then return([u : l : (−s)])
                             else  return([u : l : 1])
    end select
end.
```

Algorithm 2.2 Standardizing a Range to Lower-Bound-First Form

```
function build_range(e)
begin
    if e is a leaf expression (constant, formal, or global value; or ⊥) then
        return(e)
    for each subexpression s of e
        replace s with build_range(s)
    select form of e
        when [l1 : u1 : s1] + [l2 : u2 : s2]       [l1' : u1' : s1'] := standardize([l1 : u1 : s1])
                                                   [l2' : u2' : s2'] := standardize([l2 : u2 : s2])
                                                   return([(l1' + l2') : (u1' + u2') : gcd(s1', s2')])
        when a + [l : u : s] or [l : u : s] + a    return([(a + l) : (a + u) : s])
        when a * [l : u : s] or [l : u : s] * a    return([(a * l) : (a * u) : (a * s)])
        otherwise return(⊥)
    end select
end.
```

Algorithm 2.3 Moving Ranges to the Top Level of an Expression

```
function merge(a, b)
begin
    if a == ⊥ or b == ⊥ then return(⊥)
    if a == ⊤ or a == b then return(b)
    if b == ⊤           then return(a)

    if a is a range then let [l_a, u_a, s_a] := standardize(a)
                    else let [l_a, u_a, s_a] := [a, a, ⊤]
    if b is a range then let [l_b, u_b, s_b] := standardize(b)
                    else let [l_b, u_b, s_b] := [b, b, ⊤]

    l' := min(l_a, l_b)                  /* min, max, gcd, and abs can return ⊥ */
    u' := max(u_a, u_b)
    s' := gcd(s_a, s_b, abs(l_a - l_b))  /* gcd(⊤, a) returns a */

    if l' == ⊥        then return(⊥)
    else if s' == ⊥ then return([l' : u' : 1])
    else                   return([l' : u' : s'])
end.
```

Algorithm 2.4 Merging Expressions and Ranges

its upper bound. Algorithm 2.4 thus computes the same result for $1 \wedge 3 \wedge 5$ as for $[1 : 5 : 4] \wedge 3$; namely, $[1 : 5 : 2]$.

Array references whose subscripts refer to scalar formal parameters and global variables can be converted to parameterized sections. We build symbolic expressions for such subscripts, attempting to put the constants at the top, and assign them global value numbers so that the symbolic parts can be quickly tested for equality.

2.3 Initial Analysis

For each procedure, we construct symbolic subscript expressions and accumulate initial regular sections with no knowledge of interprocedural effects. The precision of our analysis depends on recording questions about side effects, but not answering them until the results of other interprocedural analyses are available.

2.3.1 Symbolic Analysis

Constructing regular sections requires the calculation of symbolic expressions for array subscripts. While there are many published algorithms for performing symbolic analysis and global value numbering [Kar76, RT81, RWZ88], their preliminary transformations and complexity make them difficult to integrate into PFC. Our implementation builds on PFC's existing dataflow analysis machinery to represent symbolic expressions by global value numbers.

Leaf value numbers denote constants and the global and parameter values available on procedure entry. We build value numbers for expressions by recursively obtaining the value numbers for subexpressions and reaching definitions. Value numbers reaching the same reference along different def-use edges are merged. If either the merging or the occurrence of an unknown operator creates a unknown ($\perp$) value, the whole expression is lowered to $\perp$.[1]

Induction variables are recognized by their defining loop headers and replaced with the inductive range. (For historical reasons, auxiliary induction variables are not identified in this initial analysis.) For example, consider the following code fragment.

```
SUBROUTINE S1(A,N,M)
    DIMENSION A(N)
    DO I = 1, N
        A(M*I) = 0.0
    ENDDO
    RETURN
END
```

Dataflow analysis constructs def-use edges from the subroutine entry to the uses of N and M, and from the DO statement to the use of I. It is therefore simple to compute the subscript in A's regular section, $\mathtt{M} * [1 : \mathtt{N} : 1]$, which is converted to the range $[\mathtt{M} : \mathtt{M} * \mathtt{N} : \mathtt{M}]$ (the names M and N are actually replaced by value numbers referring to their formal parameter indices). Note that expressions that are nonlinear during initial analysis may become linear in later phases, especially after constant propagation.

---

[1] Chapter 4 gives a more subtle treatment of this and other issues in symbolic expression analysis.

2.3.2 Avoiding Compilation Dependences

To construct accurate value numbers, we require knowledge about the effects of call sites on scalar variables. However, using interprocedural analysis to determine these effects can be costly.

A programming support system using interprocedural analysis must examine each procedure at least twice: once when gathering initial information to be propagated between procedures, and again when using the results of this propagation in dependence analysis and/or transformations.[2] By pre-computing the initial information, we can construct an interprocedural propagation phase which iterates over the call graph without additional direct examination of any procedure.

To achieve this minimal number of passes, all interprocedural analyses must gather initial information in one pass, without the benefit of each others' interprocedural solutions. However, to build precise initial regular sections, we need information about the side effects of calls on scalars used in subscripts. In the following code fragment, we must assume that `M` is modified to an unknown value unless proven otherwise:

```
SUBROUTINE S1(A,N,M)
    DIMENSION A(N)
    CALL CLOBBER(M)
    A(M) = 0.0
    RETURN
END
```

To achieve precision without adding a separate initial analysis phase for regular sections, we build regular section subscripts as if side effects did not occur, while annotating each subscript expression with its hazards, or side effects that would invalidate it. We thus record that `A(M)` is modified, with the sole parameter of `CLOBBER` as a hazard on `M`. During the interprocedural phase, after producing the classical scalar side effect solution, but before propagating regular sections, we check to see if `CLOBBER` may change `M`. If so, we change `S1`'s array side effect to `A(⊥)`. A similar technique has proven successful for interprocedural constant propagation in PFC [Tor85, CCKT86].

[2] Another pass is needed to examine call sites for building the call graph, but this can be combined with the initial information gathering. For some problems, the initial pass and the usage pass can be combined, but then a separate pass is still needed to build the call graph

Hazards must be recorded with each scalar expression saved for use in regular section analysis: scalar actual parameters and globals at call sites as well as array subscripts. When we merge two expressions or ranges, we take the union of their hazard sets.

2.3.3 Building Summary Sections

With the above machinery in place, the USE and MOD regular sections for the local effects of a procedure are constructed easily. In one pass through the procedure, we examine each reference to a formal parameter, global, or static array. The symbolic analyzer provides value numbers for the subscripts on demand; the resulting vector is a regular section. After the section for an individual reference is constructed, it is immediately merged with the appropriate cumulative section(s), then discarded.

2.4 Interprocedural Propagation

Regular sections for formal parameters are translated into sections for actual parameters as we traverse edges in the call graph. The translated sections are merged with the summary regular sections of the caller, requiring another translation and propagation step if this changes the summary. To extend our implementation to recursive programs and have it terminate, we must bound the number of times a change occurs.

2.4.1 Translation into a Call Context

If we were analyzing Pascal arrays, mapping the referenced section of a formal parameter array to one for the corresponding actual parameter would be simple. We would only need to replace formal parameters in subscript values of the formal section with their corresponding actual parameter values, then copy the resulting subscript values into the new section. However, Fortran provides no guarantee that formal parameter arrays will have the same shape as their actual parameters, nor even that arrays in common blocks will be declared to have the same shape in every procedure. Therefore, to describe the effects of a called procedure for the caller, we must translate the referenced sections according to the way the arrays are reshaped.

One straightforward translation method is to linearize the subscripts for the referenced section of a formal parameter, adding the offset of the passed location of the actual parameter [BC86]. The resulting section would give referenced locations of the

```
function translate(bounds_F, ref_F, bounds_A, pass_A)
begin
    if ref_F == ⊤ then return(⊤)
    consistent := true
    for i := 1 to rank(A)
        if not consistent then ref_A[i] := ⊥
        else if i > rank(F) then ref_A[i] := pass_A[i]
        else
            replace scalar formal parameters in bounds_F and ref_F
                    with their corresponding actual parameters
            bounds_i:= bounds_F[i] - lo(bounds_F[i]) + pass_A[i]
            ref_i:= ref_F[i] - lo(bounds_F[i]) + pass_A[i]
            consistent := (bounds_i == bounds_A[i])
            if consistent then ref_A[i] := ref_i
            else if stride(ref_i) == hi(bounds_A[i + 1]) - lo(bounds_A[i + 1]) then
                    /* delinearization is possible */
                if (i == rank(F)) and ((ref_i fits in bounds_A[i]) or assume_fit) then
                    ref_A[i] := ref_i
                else ref_A[i] := ⊥ /* conservative answer */
    end for
    return(ref_A)
end.
```

Algorithm 2.5 Translating a Summary Section actual as if it were a one-dimensional array. However, if some subscripts of the original section are ranges or non-linear expressions, linearization contaminates the other subscripts, greatly reducing the precision of dependence analysis. For this reason, we forego the effort of linearization and translate significantly reshaped dimensions as $\perp$.

Algorithm 2.5 shows one method for translating a summary section for a formal parameter `F` into a section for its corresponding actual parameter `A`. Translation proceeds from left to right through the dimensions, and is precise until a dimension is encountered where the formal and actual parameter are *inconsistent* (having different sizes or non-zero offset). The first inconsistent dimension is also translated precisely if it is the last dimension of `F` and the referenced section subscript value(s) fit in the bounds for `A`. Delinearization, which is not implemented, may be used to recognize that a reference to `F` with a column stride the same as the column size of `A` corresponds to a row reference in `A`.

2.4.2 Treatment of Recursion

The current implementation handles only non-recursive Fortran. Therefore, it is sufficient to proceed in reverse invocation order on the call graph, translating sections up from leaf procedures to their callers. The final summary regular sections are built in order, so that incomplete regular sections need never be translated into a call site. However, the proposed Fortran 90 standard allows recursion [X3J89], so it must be handled someday. Unfortunately, a straightforward iterative propagation of regular sections will not terminate, since the lattice has unbounded depth.

Li and Yew [LY88b] and Cooper and Kennedy [CK88b] describe approaches for propagating subarrays that are efficient regardless of the depth of the lattice. However, it may be more convenient to implement a simple iterative technique while simulating a bounded-depth lattice. If we maintain a counter with the summary regular section for each array and procedure, then we can limit the number of times we allow the section to become larger (lower in the lattice) before going to $\perp$. We suggest keeping one small counter (e.g., two bits) per subscript. Variant subscripts will then go quickly to $\perp$, leaving precise subscripts unaffected. If we limit each subscript to being lowered in the subscript lattice k times, then an array of rank d will have an effective lattice depth of $kd + 1$.

Since each summary regular section is lowered at most $O(kd)$ times, each associated call site is affected at most $O(kdv)$ times (each time involving an $O(d)$ merge), where v is the number of *referenced* global and parameter variables. In the worst case, we then require $O(kd^2ve)$ subscript merge and translation operations, where e is the number of edges in the call graph. This technique allows us to use a lattice with bounds information while keeping time complexity comparable to that obtained with the restricted regular section lattice.

2.5 Experimental Results

The precision, efficiency, and utility of regular section analysis must be demonstrated by experiments on realistic programs. For these experiments, our candidates for realistic programs are the LINPACK library of linear algebra subroutines [DBMS79] and the Rice Compiler Evaluation Program Suite. We ran the programs through regular section analysis and dependence analysis in PFC, then examined the resulting dependence graphs by hand and with the ParaScope editor [BKK+89].

2.5.1 Benchmarks

LINPACK

Analysis of LINPACK provides a basis for comparison with other methods for analyzing interprocedural array side effects. Both Li and Yew [LY88a] and Triolet [Tri85] found several parallel calls in LINPACK using their implementations in the University of Illinois translator, Parafrase. LINPACK proves that useful numerical codes can be written in the modular programming style for which parallel calls can be detected.

However, LINPACK is a set of library routines, not a complete program. To test our interprocedural analysis, we constructed a dummy program which calls all the main LINPACK entry points, passing unknown values for each parameter. Realistic programs have sparser use of linear algebra, but consistently pass the same constant value for certain library parameters (such as 1 for an array stride).

RiCEPS

The Rice Compiler Evaluation Program Suite is a collection of complete applications codes from a broad range of scientific disciplines. Our colleagues at Rice have already run several experiments on RiCEPS. Porterfield modeled cache performance using an adapted version of PFC [Por89]. Goff, Kennedy and Tseng studied the performance of dependence tests on RiCEPS and other benchmarks [GKT91]. Some RiCEPS and

RiCEPS candidate codes have also been examined in a study on the utility of inline expansion of procedure calls [CHT91]. The six programs studied here are two RiCEPS codes `linpackd` and `track`) and four codes from the inlining study.[3]

2.5.2 Precision

The precision of regular sections, or their correspondence to the true access sets, is largely a function of the programming style being analyzed. LINPACK is written in a style which uses many calls to the BLAS (basic linear algebra subroutines), whose true access sets are precisely regular sections. We did not determine the true access sets for the subroutines in RiCEPS, but of the six programs analyzed, only `dogleg` and `linpackd`, which actually call LINPACK, exhibited the LINPACK coding style.

While there exist regular sections to precisely describe the effects of the BLAS, the initial analysis phase was unable to construct them under complicated control flow. With changes to the BLAS to eliminate unrolled loops and the conditional computation of values used in subscript expressions, our implementation was able to build minimal regular sections that precisely represented the true access sets. The modified `DSCAL`, for example, looks as follows:

```
SUBROUTINE DSCAL(N, DA, DX, INCX)
    DOUBLE PRECISION DA, DX(*)
    IF (N .LE. 0) RETURN
    DO I = 1, N*INCX, INCX
        DX(I) = DA * DX(I)
    ENDDO
    RETURN
END
```

Obtaining precise symbolic information is a problem in all methods for describing array side effects. Triolet made similar changes to the BLAS; Li and Yew avoided them by first performing interprocedural constant propagation. The fundamental nature of this problem indicates the desirability of a clearer Fortran programming style or more sophisticated handling of control flow (such as that described in Section 2.7).

[3] Note that this set of programs is disjoint from the RiCEPS programs described in Appendix A. We omitted `linpackd` from the experiments of later chapters because it is not a genuine application program, and we used another version of `track` included in the Perfect Club benchmarks [CKPK90].

program name	Lines	Procs	IP only	IP +RS	% Change
`efie`	1254	18	209	232	+10
`euler`	1113	13	117	138	+15
`vortex`	540	19	65	87	+25
`track`	1711	34	191	225	+15
`dogleg`	4199	48	272	377	+28
`linpackd`	355	10	28	44	+36
total	9172	142	882	1103	+25

Table 2.1 Analysis times in seconds (PFC on an IBM 3081D)

2.5.3 Efficiency

We measured the total time taken by PFC to analyze the six RiCEPS programs. Parsing, initial analysis, interprocedural propagation, and dependence analysis were all included in the execution times. Table 2.1 compares the analysis time required using classical interprocedural summary analysis alone ("IP only") with that using summary analysis and regular section analysis combined ("IP + RS").[4]

The most time-consuming part of our added code is the symbolic analysis for subscript values of the initial sections, which includes an invocation of dataflow analysis. More symbolic analysis would improve the practicality of the entire method. Overall, the additional analysis time is comparable to that required to analyze programs after heuristically-determined inline expansion in Cooper, Hall and Torczon's study [CHT91].

Absolute timings for the array side effect analyses implemented in Parafrase by Triolet and by Li and Yew, have not been published, though Li and Yew do state that their method runs 2.6 times faster than Triolet's [LY88a]. Both experiments were run only on LINPACK; it would be particularly interesting to know how their methods perform on complete applications.

2.5.4 Utility

We chose three measures of utility:

[4]We do not present times for dependence analysis with no interprocedural information because that analysis is cut short in the presence of call sites. Little can be done to parallelize a loop containing calls with unknown side effects.

	All Dependences			Array Dep. on Calls in Loops					
				loop carried			loop independent		
source	IP	RS	% ⇓	IP	RS	% ⇓	IP	RS	% ⇓
`efie`	12338	12338		177	177		81	81	
`euler`	1818	1818		70	70		30	30	
`vortex`	1966	1966		220	220		73	73	
`track`	4737	4725	0.25	68	67	1.5	27	26	3.7
`dogleg`	1858	1675	9.8	226	168	25.7	80	59	26.2
`linpackd`	496	399	19.6	191	116	39.3	67	45	32.8
RiCEPS	23213	22921	1.25	952	818	14.1	358	314	12.3
LINPACK	12336	11035	10.5	3071	2064	32.8	1348	1054	21.8

Table 2.2 Effects of Regular Section Analysis on Dependences

- reduced numbers of dependences and dependent references,
- increased numbers of calls in parallel loops, and
- reduced parallel execution time.

Reduced Dependence

Table 2.2 compares the dependence graphs produced using classical interprocedural summary analysis alone ("IP") and summary analysis plus regular section analysis ("RS").[5]

LINPACK was analyzed without interprocedural constant propagation, since library routines may be called with varying array sizes. The first set of three columns gives the sizes of the dependence graphs produced by PFC, counting all true, anti and output dependence edges on scalar and array references in DO loops (including those references not in call sites). The other sets of columns count only those dependences incident on array references in call sites in loops, with separate counts for loop-carried and loop-independent dependences. Preliminary results for eight of

[5]The dependence graphs resulting from no interprocedural analysis at all are not comparable, since no calls can be parallelized and their dependences are collapsed to conserve space.

the 13 Perfect benchmarks indicate a reduction of 0.6 percent in the total size of the dependence graphs.[6]

Parallelized Calls

Table 2.3 examines the number of calls in LINPACK which were parallelized after summary interprocedural analysis alone ("IP"), after Li and Yew's analysis [LY88a], and after regular section analysis ("RS"). (Triolet's results from Parafrase resembled Li and Yew's.)

Most (17) of these call sites were parallelized in ParaScope, based on PFC's dependence graph, with no transformations being necessary. The eight parallel call sites detected with summary interprocedural analysis alone were apparent in ParaScope, but exploiting the parallelism requires a variant of statement splitting that is not yet supported. Starred entries ($\star$) indicate parallel calls which were precisely summarized by regular section analysis, but which were not detected as parallel due to a deficiency in PFC's symbolic dependence test for triangular loops. One call in `QRDC` was mistakenly parallelized by Parafrase [Li90].

These results indicate, at least for LINPACK, that there is no benefit to the generality of Triolet's and Li and Yew's methods. Regular section analysis obtains exactly the same precision, with a different number of loops parallelized only because of differences in dependence analysis and transformations.

Improved Execution Time

Two calls in the RiCEPS programs were parallelized: one in `dogleg` and one in `linpackd`. Both were the major computational loops (`linpackd`'s in `DGEFA`, `dogleg`'s in `DQRDC`).[7] Running `linpackd` on 19 processors with the one call parallelized was enough to speed its execution by a factor of five over sequential execution on the Sequent Symmetry at Rice.

---

[6]This implementation does not propagate information for arrays in common blocks. This deficiency certainly resulted in more dependences for the larger programs in Perfect.

[7]In the inlining study at Rice, none of the commercial compilers was able to detect the parallel call in `dogleg` even after inlining, presumably due to complicated control flow [CHT91].

routine name	calls in DO loops	Parallel Calls IP	Li-Yew	RS
·GBCO	8	1	1	1
·GECO	8	1	1	1
·PBCO	8	1	1	1
·POCO	8	1	1	1
·PPCO	8	1	1	1
·SICO	1	1	1	1
·SPCO	1	1	1	1
·TRCO	4	1	1	1
·GBFA	3		1	1
·GEDI	4		1	1
·GEFA	3		1	1
·PODI	4		2	2
·QRDC	9		5	4
·SIDI	6		3	*3
·SIFA	3		3	*3
·SVDC	15		3	7
·TRDI	4		—	1
other	47			
total(36)	144	8	27	31

Table 2.3 Parallelization of LINPACK

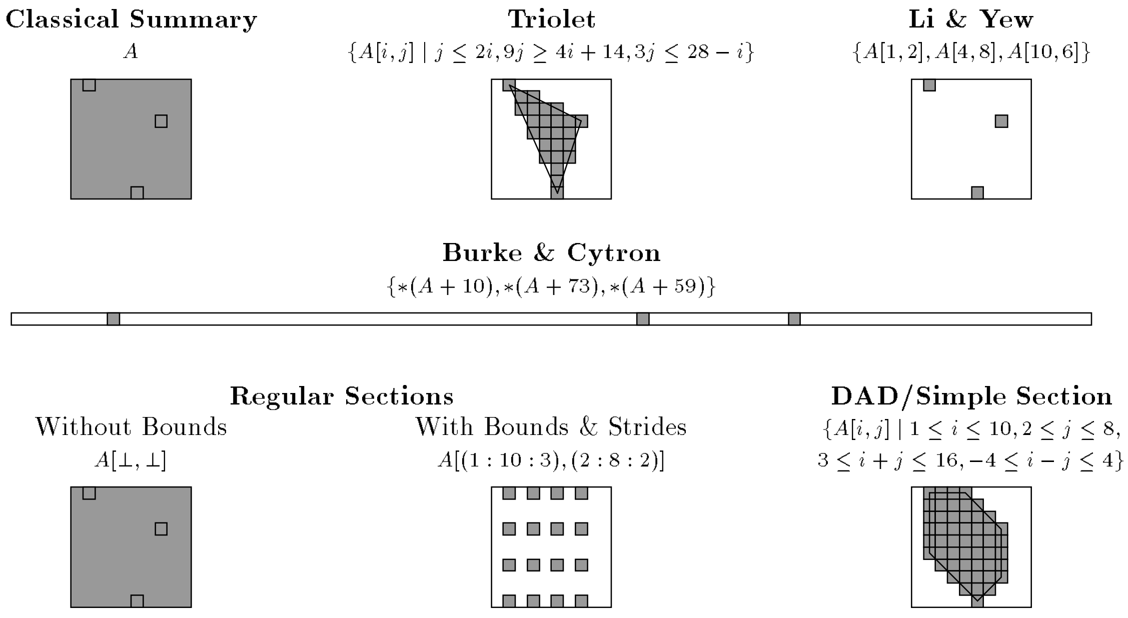


Figure 2.2 Summarizing the References $A[1,2]$, $A[4,8]$, and $A[10,6]$

2.6 Related Work

Several representations have been proposed for representing interprocedural array access sets. The contrived example in Figure 2.2 shows the different patterns that they can represent precisely. Evaluating these methods involves examining the complexity and precision of:

- representing the sets $M_{S_1}^A$ and $U_{S_2}^A$ of accessed elements,
- merging descriptors to summarize multiple accesses (we call this the *meet* operation, because most descriptors may be viewed as forming a lattice),
- testing two descriptors for *intersection* (dependence testing), and
- translating descriptors at call sites (especially when there are array reshapes).

Handling of recursion turns out not to be a major issue. Iterative techniques can guarantee convergence to a fixed point solution using Cousot's technique of *widening operators* [CC77, Cou81]. Li and Yew proposed a preparatory analysis of recursive programs that guarantees termination in three iterations [LY88b, LY88c]. Either of these methods may be adapted for regular sections.

2.6.1 Summary Methods

Summary methods describe each kind of side effect using a single subarray descriptor for each array. They include the many variations of regular sections developed at Rice, described separately below. Earlier summary methods include the classical all-or-none summary and the convex regions of Triolet et al.

Summarizing multiple array references with one descriptor is potentially less accurate than keeping separate descriptors. However, it is cheaper to translate this a single descriptor during interprocedural propagation than a list of them, and cheaper to test two descriptors for intersection than do a quadratic-time pairwise intersection between two lists.

Classical Methods

The classical methods of interprocedural summary dataflow analysis compute MOD and USE sets indicating which parameters and global variables may be modified or used in the procedure [Ban78, Bar77, CK85]. Such summary information costs only two bits per variable. Meet and intersection may be implemented using single-bit or bit-vector logical operations. Also, there exist algorithms that compute complete solutions, in which the number of meets is linear in the number of procedures and call sites in the program, even when recursion is permitted [CK88b].

Unfortunately, our experiences with PFC indicate that this summary information is too coarse for dependence testing and the effective detection of parallelism [CK88a]. The problem is that the only access sets representable in this method are "the whole array" and "none of the array" (see Figure 2.2). Such coarse information limits the detection of data decomposition, an important source of parallelism, in which different iterations of a loop work on distinct subsections of a given array.

Convex Regions

Triolet, Irigoin and Feautrier proposed to calculate linear inequalities bounding the set of array locations affected by a procedure call [TIF86, Tri85]. This representation and its intersection operation are precise for convex regions of access. Other patterns, such as array accesses with non-unit stride and non-convex results of meet operations, are given convex approximations.

Operations on these regions are expensive in the worst case; the meet operation requires finding the convex hull of the combined set of inequalities and intersection uses a potentially exponential linear inequality solver [Ban86]. A succession of meet operations can also produce complicated regions with potentially as many inequalities as the number of primitive accesses merged together. Translation at calls sites is precise only when the formal parameter array in the called procedure maps to a (sub)array of the same shape in the caller. Otherwise, the whole actual parameter array is assumed accessed by the call.

The region method can maintain arbitrary convex array accesses precisely, but if the important array accesses prove to be regular sections, the additional complexity is wasted. Recently, researchers on the PIPS project have extended the region method to handle common cases more efficiently [Iri93].

2.6.2 Reference Lists

Some proposed methods do not summarize, but represent each reference separately. Descriptors are then lists of references, the meet operation is list concatenation (possibly with a check for duplicates), and translation and intersection are just the repeated application of the corresponding operations on simple references. However, this has two significant disadvantages:

- translation of a descriptor requires time proportional to the number of references, and
- intersection of descriptors requires time quadratic in the number of references.

Reference list methods are simple and precise, but are asymptotically as expensive as in-line expansion.

Linearization

Burke and Cytron proposed representing each multidimensional array reference by linearizing its subscript expressions to a one-dimensional address expression. Their method also retains bounds information for loop induction variables occurring in the expressions [BC86]. They describe two ways of implementing the meet operation. One merely keeps a list of the individual address expressions. The constructs a composite expression that can be polynomial in the loop induction variables. The disadvantages of the first method are described above. The second method appears complicated and has yet to be rigorously described. Linearization in its pure form is ill-suited to summarization, but might be a useful fallback for a true summary technique because of its ability to handle arbitrary reshapes.

Atom Images

Li and Yew extended Parafrase to compute sets of *atom images* describing the side effects of procedures [LY88a, LY88b]. Like the original version of regular sections described in Callahan's thesis [Cal87], these record subscript expressions that are linear in loop induction variables along with bounds on the induction variables. Any reference with linear subscript expressions in a triangular iteration space can be precisely represented, and they keep a separate atom image for each reference.

The expense of translating and intersecting lists of atom images is too high a price to pay for their precision. Converting atom images to a summary method would produce something similar to the regular sections described below.

2.6.3 Summary Sections

The precise methods described above are expensive because they allow arbitrarily large representations of a procedure's access sets. The extra information may not be useful in practice; simple array access patterns are probably more common than others. To avoid expensive intersection and translation operations, descriptor size should be independent of the number of references summarized. Operations on descriptors should be linear or, at worst, quadratic in the rank of the array. Researchers at Rice have defined several variants of *regular sections* to represent common access patterns while satisfying these constraints [Cal87, CK88a, Bal89, BK89].

Original Regular Sections

Callahan's thesis proposed two regular section frameworks. He dismissed the first, resembling Li and Yew's atom images, due to the difficulty of devising efficient standardization and meet operations [Cal87].

Restricted Regular Sections

The second framework, *restricted* regular sections [Cal87, CK88a], is limited to access patterns in which each subscript is

- a procedure-invariant expression (with constants and procedure inputs),
- unknown (and assumed to vary over the entire range of the dimension), or
- unknown but diagonal with one or more other subscripts.

The restricted sections have efficient descriptors: their size is linear in the number of subscripts, their meet operation quadratic (because of the diagonals), and their intersection operation linear. However, they lose too much precision by omitting bounds information. The restricted section lattice has the finite descending chain property, which we originally thought necessary for efficient handling of recursive programs. However, we can obtain fixed-point solutions on more complicated lattices by applying the techniques of Cousot or of Li and Yew [CC77, LY88c]

Bounded Regular Sections

Anticipating that restricted regular sections would not be precise enough for effective parallelization, Callahan and Kennedy proposed an implementation of regular sections with bounds. That proposal led to our development of the bounded regular section framework in this chapter. Our sections include bounds and stride information, but omit diagonal constraints. The resulting analysis is therefore less precise in the representation of convex regions than Triolet regions or the Data Access Descriptors described below. However, this is the first interprocedural summary implementation with stride information, which provides increased precision for non-convex regions.

As noted above, the size of bounded regular section descriptors and the time required for the meet operation are both linear in the number of subscripts. Intersection is implemented using standard dependence tests, which also take time proportional to the number of subscripts.

Data Access Descriptors

Concurrently with the original work reported in this chapter, Balasundaram and Kennedy developed Data Access Descriptors (DADs) as a general technique for describing data access [Bal89, BK89, Bal90]. DADs represent information about both the shapes of array accesses and their traversal order; for our comparison we are interested only in the shapes. The *simple section* part of a DAD represents a convex region similar to those of Triolet *et al.*, except that boundaries are constrained to be parallel to one coordinate axis or at a 45° angle to two axes. Stride information is represented in another part of the DAD.

Data Access Descriptors are probably the most precise summary method that can be implemented with reasonable efficiency. They can represent the most likely rectangular, diagonal, triangular, and trapezoidal accesses. In size and in time required for meet and intersection they have complexity quadratic in the number of subscripts (which is reasonable given that most arrays have few subscripts).

The bounded sections implemented here are both less expensive and less precise than DADs. Tsalapatas subsequently extended our methods to a DAD implementation in ParaScope, but no empirical comparison of the relative benefits of regular sections vs. DADs has been completed [Tsa94].

2.7 Lessons

More experiments are required to fully evaluate the performance of regular section analysis on complete applications and find new areas for improvement. Based on the studies conducted so far, extensions to provide better handling of conditionals and flow-sensitive side effects seem promising. Both extensions require better support from the symbolic analysis infrastructure. Both have now been implemented, to differing degrees, by Tsalapatas [Tsa94].

2.7.1 Conditional Symbolic Analysis

Consider the following example, derived from the BLAS:

```
SUBROUTINE D(N, DA, DX, INCX)
    DOUBLE PRECISION DA, DX(*)
    IF (INCX .LT. 0) THEN
        IX = (-N+1)*INCX + 1
    ELSE
        IX = 1
    ENDIF
    DO I = 1, N
       DX(IX) = DA * DX(IX)
       IX = IX + INCX
    ENDDO
    RETURN
END
```

The two computations of the initial value for IX correspond to different ranges for the subscript of DX: $[(1+\mathtt{INCX}*(1-\mathtt{N})):1:\mathtt{INCX}]$ and $[1:(1+\mathtt{INCX}*(\mathtt{N}-1)):\mathtt{INCX}]$. It turns out that these can both be represented by $[1:(1+|\mathtt{INCX}|*(\mathtt{N}-1)):|\mathtt{INCX}|]$. For the merge operation to produce this precise result requires that it have an understanding of the control conditions under which expressions are computed.

One way to address this problem is to represent explicitly the conditional computation of IX; e.g.,

$$\mathtt{IX} == \textbf{if } (\mathtt{INCX} < 0) \textbf{ then } a \textbf{ else } c \textbf{ fi}.$$

This is written as $\gamma((\mathtt{INCX} < 0), a, c)$ in the notation of Chapter 3. By writing DX in this form, we can encode its dependence on the passed value of INCX.

If we delay merging sections until after interprocedural MOD and constant information are available, we will usually find that INCX is 1. In that case, the subscript expressions can be simplified to the point where the section merge is precise. In short, a richer symbolic representation would allow us to save conditional computations for later evaluation when more information is available.

2.7.2 Killed Regular Sections

We have already found programs (`scalgam` and `euler`) in which the ability to recognize and privatize temporary arrays would cut the number of dependences dramatically, allowing some calls to be parallelized. We could recognize interprocedural temporary arrays by determining when an entire array is guaranteed to be modified before being used in a procedure. While this is a flow-sensitive problem, and therefore expensive to solve in all its generality, even a very limited implementation should be able to catch the initialization of many temporaries.

The subscript lattice for killed sections is the same one used for USE and MOD sections; however, since kill analysis must produce *underestimates* of the affected region in order to be conservative, the lattice needs to be inverted. In addition, this approach requires an *intra*procedural dependence analysis framework capable of using array kill information, such as those described by Rosene [Ros90] and by Gross and Steenkiste [GS90].

While the computation of array kills does not necessarily require better symbolic information than that implemented in PFC, KILL analysis is particularly sensitive to improvements. Flow-insensitive summaries of MOD and REF sections describe elements that *may* be accessed. If our subscript analysis is imprecise in one dimension, we can say that it is $\perp$ (assuming the subscript ranges over all values) while still keeping information about other dimensions. However, if we give up on a subscript in a killed section, we say that it is $\top$ and we are not sure that any of the array is definitely modified. Failures of symbolic analysis hurt more when computing KILL information.

2.7.3 Plan of Attack

We chose to focus our attention on symbolic analysis, to improve the efficiency of array section analysis and to support dependence testing.

Intraprocedural dataflow analysis is very expensive in PFC. This is particularly true when call sites are treated as non-killing definitions of all interprocedural variables. We need a better way of handling call sites, and a better dataflow representation than traditional def-use chains. Both needs are addressed in Chapter 3.

Value numbering is still rather primitive. The class of expressions represented could be much larger, and more aggressive efforts should be made to rearrange expressions to a standard form. Chapter 4 gives more general methods for representing and rewriting symbolic expressions.

Symbolic comparisons are made differently in the array section analysis and in dependence testing for historical reasons. The new implementation in ParaScope employs a single representation of symbolic expressions supporting multiple analyses.

Conditional branches are not exploited. Inferring bounds and evaluating branches could both have significant payoff. Chapter 5 shows how to propagate predicates derived from branches.

2.8 Summary

Regular section analysis can be a practical addition to a production compiler. Its initial analysis and interprocedural propagation can be integrated with those for other interprocedural techniques. The required changes to dependence analysis are trivial—the same ones needed to support Fortran 90 sections. Our experiments demonstrate that regular section analysis is an effective means of discovering parallelism, given programs written in an appropriately modular programming style.

Chapter 3

Program Representations

3.1 Introduction

Any program analysis method requires some abstract representation of the program on which to operate. The initial representation is the source code as ASCII characters, for which the abstract syntax tree (AST) is a parsed, hierarchical equivalent. The analysis parts of the ParaScope system derive more convenient representations from the AST and then derive facts about the program. The transformation parts use these facts to rewrite the AST. Ultimately, the back end converts the AST to machine code, which we hope will run faster after all this work than if we had just left it alone.

This chapter deals with basic program representations for an individual procedure which are close to the AST: control-flow graphs; control-dependence graphs; data-flow graphs, including simple def-use chains and static single-assignment (SSA) form; and gated single-assignment (GSA) form, a data-flow representation which also encodes information about conditional control over data-flow. Handling of multiple procedures is described in Chapter 6.

Control-flow graphs are a traditional foundation of compile-time analysis, the framework on which every kind of data-flow analysis proceeds. This remains true for sequential programming models, but we now know how to write and compile programs in a dataflow model — in which all computation involves expressions passing their results to other expressions, and control decisions are just another kind of result. Control flow can be rewritten as data flow. Both control dependence graphs and GSA form descend from this realization.

There is a more subtle dichotomy involving the very nature of control-driven execution. En route to a particular statement S, we typically encounter many conditional branches. Some decide whether or not we reach S. Other branches decide only which other statements (possibly including variable definitions) are encountered on the way. GSA form provides a framework for reasoning about this distinction, which will be exploited in Chapter 4.

3.2 Control over Execution

Both control-flow and control-dependence graphs encode conditional execution of statements in individual procedures. The control-flow graph plus variable declarations encodes all the information needed for execution, but the control-dependence graph omits sequencing of variable references that must be enforced separately.

3.2.1 Control Flow Graphs

Building the control-flow graph is a first step in compiler optimization. Informally, a sequential computer executing a procedure will start at some entry point, step through the statements one at a time, jump around as dictated by structure and GOTOs, follow calls to and returns from other procedures, and finally either return to the caller or abort the whole program because of a serious error.[1] The implied execution sequencing is made explicit as edges in a graph for ease of analysis.

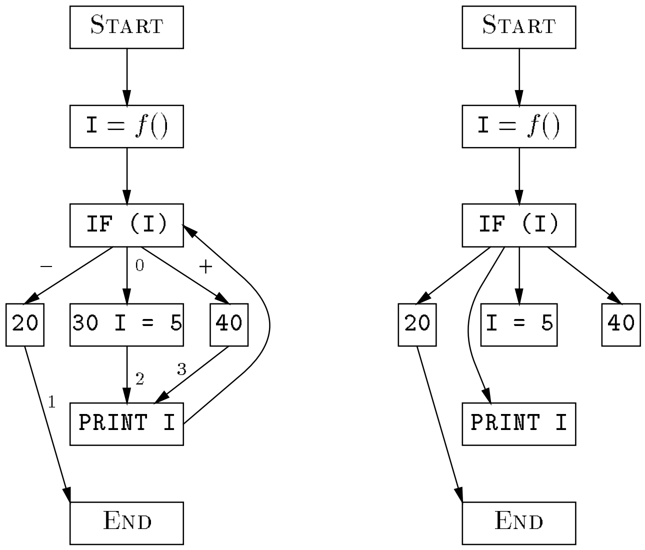
Figure 3.1 shows an example of the control-flow graph G_{CF}. Each node represents a *basic block* of zero or more statements in straight-line sequence. Edges represent the potential flow of control from the end of the source node to the beginning of the sink node. The distinguished node START has no incoming edges, and has outgoing edges to each node with a procedure entry point; the distinguished node END has no outgoing edges, and has incoming edges from every node with a return point.

A node with multiple out-edges is a *branch*, with each out-edge labeled by the corresponding value of the branch condition (such as negative, zero, and positive for the arithmetic `IF`). A node with multiple in-edges, such as `PRINT` and END in the example, is a *merge*, and its in-edges are ordered (as indicated by the notations on the in-edges to the `PRINT`).

Maps enable us to find the source statement(s) corresponding to a G_{CF} node and vice versa. Several other annotations on the control flow graph are easily derived, the most useful are described below.

[1] More sophisticated exception handling than this is difficult for a compiler to analyze; witness the simplification of PL/1 to PL/8 by removing all non-fatal exceptions [AH82].

```
    SUBROUTINE P
    I = f()
10  IF (I) 20, 30, 40
20  RETURN
30  I = 5
    GOTO 50
40  CONTINUE
50  PRINT I
    GOTO 10
    END
```



Control Flow G_{CF} Pre-Dominator Tree

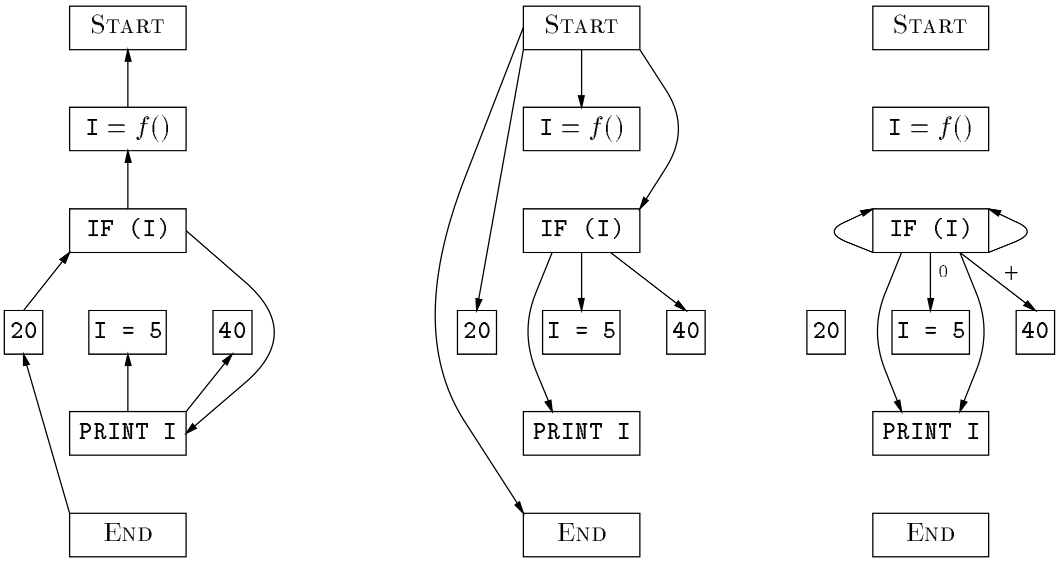
Figure 3.1 Representations of Control

Dominance Trees

Dominance trees give information about which nodes must be encountered on route to or from other nodes. They are necessary to the efficient construction of control dependences and SSA form, and useful in many other ways as well.

A node a *pre-dominates* a node b ($a \preceq b$) if all G_{CF} paths from START to b include a. The pre-dominance relation is reflexive and transitive. Eliminating the reflexive cases in the relation (where $a = b$) we get the irreflexive but still transitive *strict* pre-dominance relation ($a \prec b$). Strict pre-dominance forms a partial ordering on the nodes and can be represented as a forest of trees, with all the reachable nodes in a tree rooted by START.

The post-dominance relation is equivalent to the pre-dominance relation computed on G_{CF} with the direction of edges reversed. A node b post-dominates a ($b \succeq a$) if all G_{CF} paths from a to END include b. The strict post-dominance relation ($b \succ a$, with $a \neq b$) also forms a partial order. In the forest of post-dominance trees, any node not in the tree rooted by END is dead-end code, presumably in an infinite loop.



Post-Dominator Tree Loop Nesting Tree Control Dependence G_{CD}

Figure 3.2 More Representations of Control

Loop-nesting Tree

Most general-purpose procedural languages allow arbitrary control flow through the use of GOTOs. To understand the loop structure of the program — to both discover iterative computations and locate parallelism — we must recognize natural loops in arbitrary control-flow graphs.

Algorithm 3.1 builds a loop-nesting tree for G_{CF}. It is essentially Tarjan's algorithm for recognizing flow-graph reducibility, except that it does not fail for irreducible graphs [Tar74]. Reducible graphs are those in which each nested strongly connected region has a unique entry node, the *loop header*, that pre-dominates all other nodes in the SCR.[2]

We adapt Tarjan's method to recognize each irreducible loop and arbitrarily select one of its multiple entry nodes as the header. The letter labels to the left in the algorithm (*a:*, etc.) correspond to the same labels in Tarjan's algorithm. FIND and UNION refer to the operations of Tarjan's almost-linear disjoint-set union-find algorithm; UNION(x, y, z) combines two sets represented by x and y and makes z the

[2] A nested SCR is a cycle which is still strongly connected after deleting the headers of all surrounding SCRs.

```
procedure build_intervals(G_CF, START)
a:  build DFST of G_CF using depth-first search from START,
        numbering START 0 and the rest in preorder from 1 to |N_CF| − 1
b:  for w := 0 to |N_CF| − 1 do
        nonCycPreds[w] := cyclePreds[w] := ∅
        header[w]      := 0       // default "header" is START
        reducible[w]   := true
        foreach edge (v, w) entering w do
            if isAncestor(w, v) then add v to cyclePreds[w]
                                else add v to nonCycPreds[w]
    header[0] := nil      // START is root of header tree
c:  for w := |N_CF| − 1 to 1 step −1 do
        P := ∅
d:      foreach node v ∈ cyclePreds[w] do
            if (v ≠ w) then add FIND(v) to P
        Q := P
        while (Q ≠ ∅) do
            select and delete a node x from Q
e:          foreach node y ∈ nonCycPreds[x] do
                y' := FIND(y)
                if not(isAncestor(w, y')) then reducible[w] := false
                                          else if (y' ∉ P) and (y' ≠ w) then
                                                   add y' to P and to Q
        foreach node x ∈ P do
            if (header[x] == 0) then header[x] := w
            UNION(x, w, w)
```

Algorithm 3.1 Building Loop-Nesting Tree representative of the new set. After executing UNION(x, y, z), we have FIND(x) == FIND(y) == z.

As in the original, the first step (a) is to build a depth-first spanning tree of G_{CF}, beginning at START. We number the nodes by the preorder of the DFST, and henceforth equate the name of each node with its number. By saving the number of the last descendent of each node w as last$[w]$, we can easily test for w's being an ancestor of a node v:

$$\text{isAncestor}(w, v) \equiv ((w \leq v) \textbf{ and } (v \leq \text{last}(w)))$$

The next step (b) builds lists of cycle edges and non-cycle edges into each node. Cycle edges go an ancestor in the DFST, others go to descendents or non-ancestors. Actually, since we are only interested in the sources of these edges, we build the sets cyclePreds and nonCycPreds of G_{CF} predecessors which lie along cycle and non-cycle edges, respectively. We initialize the new field reducible[] to **true**.

The loop at (c) steps through the DFST in reverse postorder. The set P represents the loop children of the current node w, the set of nodes in its loop (if any). At (d), cycle predecessors of the current node are added to P. In loop (e), we chase up the non-cycle edges from each cycle predecessor, adding nodes to P, until we reach w. If, before reaching w, we encounter a non-cycle predecessor which lies *above* w in the DFST, then w is one of multiple entry points to an irreducible loop. We select w as the header of the irreducible loop, setting reducible$[w]$ to **false**.

Finally, all the nodes in P are marked with w as their header. The exception is w itself, which will later be marked with the header of the immediately enclosing loop (or with START, if there are no outer loops).

Having built the loop-nesting tree, back-edges are edges going from a node to the header of an enclosing loop. We have frequent need of the forward control-flow graph F_{CF}, which is G_{CF} with loop back-edges removed.

3.2.2 Control Dependence Graphs

The control dependence graph G_{CD} consists of the basic blocks from G_{CF} with a different set of edges.[3] Given a G_{CF} edge e with label L from a branch node B,

[3]Another common form, the *factored* control dependence graph, contains these nodes plus *region* nodes wherever there are control dependence merges [BMO90, FOW87].

procedure *build_control_deps*(G_{CF}, G_{CD})
    $N_{CD} := N_{CF}$
    $E_{CD} := \emptyset$
    foreach edge $(v, w) \in E_{CF}$ **do**
        lab := label of edge
        $y := w$
        ℓ := INDEPENDENT
        while $(y \not\succ v)$ **do**
            z := deepest_common_header(y, v)
            if (ℓ == INDEPENDENT) **and** $(z == w)$ **then**
                ℓ := level(w)
            add (v, y) with level ℓ and label lab to E_{CD}
            y := immediate post-dominator of y

Algorithm 3.2 Building Control Dependences

control dependence edges, also labeled L, go from B to every node that must execute if e is taken [CFR+91].

Algorithm 3.2 gives a method for building control dependences, extended by us to mark each dependence with the loop carrying it [CFS90a]. Loop-carried control dependence edges run from conditional branches out of a loop to the header block of the loop, and are labeled with the level of the header. Note that statements post-dominating the header, but not post-dominating the exit branch, must lie inside the loop. Ignoring the loop-carried edges leaves the acyclic forward control-dependence graph F_{CD} [CFS90b].

3.3 Data Flow

We ultimately require a representation of the flow of values through variables and expression trees that captures the essentials of *what* is computed but disregards *if* it is computed. Such a representation will help us to compare computations that are executed in different contexts. For example, consider performing testing for dependence between array references in the following fragment:

```
do i = 1, 10
  A[i] := ...
  if P then
    ...  := A[i]
```

```
        endif
unaliased enddo
```

It's obvious from looking at the code that the two subscripted references to `A[i]` access the same memory on the same iteration of the loop, and that different iterations access different memory. While we can perhaps refine these facts even further by looking at `P`, we should not allow the presence of control flow to confuse us.

The representations of control flow given above are important in themselves, but they also give us a framework for analyzing the flow of information in a program. The missing part is what happens within a statement (or basic block) and an understanding of variable references.

A basic block consists of a list of simple statements that modify or use variables. The most interesting type of statement for our purposes is the assignment, which does both, and has the form

$$v_0 := f(v_0, ..., v_n)$$

where the v_i are variables and f is an expression with embedded references to the v_i.

We wish to analyze the flow of values between statements so as to build *def-use chains*, linking definitions of each variable v, such as assignments with v on the left-hand side, with uses, such as assignments with v on the right. In Chapter 4, we combine this dataflow graph with the program's expression trees, producing a value graph as a unified representation of how values are computed.

3.3.1 Variable References

We focus on the analysis of scalar values in the absence of aliasing (where multiple names refer to the same memory) and ambiguous references (where the same name applies to multiple memory locations). We can handle some cases of aliasing and ambiguity:

- If a set of overlapping EQUIVALENCEd variables are all scalars of the same size, they are treated as a single scalar variable. Otherwise, they are treated as described below for a single array. FanOut

- Formal arguments (parameters) to procedures are treated as . This is allowed by the Fortran standard. If two variables are later found to be aliased, one or both modified and both referenced, we give up on all computations involving those variables.

- Subscripted array references are treated as compositional functions of monolithic variables [CFR+91].

  ▹ Each array definition is modeled with the `update` function:
  `A[i] := f(j)`
  is treated as
  `A := update(A, i, f(j))`
  where the first argument of `update()` is the array (this use will get edges from the reaching definitions), the second is the subscript, and the third is the new value of the subscripted location(s). The result is taken to be a new monolithic array value.

  ▹ Each array reference is modeled with the `access` function:
  `...  := A[i]`
  is treated as
  `...  := access(A, i)`
  where the first argument of `update()` is the array (this use will get edges from the reaching definitions), and the second is the subscript. The result is taken to be the scalar value of the subscripted element.

- Pointers are not handled. They are not present in Fortran 77. A system which did handle pointers could benefit from a preliminary scalar symbolic analysis (e.g., to recognize pointers that step through memory like auxiliary induction variables).

A commonly used method of connecting definitions of variables with their uses is using traditional bit-vector data-flow analysis [ASU86]. This builds def-use chains linking every definition with all reachable uses of the same variable (where *reachable* implies no intervening definition). A definition at *S1* reaches a use at *S2* if there are one or more control flow paths p_i from *S1* to *S2*, the references are believed to access overlapping memory locations, and for some p_i there is no intervening definition known to overwrite the same memory locations.

By careful insertion of pseudo-definitions, the linking of definitions and uses can be made much simpler and faster, with the added benefit that only a single definition will reach each use.

3.3.2 SSA Form

Converting the original program to static single-assignment (SSA) form makes the process of dataflow analysis simple and efficient. Any interesting program will have points where multiple definitions of a variable reach a particular use:

```
x := f(...)
if (x > 0)
    x := -x
endif
print x
```

Both definitions of `x` reach the use. This may not seem like much of a problem, but if there were n conditionally executed definitions of `x` and m uses, we could end up with $O(nm)$ def-use chains.

Static single-assignment (SSA) form enables efficient construction of program data-flow information [CFR$^+$91]. From constant propagation to register allocation, SSA-based algorithms have proven more powerful and efficient than those based on traditional def-use edges [WZ91, BCT92].

A distinctive feature of SSA form is the placement of a minimal number of pseudo-assignments at program merge points so that no statement, except a pseudo-assignment, is reached by multiple definitions. These pseudo-assignments use ϕ-*functions* to select which of the merged definitions flows to successive uses. Each input to a ϕ-assignment is associated with a unique control-flow edge along which that value reaches the merge. For example, in the following fragment, knowing the value of P would tell us which path to the merge is executed and allow us to prove a constant value for v_3.[4]

$v_1 := 1$
if (P) **then** $v_2 := 2$
$v_3 := \phi(v_2, v_1)$

Note that both v_1 and v_2 may both execute; we must know not only which nodes but which edges execute in order to select the correct input [WZ91].

The necessity of examining control-flow edges when interpreting SSA form limits its utility for many purposes. For example, global value numbering techniques based on standard SSA form cannot safely find a_3 and c_3 equivalent in the following fragment.

[4] We use the subscripted variable name v_i to indicate definition i of v or the value of that definition.

$a_1 := 1;\ b_1 := 1;\ c_1 := 1;$
if (P) **then** $a_2 := 2$
$a_3 := \phi(a_2, a_1)$
if not (P) **then** $b_2 := 2$
$b_3 := \phi(b_2, b_1)$
if (P) **then** $c_2 := 2$
$c_3 := \phi(c_2, c_1)$

Both a_3 and c_3 have the same value, provided P does not change. However, to pattern-match only on the ϕ-functions and their inputs would mistakenly find b_3 equivalent to a_3 and c_3. To be conservative, we must assume that ϕ-assignments at different program points have independent values [AWZ88].

SSA form was introduced by researchers at IBM and at Brown University [RWZ88, AWZ88]. The ϕ-assignments are direct descendants of Reif *et al.*'s birthpoints, used to refine def-use chains for symbolic analysis [RT81, RL86].

The construction of SSA form makes it a good basis for a dataflow graph. Pseudo-assignments of the form $v_\phi := \phi(v_1, \cdots, v_n)$ are inserted so that exactly one definition of a variable v reaches any non-ϕ use of v. The ϕ-assignments are placed at the earliest control flow merge where multiple definitions reach. For a given ϕ-assignment, there is one-to-one correspondence between arguments to the ϕ and in-edges of the control flow merge. This allows us to annotate each G_{DU}^{SSA} edge e into a ϕ with e.*CfEdge*, the last G_{CF} edge on the path by which its definition reaches the control flow merge.

Algorithm 3.3 gives the method for adding ϕ-assignments, modified for construction of GSA form as described later. While the worst-case complexity is cubic in the size of the program, SSA construction takes linear time in practice [CFR+91]. Edges linking definitions to uses are added during a walk over the pre-dominator tree, in time proportional to the number of edges [CFR+91]. We keep a stack for each variable of the definitions encountered as we walk down from the root.

- At a definition of v, push the index of that definition onto the stack for v.
- At a non-ϕ use of v, add an edge from the top definition of the stack for v.
- At each node n, for each edge to a node m with $(n \not\prec m)$, add edges to the ϕ-assignments at m from the top definitions of the appropriate stacks. These def-use edges must be associated with the edge (n, m), either explicitly, by a map, or implicitly, by their ordering.

- When returning from a subwalk, pop any definitions that were at the current node.

Linking definitions to uses in SSA form is often referred to as *renaming*, because each use of a variable is now reached by a unique definition, so that the different live ranges (a definition and its uses) can be thought of a separate variable [CFR+91].

3.4 GSA Form: Dataflow and Conditionals

Gated Single-Assignment (GSA) form extends SSA form with information about branch predicates controlling merges. That is, the most visible difference is the replacement of ϕ functions with new functions giving not just the merged definitions but the predicates determining which definition reaches. This extension enables GSA form to represent some, but not all, control information:

- It does represent which control over which definitions reach which statements, assuming those statements are executed.
- It does *not* represent control over which statements execute.

For example, in the following:

```
read a0
if P:(a0 < 0) then
    a1 := -a0
endif
a2 := gamma(P, a1, a0)
```

we have a complete representation of the conditions under which each definition reaches `a2`. Examining the statement defining `a1`, GSA form gives us no direct way to tell us whether or not that statement executes.

3.4.1 Definition of TGSA Form

Alpern *et al.* introduced the first gated version of static single-assignment form as high-level SSA form [AWZ88]. Ballance *et al.* introduced the terminology of gated single-assignment form [BMO90]. TGSA form is an extension of high-level SSA form to unstructured code, for which we use the more convenient GSA-form notation [Hav93]. Detailed comparisons with the prior versions are given in Section 3.6.

Building TGSA form involves adding pseudo-assignments for a variable V:

(γ) at a control-flow merge when disjoint paths from a conditional branch come together and at least one of the paths contains a definition of V;

(μ) at the header of each loop that contains at least one definition of V; and

(η) at every exit from each loop that contains at least one definition of V.

The first two cases, merges in forward control flow and at loop entry, are handled with ϕ nodes in SSA form. The third case, merging iterative values at loop exit, is ignored by SSA form.

Gamma: Merge with Predicate. The pseudo-assignment inserted for a dataflow merge in forward control flow employs a γ function. In a simple case, the fragment

$V_1 := ...$
if (P) **then** $V_2 := ...$

is followed by $V_3 := \gamma(P, V_2, V_1)$. This indicates that if control flow reaches the merge and P was true, then V has the value from definition V_2.

We insert γ functions only to replace ϕ functions at merges of forward control flow. Whenever otherwise disjoint F_{CF} paths from an n-ary conditional branch (with predicate P) come to a merge, and at least one path redefines a variable V, we insert a definition $V' := \gamma(P, V_1, ..., V_n)$ at the merge point. Each of the V_i is the last definition (possibly another pseudo-assignment) occurring on a path from subprogram entry, through the branch, to the merge. The V_i are called the value inputs, P the predicate input. A γ is strict in its predicate input only; when the predicate has value i, the quantity from the i^{th} value input is produced.

In unstructured code, paths from multiple branches may come together at a single merge. Replacing the ϕ then requires multiple γ functions, explicitly arranged in a dag. Such a γ dag is structured so that the immediate pre-dominator of the merge provides the predicate for the root of the dag, and paths in the dag from the root to the leaves pass through γ functions with predicates from branches in the same order that F_{CF} paths from the immediate pre-dominator to the merge pass through the branches themselves.

If some edges from a branch cannot reach a merge, then the corresponding locations in γ functions are given as $\top$, indicating that no value is provided. We disallow the case where all non-predicate arguments save one are $\top$, as in

$$\gamma(P, \top, ..., \top, V_i, \top, ..., \top)$$

If execution reaches the merge, then definition V_i must be providing the value. This simplification is one difference between *thinned* GSA form and original GSA form, which we exploit in our algorithms for building TGSA form.

Mu: Loop Merge. A pseudo-assignment at the header of a loop uses a μ function. For each variable V defined in a loop body, we insert a definition $V' := \mu(V_{init}, V_{iter})$ at the loop header, where V_{init} is the *initial input* reaching the header from outside ($\mathtt{I}_0$ in Figure 3.3) and V_{iter} is the *iterative input*, reaching along the back-edge ($\mathtt{I}_2$). If every loop header has one entry edge and one back-edge, then every ϕ at a loop header can be replaced exactly by one μ (otherwise, a γ dag may also be required to assemble one or both of the two inputs).

Since TGSA form is organized for demand-driven interpretation, there is no control over values flowing around a loop. A μ function can be viewed as producing an infinite sequence of values, one of which is selected by an η function at loop exit. (In original GSA form, μ functions have a predicate argument controlling iteration.)

Eta: Loop Value Selection. Pseudo-assignments inserted to govern values produced by loops use η functions. Given a loop that terminates when the predicate P is true, we split any def-use edge exiting the loop and insert a node $V' := \eta(P, V_{final})$, where V_{final} is the definition ($\mathtt{I}_1$ in Figure 3.3) reaching beyond the loop. In general, a control-flow edge may exit many loops at once; for a variable that has been modified in the outermost L loops exited, we must insert a list of L η functions at the sink of the edge. The η function in TGSA form corresponds to the η^T function in original GSA form.

Loop-variance Level. The *level* of a node in G_{DU}^{GSA} is the nesting depth of the innermost loop with which it varies (or zero, for loop-invariant values). The level of a μ function equals the depth of the loop containing it; the depth of an η function equals the depth of the loop exited less 1. For any other function, the level is the maximum level over all its inputs.

Level information is required to distinguish μ functions that vary with different loops in a nest. For other nodes, level information is just a convenient annotation.

Interpretation. We give no formal semantics for the interpretation of TGSA form. In Chapter 4, we define a value graph combining GSA-form def-use chains with the

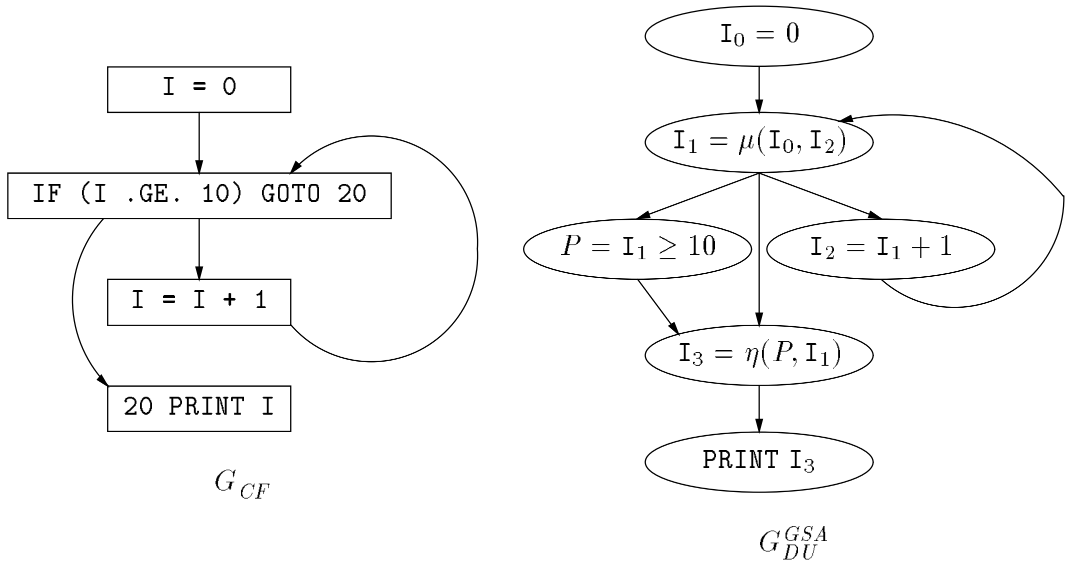


Figure 3.3 Loop Representations

procedure's expression trees. Expressions represented by *congruent* (isomorphic) subgraphs have the same value when they both execute, provided that they are loop-invariant or that both executions are on the same iteration of corresponding loops.

3.4.2 Construction of TGSA Form

We present algorithms for the two crucial steps in converting from SSA form to TGSA form: replacing a ϕ function with a directed acyclic graph of γ functions and building the controlling predicate of a loop for use in its η functions.

Augmenting G_{CF} and G_{DU}^{SSA}

For convenience, we assume that every G_{CF} node is reachable from Start and reaches End (neither unreachable nor dead-end code exists). Loops are assumed reducible, so that all cycles pass through a unique header. (In the implementation, irreducible loops are conservatively left alone.) We build tree representations of loop nesting and of the pre- and post-dominator relations [LT79, Tar74]. By saving a preorder numbering of each tree, we can test for transitive as well as immediate loop nesting and dominance in constant time.

We augment loops, as shown in Figure 3.4, to provide suitable places for the placement of μ and η nodes. For each loop we add a preheader node (`PH`), if the

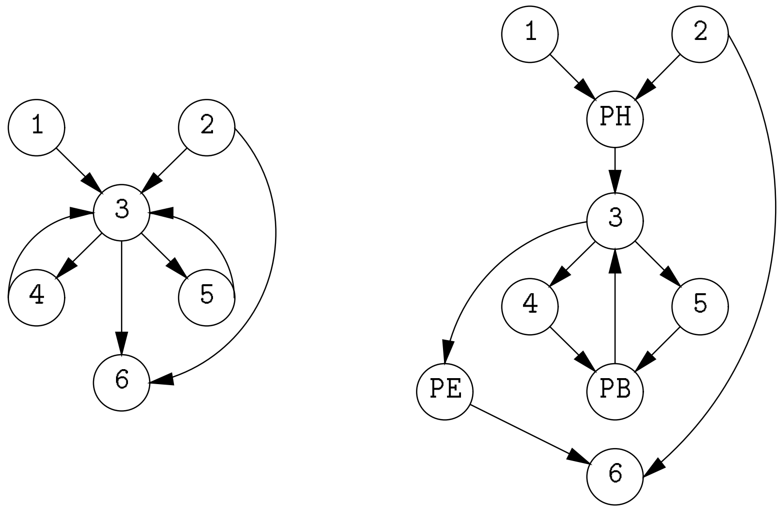


Figure 3.4 Insertion of Preheader, Postbody and Postexit Nodes

header has multiple forward in-edges, and a postbody node (PB), if the header has multiple backward in-edges or if the source of the back-edge is in an inner loop. The postbody node for each loop then terminates each complete iteration, and is post-dominated by the header. Wherever the sink of a loop-exit edge has other in-edges, we split the loop-exit edge with the addition of a postexit node (PE) [CFS90b]. These changes cause threefold growth in G_{CF} at worst.

The dominance and loop nesting trees are easily updated, and control dependence construction proceeds normally on the augmented G_{CF} [CFS90a]. As none of the new nodes are branches, control dependence edges among the original G_{CF} nodes are unaffected.

Construction of SSA form proceeds normally except for tweaks to the ϕ placement phase, given in Algorithm 3.3. Having ensured that each loop header has exactly two in-edges, we place μ nodes at loop headers instead of ϕ nodes. Every μ node added represents a variable modified in the loop, and we immediately add an $\check{\eta}$ ("eta-hat") definition for the same variable at the postexit node for the loop.[5] These trivial pseudo-assignments ($V' := \check{\eta}(V)$) hold the place for later creation of η functions with predicates, and ensure proper placement of ϕ assignments for merges of values from different exits.

[5]If the postexit node is shared with a surrounding loop, we add nothing, as the appropriate $\check{\eta}$ will be added for the outer loop.

procedure *build_SSA*(G_{CF}):
    build dominance frontiers;
    insert_ϕs(G_{CF});
    build def-use chains G_{DU}^{SSA};

procedure *insert_ϕs*(G_{CF}):
    foreach variable V **do**
        Worklist := {};
        foreach assignment *Def* to V **do** *Worklist* += *Def*;
        while *Worklist* $\neq$ {} **do**
            remove statement X from *Worklist*;
            foreach Y in X's dominance frontier **do**
                if there is no pseudo-def of V at Y **then** *add_pseudo_def*(V, Y);

procedure *add_pseudo_def*(V, Y):
    if Y is a loop-header statement **then**
        add a μ for V at Y;
        foreach exit from Y's loop **do** add an $\check{\eta}$ for V at the loop exit node;
    else add a ϕ for V at Y;

Algorithm 3.3 Modified SSA Form Construction

Once ϕ, μ, and $\check{\eta}$ assignments have been placed, the linking of definitions to uses (often referred to as *renaming* in the SSA literature) proceeds as for standard SSA form, producing an variant of G_{DU}^{SSA}. Def-use edges are loop-independent unless they go to a μ assignment at the header of a loop from a definition inside the loop, in which case their level is the nesting depth of that header.

Data Structures

With μ and $\check{\eta}$ assignments out of the way, the main work of building TGSA form lies in converting the remaining ϕ functions to dags of γ functions, and building other γ dags to represent loop-termination conditions.[6] Our solutions for these similar problems share many data structures, described below.

The procedure *replace_ϕs()*, shown in Figure 3.4, examines one ϕ function at a time and replaces it with a dag of γ functions. The γ dag has the same dataflow

[6]These algorithms are described in terms of building G_{DU}^{GSA}; they also can be used to build the GSA-form value graph on the fly.

predecessors and successors as the original ϕ, except that each γ function also takes the result of a branch predicate as input. All of the γ functions in the dag replacing a ϕ are placed at the beginning of the basic block where the merge happens (ϕ•*Block*).

Likewise, *build_loop_predicate()* proceeds one loop at a time. The γ functions making up the dag for the loop-termination condition are placed at the postbody for the loop.

Branch•Choices: for a G_{CF} node *Branch*, a vector of one choice per out-edge. Each choice represents a value that will reach the merge if the branch goes the right way. A choice is either a leaf definition (pointing to a value computed at another G_{CF} node), another branch (representing the value chosen there), or $\top$ (indicating a choice to avoid the merge node).

In ϕ-replacement, a leaf definition is the choice if all the paths through the corresponding out-edge to the ϕ merge point pass through the same final definition. This chosen definition may occur before or after *Branch*. If there are instead multiple final definitions on these paths, then the branch choosing between them is specified as the choice for this out-edge. Such a branch must lie between *Branch* and the ϕ merge point. If no path through an out-edge reaches the ϕ merge point, then its corresponding choice entry is $\top$.

When building loop predicates, a leaf choice is **true** if an out-edge must lead to the postbody or **false** if it cannot. If some paths through the out-edge lead to the postbody and some do not, then a branch choice involved in the decision is specified.

Selectors: a set of G_{CF} branch nodes whose predicates will be used in building the γ dag for the current loop predicate or ϕ.

In ϕ-replacement, a branch is added to *Selectors* if and only if it affects, directly or indirectly, the choice of definition reaching the control flow merge. We add the branch to *Selectors* when we detect that two of its *Choices* are different and not $\top$. More precisely, the selectors of a ϕ assignment comprise the set of $n \in N_{CF}$ such that (1) $\exists$ at least two paths in F_{CF}, from n to ϕ•*Block*, disjoint except for their endpoints, and (2) $\exists$ such a path containing a definition of the ϕ's variable not at n or at ϕ•*Block*.

```
procedure replace_φs(Merge):
    Predom := Merge•Ipredom;
    foreach φ at Merge do
        foreach CfEdge entering Merge in F_CF do init_walk(CfEdge•Source);
        Selectors := {};
        foreach DefEdge entering φ in F_DU do
            CfEdge := DefEdge•CfEdge;    Def := DefEdge•Source;
            process_choice(Def, CfEdge•Label, CfEdge•Source);
        recursively build a dag γ from Predom.Choices;
        replace φ with γ;

procedure init_walk(Node):
    if (Node•Fanout > 1) then
        Node•Visits++;
        if (Node•Visits == 1) then Node•Choices[*] := Node•FirstChoice := ⊤;
        if (Node == Predom) or (Node•Visits ≠ 1) then return;
    /* on our first visit to non-Predom */
    foreach edge Cd entering Node in F_CD do init_walk(Cd•Source);

procedure process_choice(Choice, Label, Node):
    NewChoice := Choice;
    if (Node•Fanout > 1) then
        Node•Choices[Label] := Choice;    Node•Visits−−;
        if (Node•FirstChoice == ⊤) then Node•FirstChoice := Choice;
                                   else  if (Node•FirstChoice ≠ Choice) then
                                             Selectors += Node;
        if (Node == Predom) or (Node•Visits ≠ 0) then return;
        if (Node ∈ Selectors) or
          ((not Thinned) and (Merge ∉ post-dominators(Node))) then
            NewChoice := Node;
    /* on our last visit to non-Predom */
    foreach edge Cd entering Node in F_CD do
        process_choice(NewChoice, Cd•Label, Cd•Source);
```

Algorithm 3.4 Replacing a ϕ with a dag of γs

When building loop predicates, *Selectors* contains all branches inside the loop whose immediate post-dominator is outside the loop (i.e., the nodes within the loop on which the postbody is transitively control dependent).

Branch•*FirstChoice*: used only in *replace_ϕs()*; it saves the first choice propagated back to the branch. Each choice propagated back to *Branch* is compared against this field. If the *FirstChoice* field already holds a different value besides $\top$, then another choice has already been propagated, and we add *Branch* to *Selectors*.

Branch•*Visits*: a counter, used only in replacing a ϕ. Initialized to zero, the multiple calls to *init_walk* increment *Visits* for branch nodes until it reaches the number of edges from *Branch* that can lead to the merge. Later, calls to *process_choice* decrement *Visits* each time an edge from the *Branch* is processed. When *Visits* reaches zero again, then all paths from *Branch* to the merge have been examined. We then propagate to *Branch*'s G_{CD} predecessors the appropriate choice: *Branch* itself, if it is a selector, else the one non-$\top$ value from its choices.

CfNode•*Exits* This is the γ dag representing when the loop exits. While we could compute distinct conditions for each exit, it is sufficient to determine termination test for the entire loop.

Miscellaneous:

CfNode•*Ipredom* = immediate pre-dominator;
SsaDuEdge•*CfEdge* = final G_{CF} edge along which a definition reaches a ϕ;
CfEdge•*Label* distinguishes edges sourced in the same conditional branch;
dominators *CfNode*• = the number of out-edges from a G_{CF} node.
Thinned is true for all examples given here; when false, extra $\top$ entries are left in γ functions.

γ Conversion

The routines in Figure 3.4 show how to replace a ϕ function with a dag of γ functions. We call *replace_ϕs()* for each node in topological order on F_{CF}, and it replaces any ϕ functions located there. The replacement proceeds one ϕ at a time. Auxiliary routines visit each control-dependence ancestor of F_{CF} predecessors of ϕ•*Block* in two passes. Consider their effects during *replace_ϕs(*`A3`*)* for the only ϕ assignment in Figure 3.5.

```
   a0 := ...
   if (x) 10, 20, 30
10 if (q) goto 99
   a1 := ...
   goto 40
20 if (p) goto 10
30 if (s) a2 := ...
40 a3 := φ(a1, a2, a0)
   ...
```

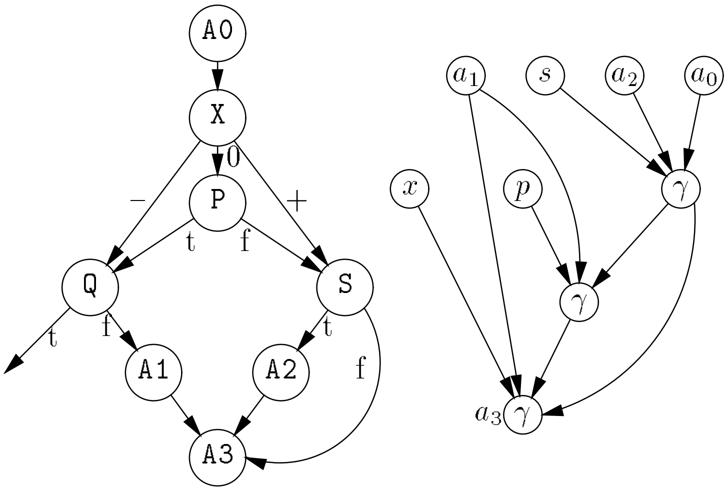


Figure 3.5 SSA-form Source, G_{CF}, and γ dag for Example

The first pass, by recursive calls to *init_walk()*, initializes fields for each node that might be a branch point for distinct paths to the merge. *Node*•*Visits* counts the number of visits in this walk; we will count back down on the next walk. At the end of this pass, the values for *Visits* in the example are: 1 for `Q` (because the other out-edge cannot reach `A3`), 2 for `S`, 2 for `P`, and 3 for `X`. All choice vectors and *FirstChoice* fields are $\top$.

The second walk routine, *process_choice()*, propagates each reaching definition backwards from the ϕ until a branch is encountered. It is then entered as a choice for that branch. If another, different choice has already been entered (*Node*•*FirstChoice*), then the branch *Node* is added to *Selectors* for this ϕ. When we have finished all visits to a branch, we then push a choice (the branch itself, if it has been marked a selector) upwards from this branch to its controlling branch.

If we process G_{DU} edge (a_1, a_3) first, we execute *process_choice(*a_1*,* **nil***,* `A1`*)*. As this recurses, we decrement `Q`•*Visits* to 0, and continue propagating a_1 as the choice, since that was `Q`'s first and only choice. When we return to the top level, we have `P`•*Choices* $= [a_1, \top]$ with one visit left, and `X`•*Choices* $= [a_1, \top, \top]$ with two visits left. (However, *Visits* is not always the same as the number of $\top$ choices left.)

If we next process G_{DU} edge (a_2, a_3), we execute *process_choice(*a_2*,* **nil***,* `A2`*)*. We then call *process_choice(*a_2*,* **true***,* `S`*)*. We return having `S`•*Choices* $= [a_2, \top]$, with one visit left.

The last edge processed at the top level is (a_0, a_3), and causes us to call *process_choice(*a_0, **true**, `S`*)*. This finishes off `S`'s visits and makes it a selector, and finishes off `P` and `X` directly. We are left with *Selectors* $= \{\mathtt{S}, \mathtt{P}, \mathtt{X}\}$, with their choice vectors $[a_2, a_0], [a_1, \mathtt{S}]$, and $[a_1, \mathtt{P}, \mathtt{S}]$, respectively.

When the second walk is done, *Node*•*Choices* is fully defined for each selector. We then read off the γ dag starting at *Predom*. The γ function at the root of the dag takes its predicate input from the predicate controlling *Predom*'s branching, and its value inputs from the corresponding entries in the choice vector. Choices that are leaf definitions are made direct data inputs to the γ; a choice that is a selector is replaced with a γ function built recursively, in the same fashion, from the predicate and choice vector for the selector. The resulting γ dag for a_3 is shown rightmost in Figure 3.5.

We give no formal proof for the correctness of this procedure. Informally, one can easily show that *Selectors* is properly computed. To show correctness of the γ dags, we prove a one-to-one correspondence between paths in the γ dag replacing a ϕ and sequences of selectors encountered in execution paths reaching ϕ•*Block* in G_{CF}, based on the labels produced by the predicates at the selectors. The last definition on a G_{CF} path from START to ϕ•*Block* is the same as the leaf definition selected by the corresponding path through the γ dag.

η Construction

Algorithm 3.5 shows the procedures to build the γ dag for a loop-termination predicate. For each loop header, we pass its unique back-edge predecessor (its postbody node) to *build_loop_predicate()*. The γ dag built represents the negation of its control conditions, which correspond to the conditions for loop exit. We store this condition as *Loop*•*Exits*.

We now expand each $\check{\eta}$ — inserted earlier, one per loop-exit G_{CF} edge — so that there is one η per loop exited, and make the predicate input of each η refer to the loop-termination predicate for the corresponding loop. Each η takes input from the η from the next-inner loop exited on the same G_{CF} edge, and feeds the η for the next-outer loop exited, except that the innermost η gets the input of the original $\check{\eta}$ and the outermost η feeds the original uses of the $\check{\eta}$.

```
procedure build_loop_predicate(End):
    Loop := End•Header;    Selectors := {};
    process_cds(End);
    Predom := Loop;
    while (Predom•Ipostdom pre-dominates End) do Predom := Predom•Ipostdom;
    recursively build a dag from Predom.Choices;
    save negated dag as Loop•Exits;

procedure process_cds(Node):
    foreach edge Cd entering Node in F_CD do
        Pred := Cd•Source;
        if (Pred == Loop) or (Loop pre-dominates Pred) then
            if (Pred ∉ Selectors) then /* assume all branches exit */
                Pred•Choices[*] := true;
                Selectors += Pred;
                process_cds(Pred);
            if (Node == End) then Pred•Choices[Cd•Label] := false; /* non-exit */
                             else /* possible exit path */
                                 Pred•Choices[Cd•Label] := Node;
```

Algorithm 3.5 Building a loop-termination predicate

3.5 Efficiency

We extended the ParaScope system to build control-flow graphs, dominator trees, loop-nesting trees, control-dependence graphs, SSA-form def-use chains and TGSA-form def-use chains. We focus on the asymptotic and empirical performance of GSA form, because the prior work used a different algorithm and gave no experiments [BMO90].

Per Procedure		Mean	90^{th} Quant.	99^{th} Quant.	Max
Max branch arity	c_b	1.9	3	7	22
Max unstruct. depth	c_u	4.0	9	16	31
Max loop(exits times depth)	$c_x c_l$	2.0	8	21	156

Table 3.1 Control-Flow Statistics of Benchmarks

3.5.1 Control-Flow Characteristics

The time and space required to build TGSA form are closely related to the size of SSA form, especially if reasonable bounds hold for a few procedure control-flow characteristics: the fan-out of conditional branches (c_b), the depth of loop nesting (c_l), the number of exits from any single loop (c_x), and the length of any F_{CD} path from the immediate pre-dominator of a merge node to an F_{CF} predecessor of the merge (also referred to here as "the depth of unstructured control," c_u). Our notation reflects the belief that these quantities can be treated as small constants.

Table 3.1 give the maximum and mean, over all the benchmark procedures of Appendix A, of each procedure's maximum branch fanout, maximum depth of unstructured code, and maximum loop exit branches times loop headers. While the average maximums is quite low for all three measurements, the maximum over all procedures is high. The value at the 90^{th} quantile is still low. While not all the procedures with large values for these quantities showed extra-linear growth in analysis time, those that did show extra-linear growth had large values.

3.5.2 Asymptotic Complexity

Here we reason about the asymptotic complexity of these methods, as related to various program metrics. While our methods may take exponential time and space for contrived examples, they take linear time for programs satisfying loose and reasonable structural requirements.

Auxiliary Analyses and Data Structures.

Efficient methods exist for computing each of pre- and post-, Tarjan intervals and control dependences [LT79, Tar74, CFR+91, CFS90a]. While the asymptotic time bounds range from almost linear to quadratic, for practical purposes, these methods are linear in the number of G_{CF} edges (E_{CF}).

The addition of blocks and edges to G_{CF} is bounded by the number of loop headers and loop-exit edges. The total observed growth in G_{CF} nodes was eight percent; the maximum growth in any procedure was a factor of two.

The addition of ϕ functions for SSA form affects the size of the dataflow graph. While the number of additional ϕ assignments *can* be quadratic in the number of original references, it is linear throughout extensive practical experiments [CFR+91]. The number of edges in G_{DU}^{SSA} is linear in the number of references if the arity of merges is constant. In practice, G_{DU}^{SSA} should be far smaller than traditional def-use chains, which are often quadratic in the number of references. Our results agree with the prior literature here.

Auxiliary data structures required to build GSA form from SSA form, such as the vectors of choices, are reused by different calls to *build_loop_predicate()* and *replace_ϕs()*. Their total size is $O(E_{CF})$.

μ and η Functions.

There is one μ per loop per variable defined in the loop, which is exactly the number of ϕ functions at loop headers in standard SSA form. There are two cases of η functions. For variables defined in a loop, we need create only one η per level exited, so those η functions are at most $c_x c_l$ times as numerous as the μ functions. Other η functions must be created for predicates, when multiple loop-exit branches merge outside the loop. At the very worst, there are $c_x c_l$ of these η functions per γ dag (*i.e.*, per old ϕ). So the total number of η functions is $O(c_x c_l(\#\mu))$. Our experiments confirm this tight linear relationship, with an average of 1.5 η functions per μ. However, there is wide variation in the number of μ and η functions per loop, presumably because of differences in loop size.

The predicate input of an η refers to the corresponding loop's termination predicate, a γ dag with at most $(c_b - 1)$ γ functions for each block in the loop. (This worst case would require that every block end with a a multi-way loop exit.) The termination predicate is built in time proportional to its size. Since each block can be in at most c_l loops, the total size of all loop-termination predicates is $O(c_b c_l N_{CF})$. Our measurements do not distinguish γ functions in loop predicates from those replacing ϕs.

γ Functions.

In structured code, there is only one merge point for each branch (i.e., paths from a branch merge only at its post-dominator). In this situation, the γ dag replacing each ϕ has exactly the same inputs and space requirements as the ϕ, except for the addition of the predicate inputs.

In unstructured code, every branch between the merge point of a ϕ and the predominator of the merge can potentially contribute to the size of the γ dag. A coarse bound for the maximum dag size in this case is $min(N_{CF}, {c_b}^{c_u})$. For structured code,

${c_b}^{c_u}$ is smaller, so the time and space required to build the γ functions is linear in the number of ϕ functions. For unstructured code (large c_u), N_{CF} is smaller, so the resource requirements can be at worst quadratic ($O(N_{CF}N_{DU}^{SSA})$).

3.5.3 Experimental Performance

We tested our implementation on over 1000 scientific Fortran programs described in Appendix A.

Figure 3.6 plots the total size of G_{DU}^{SSA} (nodes and edges, with μ and $\check{\eta}$, but with not γ conversion) against the number of plain variable references (i.e., programmer-specified definitions and uses). While there is some spread, the relationship appears to be linear. The significant outlier is `master` from the RiCEPS program `boast`, with over 30 SSA nodes and edges per plain reference. This routine contains one loop with an exceptionally large number of loop exit branches, with the second highest value for the maximum product of loop exits and loop depth ($c_x c_l == 81$)

Figure 3.7 compares the total size (nodes and edges) of G_{DU}^{GSA} and G_{DU}^{SSA}. The total growth over all procedures was 18 percent. The outlier `mosfet`, from the SPEC version of `spice`, had a the third highest maximum depth of unstructured control ($c_u == 27$).

Figure 3.8 plots the total size of G_{DU}^{GSA} divided by the number of plain variable references. Only a very few small procedures have more than 20 TGSA elements per plain reference. The trend on this graph seems flat, implying a linear relationship between the size of TGSA form and the size of the program.

The total time required to build everything from G_{CF} to G_{DU}^{GSA} is plotted as milliseconds per G_{CF} node and edge in Figure 3.9. This graph is also quite flat, suggesting a linear relationship between analysis time and program size. The average time required per G_{CF} element on the entire sample was 2.4 ms; the average of the procedure averages was 3.1 ms.[7]

These results confirm prior results that SSA form has time and space requirements linear in the size of the procedure [CFR+91]. They also show that GSA form is generally about the same same size as SSA form.

---

[7]These measurements were taken inside the ParaScope programming environment, optimized with `gcc` version 2.4.5, running on a Sun MicroSystems Sparc 10 with 64 Mbytes of memory.

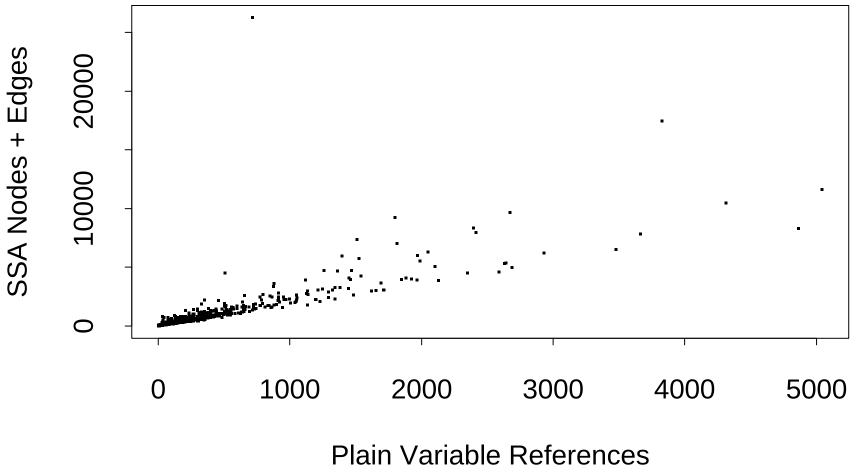


Figure 3.6 SSA size vs. Plain References

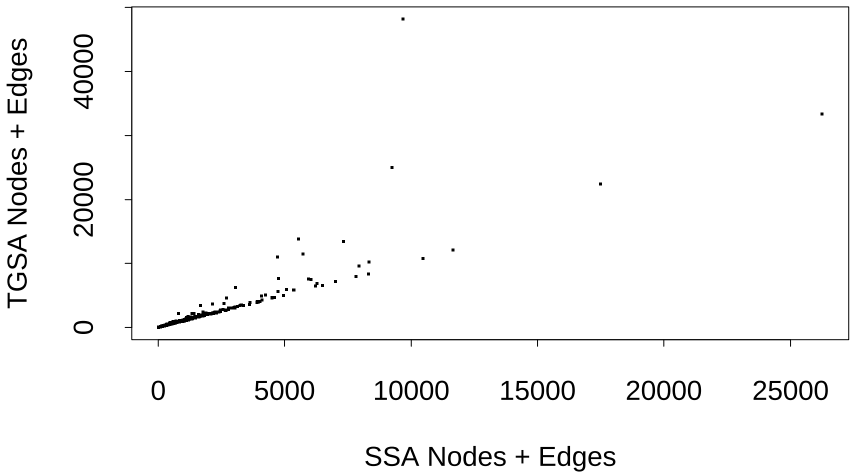


Figure 3.7 Comparing size of TGSA and SSA forms

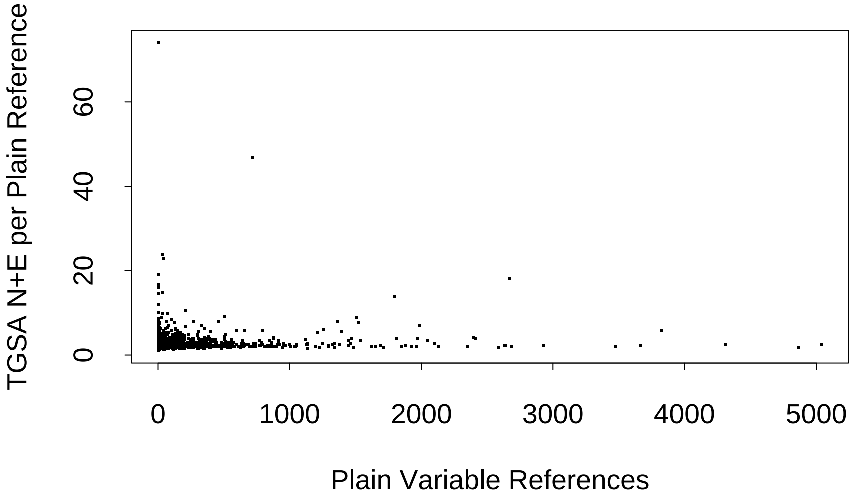


Figure 3.8 TGSA Elements per Plain Reference

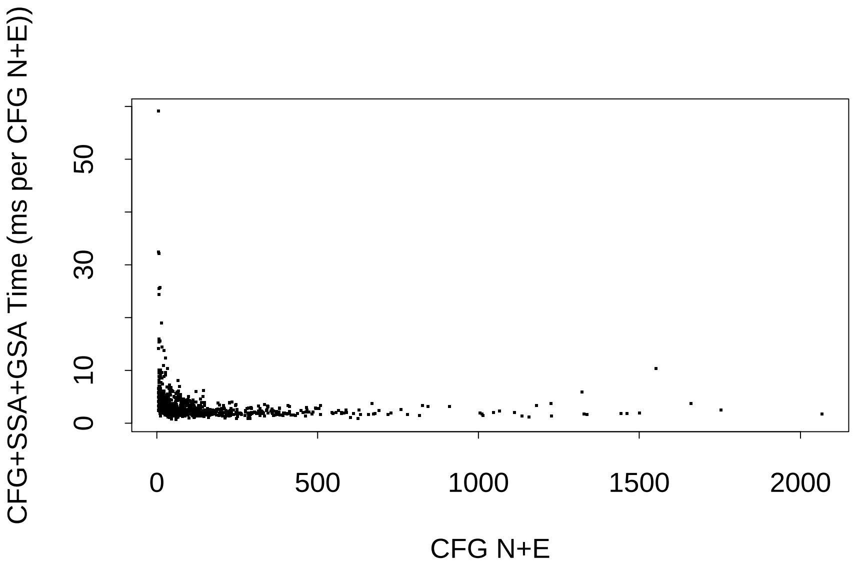


Figure 3.9 Total Analysis Time per CFG Element

3.6 Related Work

3.6.1 Groundwork

Control-flow graphs, dominator trees, control-dependence graphs and SSA form are covered extensively in the literature [ASU86, LT79, CFR+91]. However, while construction of loop-nesting trees is well-known tool, it is seldom documented. We use the algorithm of Tarjan's algorithm for testing reducibility [Tar74] extended to recognizing irreducible loops in a way that isolates the problems they cause.

Whether a particular control dependence is loop-independent or loop-carried may seem obvious in sufficiently structured code. Our algorithm is the first we have seen to compute the carrying level for control dependences in any reducible control-flow graph. (We also compute these levels in irreducible graphs, but their interpretation is more problematic when the carrying loop is irreducible.)

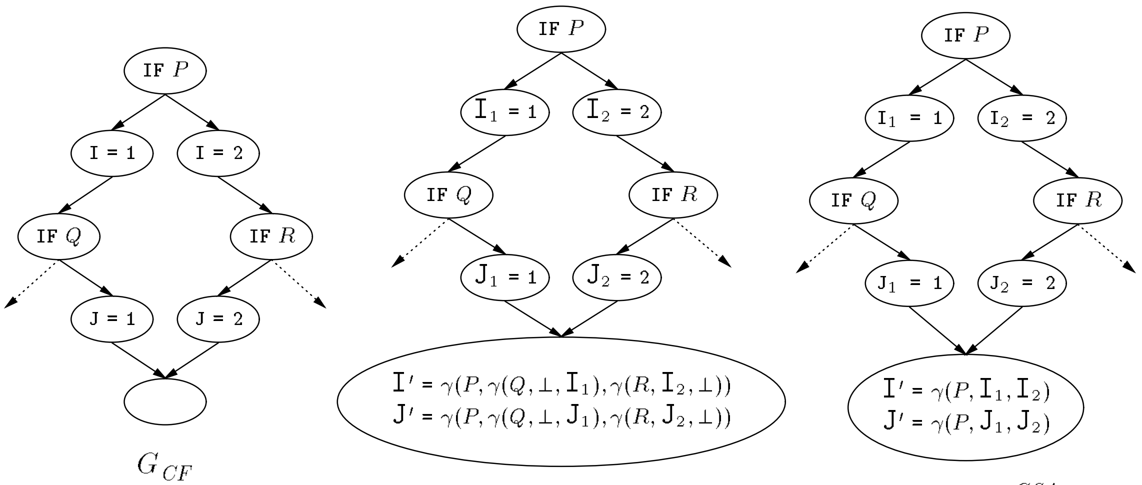
3.6.2 High-level SSA form

Alpern *et al.* presented SSA form and proposed extensions to improve handling of conditional merges and loops [AWZ88]. For structured programs, their $\phi_{\texttt{if}}$, $\phi_{\texttt{enter}}$, and $\phi_{\texttt{exit}}$ are exactly equivalent to the TGSA-form versions of γ, μ, and η, respectively. They showed how to find the partition of nodes in a *value graph* based on SSA form that gives the maximum number of congruent nodes (equivalent expressions) without rewriting the graph.

TGSA form is identical to high-level SSA form for structured code. The same value partitioning methods apply to TGSA form, extending these results to unstructured programs.

3.6.3 Original GSA form

Ballance *et al.* introduced GSA form as a component of their program dependence web (PDW) [BMO90]. GSA form was inspired both by high-level SSA form [AWZ88] and PDGs with valve nodes [CF89].



G_{CF} Ballance *et al.* G_{CF}^{GSA} Thinned G_{CF}^{GSA}

Figure 3.10 Different versions of GSA form on G_{CF}

Original GSA form has a third input to the μ function, the loop-continuation predicate. In addition, it allows γ functions to have a single non-$\top$ value input, as in Figure 3.10.[8]

Ballance *et al.* need this additional information to drive the insertion of switches, creating the data-driven form of their PDW. For symbolic analysis, it is sufficient to control values flowing out of a loop, and the predicates for η functions in thinned GSA form satisfy this purpose. Omitting the predicates on μ functions clarifies the independence of iterative sequences, such as induction variables, from the iteration count.

Extra γ functions can obscure equivalences. The original form's γ dags include both the predicates determining which definition reaches the merge *and* those determining whether or not the merge would execute.[9] In Figure 3.10, if $\mathtt{I}_1$ and $\mathtt{I}_2$ were equal, it would be much easier to notice the simplification for $\mathtt{I}'$ in TGSA form than in the original GSA form.

[8] Actually, they use $\perp$, which makes sense in their dataflow interpretation, where unexecutable expressions are treated as divergent. In a demand-driven interpretation, unexecutable expressions can be assumed to have any value ($\top$), since that value will never be demanded.

[9] Because conditions for branches preceding the pre-dominator of the merge are implied through the predicate of the γ, the extra functions occur only for unstructured merges.

Original GSA form can be converted to thinned GSA form by globally simplifying the extra γ functions and ignoring the predicate inputs to μ functions. While both algorithms for building GSA form seem to have the same asymptotic complexity, ours is simpler to implement, especially when working with an unfactored control dependence graph.

The Program Dependence Web researchers have recently developed another revised definition of GSA form, which retains the major differences between with thinned GSA form described above [CKB93].

3.6.4 Program Dependence Graphs

Ferrante, Ottenstein and Warren introduced the program dependence graph, comprising data dependences and their now-standard formulation of control dependence [FOW87]. Groups at Wisconsin and Rice have developed semantics for PDGs [HPR88, CF89, Sel92]. Selke gives semantics for programs with arrays and arbitrary control flow, and shows how to insert valve nodes efficiently.

Despite the greater maturity of PDGs as a program representation, they are inferior to TGSA form for value numbering. The inputs to each node include the control dependences under which that node executes, which would make value numbering overly conservative for dependence testing. However, ignoring control dependences in value numbering would produce results similar to those for SSA form; we would be unable to compare merged values from different parts of the program.

Perhaps rewriting systems can be defined for PDGs that have the same power as value numbering, but the practical efficiency of such an approach is uncertain.

3.6.5 Semantics

Recent work by John Field gives a formal treatment of graph rewriting on a representation resembling original GSA form [Fie92]. Field's work still focuses on data-driven models of execution, where control dependence is converted to data dependence. (Doing this by use of control dependences is much more efficient than IF-conversion.)

The program representation graph of Horwitz *et al.* also resembles GSA form [YHR89], and is also used to recognize a form of equivalence between program fragments.

3.7 Summary

The major result of this chapter is the introduction of thinned gated single-assignment (TGSA) form. Like SSA form, TGSA form adds pseudo-definitions in carefully selected places so that there are fewer def-use chains and they are easier to build.

All versions of GSA form support the conversion of conditionally merged values to a functional form. The originally published algorithms for GSA form are based on a factored control dependence graph. Our algorithms are shorter, simpler, and use a statement-level control dependence graph, as required for integration with the ParaScope programming environment.

Testing on several hundred scientific Fortran procedures confirms that while GSA form is dozens of times larger than the G_{CF} in a few cases, its growth is linear over the vast majority of procedures. The few procedures for which GSA form is much larger than SSA form have are either very unstructured or else have many loop exit branches.

Chapter 4

Symbolic Expressions

4.1 Introduction

Symbolic analysis requires the ability to represent and manipulate unknown values. We believe that the most important feature of each unknown value is the way that it is computed; for example, is it a completely mysterious value from outside this procedure, or is it computed from other unknown values and constants?

We need a representation for computations that is as complete as is possible at compile time, that makes identical values easy to spot, and that can easily be rewritten to exploit arithmetic properties and other higher-level knowledge.

In Fortran and related languages, values are built up by expression evaluation and loads and stores of variables. By combining expression trees with the global, GSA-form dataflow graph, we produce a *value graph* [AWZ88] for the procedure's computations that enables powerful symbolic pattern-matching and rewriting techniques.

Pattern Matching

The γ functions we build to represent merges are, like arithmetic operators, *compositional*: the meaning of a phrase such as $\gamma(P, v_1, ..., v_n)$ (or its equivalent graph) is dependent only on the operator (γ) and the meanings of the inputs $(P, v_1, ..., v_n)$. Another occurrence of the same operator with the same (or same-meaning) inputs therefore has also the same meaning.[1]

This gives us an extremely powerful tool for handling non-constant values. The compiler doesn't have to understand the meanings of expressions. It only needs to discover some primitive phrases with the same meanings, such as identical constants or

[1] One small caveat is necessary: a γ function giving the value of a variable at a program point, like any other expression, only has meaning *if* execution reaches that program point. We can ignore this distinction during construction and matching of value graph phrases, but enforce it to our profit when relating these values to particular program points.

uses of the same definition of the same variable. By combining dataflow information and expressions into one compositional value graph, we can apply graph pattern-matching to discover complex phrases with the same meaning.

Rewriting from Domain Knowledge

Pattern matching alone is not adequate to our needs, because many values with interesting relationships do not look exactly the same. Some can be rewritten to be the same, such as $(2 * x)$ and $(x + x)$. Others are fundamentally inequivalent but related, such as two auxiliary induction variables that can both be expressed as linear functions of a main loop induction variable.

Overview

The use of gated single-assignment form allows us to represent merges of different variable definitions with γ functions controlled by the control condition selecting the definition. When combined with expression trees, the resulting value graph models all the computations in the procedure. Conservative approximations are only for external values provided by other procedures or by system input. We maintain a many-to-one map from source-level program expressions and variable references to their representative value graph nodes. Section 4.2 gives the details of value graph construction.

Simply combining G_{DU}^{GSA} with expressions would produce a naïve value graph with a unique node for every expression. To detect distinct occurrences of equivalent expressions, we have to discover the matching subgraphs. Section 4.3 gives methods for pattern matching and assigning value numbers to sets of congruent nodes.

Pure value numbering applies a very narrow rule of exact isomorphism in discovering identities. To increase the power of our methods, we exploit arithmetic properties such as distributivity, associativity, and commutativity, simultaneous with value numbering, as discussed in Section 4.4.

Earlier researchers have employed value numbering to find identities, or graph rewriting to achieve more ambitious results. Section 4.7 explores the prior work in detail. The novelty of our work lies in combining value numbering and graph rewriting, expounding on the details, and giving experimental results.

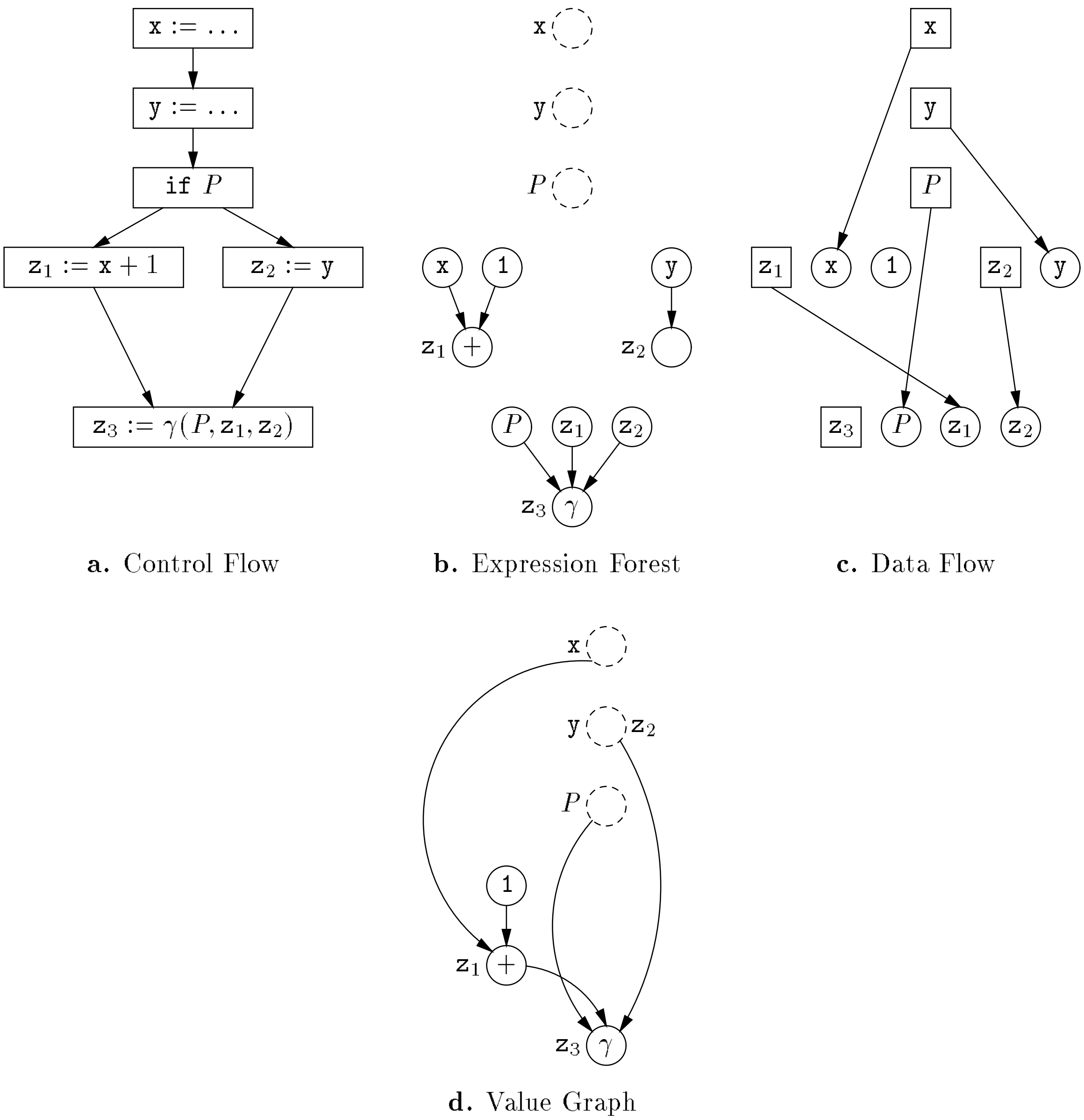


Figure 4.1 Building a Value Graph

4.2 Value Graphs

4.2.1 Formation

A value graph combines information from expression trees and def-use chains to represent the computations in a procedure. This much it has in common with other program representations, such as some varieties of program dependence graphs. Value graphs, in particular, are tuned to maximize the cases where computations with identical values have identical representations.

The expression forest of a procedure comprises the union of the expression trees from all its statements. Top-level expressions are found on the right-hand side of assignments and in the output list of `WRITE` statements. The forest is a graph $(U \cup S, E)$, with nodes for variable and constant uses (U) and for operators (S). Edges are in $(U \cup S) \times S$, going from uses and subexpression operators to operators of parent expressions. Some expressions represent the right-hand sides of variable assignments. For convenience of presentation, the root of each subexpression is assumed to be an operator (possibly the identity function).[2]

The other precursor of the value graph is the dataflow graph, linking the expression trees together with edges from definitions of variables to their uses. The dataflow graph $G_{DU}^{GSA} = (D \cup U, C)$ has definition nodes D representing reads and left-hand sides of assignments, use nodes U representing writes and right-hand sides of assignments, and edges C, also known as def-use chains. Each chain $(d, u) \in C$ connects a variable definition $d \in D$ with a reference $u \in U$ to the same variable.

Figure 4.1 shows a value graph built from the expression forest and the GSA-form dataflow graph by the following steps

1. Form the naïve value graph $V_0 = (D \cup U \cup S, C \cup E)$.

2. For each assignment statement, fuse the left-hand side (definition) node with the root node of the right-hand side expression.

3. For each def-use edge, fuse the source (definition) node with the sink (use) node and delete the edge. (This handling also applies to the special edges from conditional expressions in branches to uses as the predicates of γ functions.)

[2]Other researchers have expression edges going the opposite direction, from operators to uses [AWZ88], or have subexpressions pictured below their parent expressions [ASU86]. We adopt our conventions for ease of combination with dataflow graphs depicted in the traditional way.

4. For each copy node (operator node employing the identity function), combine the copy node with its input and delete the input edge.

5. If using SSA form, tag each ϕ function with the statement identifier for the merge point. If using GSA form, tag each μ function with the level of the loop header where it was added, and each η function with one less than the level of the loop exited.

6. Tag each external input (value passed on entry, value returned by call, or value returned by `READ`) with function identifier unique to that input and with the level of the deepest containing loop.

7. Keep a map from original definition and use nodes to their representatives in the value graph.

The resulting value graph V has combined all nodes that are equivalent due to copies, while remaining compositional. We have not yet combined repeated occurrences of the same subgraph generated from distinct original statements.

4.2.2 Congruence

Structural congruence is a form of graph isomorphism. Let the *slice* for a node be the subgraph of all nodes and edges reachable by tracing backwards through the flow graph. Two nodes are congruent if their slices are isomorphic — if they can be matched node to node, with corresponding nodes having the same operators, and edge to edge, with corresponding edges going from matched source nodes to the same-numbered inputs of matched sink nodes. To recognize congruence without actually extracting slices and comparing them, we employ the following, equivalent definition [AWZ88].

Definition: Two nodes are *congruent* if and only if:

- they are labeled by the same function,
- they have the same number of inputs (incoming edges), and
- corresponding input nodes (sources of incoming edges) are congruent.

Since nodes for constants have no inputs, they are congruent if they are labeled by the same constant function. Nodes for externally generated values (formal parameters, `READ` results, and unanalyzed side effects of procedure calls) have no inputs, but are never congruent because we label each with a unique function.

We maintain a map from program expressions to their representative value graph nodes. Program expressions with congruent representatives are said themselves to be congruent. Because congruent nodes cannot be distinguished, they may as well be combined in the representation. In the resulting *collapsed* value graph, congruent expressions are represented by the same value node.

Potential ordering problems complicate recognition of congruence for nodes with inputs. If the portion of value graph reachable from the nodes forms a directed acyclic graph, matching for congruence can be done in topological order, working from the leaves to the roots. However, if the graph has cycles, we need another method. Resolution of this problem is discussed in Section 4.3.

4.2.3 Equivalence

Recall that we maintain a map from program expressions to nodes in the value graph. If two expressions map to congruent value graph nodes, then by extension we call them congruent expressions. Here we discuss the conditions under which congruent expressions produce equivalent values.

In straight-line and branching code, each program expression executes once or not at all. Congruent expressions will produce the same value if they both execute. GSA form enforces this in merging code. In SSA form, we must conservatively assume every pair of ϕ functions to be different (to have distinct operators) unless they occurred at the same program point.

When loops are present, expressions may execute multiple times, producing different values each time. An expression is not necessarily even equivalent to itself across all loop iterations. We need to detect when values vary with loops, and which loops are involved.

Dags without Merges

For program fragments with straight-line code (basic blocks) or with branching code and no merges (extended basic blocks), value graphs are the same dags used in traditional value numbering [ASU86]. Expressions with congruent value nodes have the same value on the same execution of the basic block.

To demonstrate this, first consider congruent zero-height value dags (leaf nodes). In this case, there are no input edges, but the function labels are identical. For constants, the function labels are the values, so congruent constants are equivalent. Other leaf nodes indicate externally computed variables, such as parameters or globals passed from the calling procedure, side effects of called procedures, or variables `READ` from the input/output system. These are all given unique function labels. The only way that two expressions can get congruent value nodes for external values is if they derive from references or copies of the external value, which are equivalent.

Now assume that all congruent value dags of height $(n-1)$ or less correspond to equivalent run-time values. Do congruent value dags of height n correspond to equivalent run-time values? Their corresponding inputs are congruent, of height less than n, and therefore equivalent. Their function labels are the same. So long as the functions involved are compositional (producing values dependent only on their inputs), the run-time values will be equal.

Calls to non-compositional functions are treated the same as `READ` above. While these are treated as having inputs in the construction of GSA form, for value numbering we treat them as being leaf expressions with unique function labels, so they will not be found congruent to any value not copied from themselves.

Note that to produce these dags we may have combined nodes to eliminated def-use chains and identity functions. But so long as there is only one definition reaching each use (as is true with SSA and GSA forms), a use always has the same value as its reaching definition, so this is safe. Neither way of copying values introduces spurious congruences.

Dags with Merges

When there are merges in the code but no loops, then multiple definitions of a variable may reach a use in the original code. Conversion to SSA or GSA form adds new assignments so that, in the resulting value graph, ϕ (in SSA) or γ (in GSA) function nodes combine the multiple reaching values and select the appropriate one.

The ϕ function is not compositional: it depends on the context of the merge in the control-flow graph. Therefore, we mark each ϕ to uniquely indicate the control-flow merge where it was built. Congruent ϕ function nodes must not only have congruent inputs, but also derive from the same control-flow merge.

We therefore prefer GSA form, which gives us compositional γ functions for merged values. We can therefore identify congruent merged values computed at different points in the control-flow graph. GSA form helps us to reason about merged values independent of location. The γ functions encode the predicates selecting which input value is used at the merge point. Equivalent predicates selecting equivalent corresponding data inputs imply equivalent γ function results.

Cycles

When statements execute multiple times, we need to know which executions we are comparing before we assert that congruence implies equivalence. On GSA form, inclusion of loop level in the function labeling for μ and η is sufficient to guarantee that if two expressions are congruent, their run-time values are equivalent when both execute for the same iteration of the relevant surrounding loops.

The relevant loops are those with which the expression varies.

Definition: The *loop-variance* level of a node is

- for a μ operator, the nesting depth of the loop header where the GSA-form μ was originally built;
- for an η operator, one less than the depth of the outermost loop exited;
- for return values of calls and `READ`s in a loop, the nesting depth of the innermost loop containing the original statement;
- for constants and values on entry, zero; and
- for all other operators, the maximum level of all inputs.

Pattern matching need only explicitly consider levels on the first three categories of nodes, which will then induce the appropriate matches and mismatches on the other operator nodes.

Congruent expressions necessarily have the same variance level. If that level is ℓ, then the expressions assume the same value whenever the outermost ℓ loops surrounding each program expression have executed the same number of times — with the iteration counts for each corresponding loop matching exactly.

For example, if we are trying to evaluate a comparison test in a branch statement, we need not worry about the loop variance levels. All the expressions of concern are being compared for the same execution of the same statement.

If, however, we are involved in dependence testing, just saying whether expressions are equivalent within one iteration is not very interesting. We will need more aggressive arithmetic manipulation, as described in Section 4.4 to answer the right questions. Section 4.5.3 shows how to use level information in comparing values across iterations.

4.3 Value Numbering

The process of finding congruent expressions is frequently called value numbering. We assign a unique integer, or *value number* v to each set of congruent value graph nodes. Depending on the method used, we may combine congruent nodes in the value graph or leave them separate.

A precise value numbering method gives two nodes the same value number if and only if they are congruent. It is, however, still safe to assign different numbers to congruent nodes.

While the equality of run-time values is undecidable, value numbering is a conservative approximation that is both decidable and computable. Techniques for deciding congruence vary with their generality.

4.3.1 Partitioning

Because congruence is commutative and transitive, it constitutes an *equivalence relation* on value nodes. The set of all value nodes can therefore be partitioned completely into non-overlapping subsets (*equivalence classes*) of mutually congruent nodes.[3]

Almost-linear algorithms exist for finding this partition, starting with the assumption that all nodes are congruent (putting them all in one class) and progressively splitting congruence classes [AWZ88]. Termination occurs when no nodes assumed in the same class can be proven not congruent.

---

[3]Just because a relation is *an* equivalence relation doesn't mean that it is *the* equivalence relation (i.e., equality of value). However, congruence does happen to be connected to equality. To avoid further confusion, we will write no further on the general notion of equivalence relations, but only on congruence in particular.

By starting with an optimistic assumption, the partitioning method is able to solve the congruence problem exactly. This is particularly important for value nodes involved in cycles, which are very difficult to prove congruent if they are first assumed not to be.

However, this strength is outweighed, for our purposes, by a related weakness. The partitioning process never adds nodes to a class, but only divides them. In order to recognize two nodes congruent, it must have the congruence asserted and then fail to disprove it. Therefore, partitioning must either

- execute only the full value graph, or
- execute on incomplete value graphs first, then on the full graph.

The first option rules out most interesting kinds of graph rewriting, such as ordering commutative arguments according to their value numbers. The second could be prohibitively expensive, because partitioning incomplete graphs doesn't make partitioning the full one any easier.

While we cannot rule out tricks to improve value numbering by partitioning, we chose to use a pattern-matching method more compatible with graph rewriting.

4.3.2 Hashing

One trivial method of building value numbers is to traverse the value graph in topological order, keeping a list of all patterns found. When a particular combination of operator and input value numbers is not found in the list, it is added. Whether we have newly added it or found an old copy, we return the position in the list as the value number [ASU86].

Value hashing is exactly this method, optimized by substituting a hash table for the list. So long as we have no cycles in our value graph, it will discover exactly the same congruences as the partitioning method. It is inexpensive, as the expected cost of looking up a pattern is the size of the pattern being hashed plus a constant.

A particular benefit of value hashing is that it can be made demand-driven. In our implementation, the expression forest and the GSA-form dataflow graph constitute the naïve value graph, which we value number by recursively requesting value numbers for the inputs, then looking up the operator and input value numbers in the table.

Like all bottom-up pattern-matching schemes, hashing is essentially pessimistic, assuming nodes are not congruent until proven otherwise. It therefore cannot prove congruence of cyclic expressions. We address this problem in two ways:

- When the original code is reducible, a μ operator node must occur at the entry to every cycle in the value graph. We can conservatively number cycles by giving a distinct value number to every μ node.
- We can use more powerful techniques than pattern-matching to analyze iterative values.

Note in particular that the most interesting related values in loops are not exactly equivalent. More common are auxiliary induction variables and other values which are a function of the loop iteration count and loop-invariant values. The linear auxiliary induction variables, such as `j` in the following, are particularly amenable to symbolic manipulation.

```
do i = 1, n
    j = j + k
    A[j] = ...
enddo
```

The value of `j` at the beginning of each iteration, before being incremented, can be written as (j_0 `+ (i-1)*k`). No pure pattern-matching technique can recognize this relation between `i` and `j`, or between them and other inductive values.

Thus, while hashing is limited in its ability to handle cycles, even partitioning, a precise pattern-matching technique, cannot derive many important properties of loop-variant values. We therefore feel justified in value numbering by hashing, which gives us more freedom to rearrange expressions.

4.4 Rewriting

Properties of particular operators may allow us to rewrite the value graph. This can be especially useful when using value hashing, as inputs to commutative and associative operators can be sorted based on their value numbers *as the value graph is built.*

Rewriting floating-point expressions is controversial with scientific programmers, because

- the machine compiled on may have different floating-point precision than the target machine, and
- even if the compile-time precision is the same as the run-time precision, the magnitude of roundoff error may change with the order of evaluation.

Some programmers will want simplification of floating-point expressions, but it should not be the default. We will rewrite those floating-point expressions whose inputs and result are all integer-valued constants.

4.4.1 Constant Folding

When the arguments to an arithmetic operator or built-in function are integer or logical constants, we can frequently evaluate the expression. If we know the size of the target machine's integers, we can even spot overflow.

For example, the following operations can be folded when constant; we also give simple rules for detecting overflow. In evaluating constant floating-point expressions, we should check for overflow using integers with the same length as the mantissa in the floating-point representation.

Addition overflows if the inputs have the same sign but the result sign is different (or, in unsigned arithmetic, if the result is smaller than one or both of the inputs).

Subtraction is converted to addition.

Integer division can alway be applied (with truncation) to integer inputs, but can only be applied to integer-valued floats if the remainder is 0.

Multiplication of j and k overflows if and only if $\lfloor \log_2(j) \rfloor + \lfloor \log_2(k) \rfloor$ is equal to or greater than the number of bits available.

Logical and comparative operators never overflow (although we must be careful to avoid intermediate overflows in comparing very negative and very positive numbers).

4.4.2 Shallow Normalization

Some normalizations can be applied while only looking to a constant depth. By performing these on the fly (as each expression is added to the value graph), we can guarantee that the inputs to every node are normalized.

Commutative operators ($+$, $*$, `and`, `or`) should have their inputs consistently ordered.

- If one input is constant, put it on the right side.
- If both inputs are symbolic, put the one with smaller value number on the right (this requires that value numbers and value graph construction proceed in tandem).

Division can sometimes be evaluated. Where c and k are constant, and k divides c,

- replace $(c * x)/k$ with $(c/k) * x$
- replace $(c * x) \bmod k$ with 0

Comparisons should be converted, using subtraction and the `not` operator, to one of a few forms:

- $e > 0$
- $e == 0$
- $e \geq 0$

γ **functions** can sometimes be normalized:

- $\gamma(P, v_1, v_1) \rightarrow v_1$
- $\gamma(\mathbf{true}, v_1, v_2) \rightarrow v_1$
- $\gamma(\mathbf{false}, v_1, v_2) \rightarrow v_2$
- $\gamma(\mathbf{not}(P), v_1, v_2) \rightarrow \gamma(P, v_2, v_1)$

Arithmetic constants should be pulled out of γ functions *in a consistent way* to expose them to simplification of next section. For constant c_1, c_2;

- $\gamma(P, v_1 + c_1, v_2 + c_2) \rightarrow \gamma(P, v_1, v_2 + (c_2 - c_1)) + c_1$
- $\gamma(P, v_1c_1, v_2c_2) \rightarrow g\gamma(P, v_1c_1/g, v_2c_2/g)$
  where $g == \mathrm{gcd}(c_1, c_2)/\mathrm{sign}(c_1)$

4.4.3 Arithmetic Simplification

Integer arithmetic operations are relatively common and are so often used in combination in such a way that aggressive rewriting is both possible and useful. We perform arithmetic normalization on the fly while building the value graph and value numbering. Multiplication is distributed fully over addition, and we sort factors within

function *score*(e)
  if (e is a constant) **then**
    return 0
  else if (e has the form $c * t$) **then**
    return *score*(t)
  else
    return (*level*(e), *valnum*(e))

Algorithm 4.1 Scoring an Expression for Sorting

a product and terms within a sum. In the following, we use letters to designate the various forms:

- s — sum of two or more terms
- t — term: leaf value or expression with non-additive operator
  - ▹ c — constant
  - ▹ p — product of two or more sub-terms (not sums)
  - ▹ x — external input or non-arithmetic operation

All products and sums are maintained as sorted lists of factors and terms, respectively. The score used in sorting is very important, because it makes sure that constants come out at the top of expressions so they can be combined, and terms identical to each other (except for constant coefficients) come out next to each other and can be cancelled. The method for computing the score is shown in Algorithm 4.1.

Both products and sums are sorted into left-leaning trees, with lowest-scored items nearest the top. As the sort progresses, items with the same score are combined. This gives the constant coefficient for products, and gives a constant term and cancels adjacent terms for sums.

Breaking down every new arithmetic expression and sorting from scratch could be prohibitively expensive. However, most operations can be handled with minor adjustments of the sorted input subexpressions, avoiding a complete sort. Letting $|s|$ denote the number of terms in a sum s, $\langle s \rangle$ the number of factors in an average term in s, and $|p|$ the number of terms in a product p, the complexity of building a sorted expression given various forms of operators and subexpressions is as follows.

- $c \; op \; c$
  $O(1)$: Combine to a single constant (constant folding).

- $c * x, c + t, t + t$
  $O(1)$: Put in order at top level (shallow normalization).

- $c * p, c + s$
  $O(1)$: Combine new constant with any existing constant coefficient or term (found at top level) and leave new constant on the right side at the top level.

- $c * s$
  $O(|s|)$: Distribute over the sum and combine new constant with old constant coefficient of each term. Renumber, but do not re-sort.

- $x * p, x + s, p + s$
  $O(|p|), O(|s|)$: Search through sorted list and insert in proper place. Renumber list while unwinding, but do not re-sort.

- $p_1 * p_2, s_1 + s_2$
  $O(|p_1| + |p_2|), O(|s_1| + |s_2|)$: Merge sorted lists and renumber; do not re-sort.

- $x * s$
  $O(|s| * \langle s \rangle + |s| \log |s|)$: Distribute over the sum. Insert x in each term of sum, renumber each term, sort the sum, renumber the sum.

- $p * s$
  $O(|s| * (|p| + \langle s \rangle) + |s| \log |s|)$: Distribute over the sum. Merge product into and renumber each term, sort and renumber the sum.

- $s_1 * s_2$
  $O(|s_1| * |s_2| * (\langle s_1 \rangle + \langle s_2 \rangle) + |s_1| * |s_2| \log(|s_1| * |s_2|))$: Multiply out the two sums. Merge and renumber terms, sort and renumber the new sum.

4.4.4 Recurrence Recognition

Our original plan was to use value numbering to recognize and simplify recurrences (iterative computations). First, mark each μ at the beginning of a recurrence with a placeholder value $\mathcal{M}$. Then value number normally. If the iterative input to that μ

```
function number(GsaNode, Level)
    if (GsaNode.Value has been defined) then
        return GsaNode.Value;
    ... handle other types ...
    else if (GsaNode.Type == μ) then
        if (GsaNode.Level < Level) then
            return new_variant(GsaNode);
        else if (μ ∈ MuStack) then if (μ == top(MuStack)) then
                                            return 𝓜; /* placeholder */
                                        else return new_variant(GsaNode);
        else push μ onto MuStack;
            init := number(GsaNode.Init.Source);
            iter := number(GsaNode.Iter.Source);
            pop μ from MuStack;
            recursively unnumber and renumber GsaNode.Iter.Source;
            GsaNode.Value := new_hashed(init, iter, GsaNode.Level);
            return GsaNode.Value;
```

Algorithm 4.2 Value numbering a μ

ends up being, for example, $(\mathcal{M} + 1)$, then we have an auxiliary induction variable with step 1.

This trick, with a few more details, seems to work fine for single loops. We are forced to repeat value numbering for those values depending on the μ after it has been replaced with an inductive expression.

However, renumbering each iterative input can become burdensome with multiply nested loops. Assume that there is an induction variable used by all the loops, but modified (incremented by 1) only in the innermost loop. We would end up making multiple passes of value numbering and renumbering as we recognize the induction variable at each level, the total number of passes being quadratic in the nesting depth.

Our implementation uses Algorithm 4.2 to recognize auxiliary induction variables varying at one loop level. For nested loops, we should use analytical techniques for identifying induction variables [Wol92].

Induction Variables

An auxiliary induction variable's value will be an invariant iterative function of the placeholder; e.g., $(\mathcal{M} + 1)$. When this function is an invariant increment, this can be normalized to a linear expression of the loop iteration count. Similar conversions are even possible with non-additive increment functions such ($\mathcal{M} * 2$), which is the initial value times 2 to the power of the iteration count.

Other Reductions

We can have a variant yet analyzable iterative function: for example, $\mathcal{M} + A[i]$. If i is the loop index variable and A is not modified in the loop, then we are computing a sum over a section of A.

4.5 Comparing Program Expressions

4.5.1 Constant Differences

We can compare values v and w in straightforward fashion by building $(v - w)$ and simplifying. A constant difference that is positive, zero, or negative tells us that v is greater than, equal to, or less then w, respectively. Building and simplifying the difference will take linear time at worst.

In practice, this test can be done in constant time, without building the difference. Because the constant terms in sums are always kept at the top of the left-leaning tree, we can compare arbitrarily long sums by just checking first, to see if the two non-constant parts have the same value number, and if they do, comparing the constant terms.

4.5.2 Exploiting Bounds

If the difference $(v - w)$ is not constant, then our options are limited. We can either give up (which we must do in some cases regardless), or we can construct the full difference and try to use bounds information, described in Chapter 5, to determine if it is positive or negative.

4.5.3 Dependence Testing

When testing for dependence at level ℓ, we can handle all the index values varying with level ℓ or less and all the non-index values which vary less than ℓ (call the latter *loop-invariant* with respect to ℓ). Any values varying with deeper loops will have to be analyzed on the basis of range information alone.

Consider the following loop:

```
do i = 1, n
    ...
    A[c1*i + k1] := ...
    ... := A[c2*i + k2]
enddo
```

If we are testing for any possible dependence between the definition and the use of A, then we must determine whether or not

$$\texttt{c1} * x_1 + \texttt{k1} == \texttt{c2} * x_2 + \texttt{k2}$$

for any x_1 and $x_2 \in [1 : \texttt{n}]$. If there is a reuse, then the dependence distance $(x_2 - x_1)$ is the number of loop iterations between definition of a location in A and its use (if the distance is negative, then the use came first). The distance is constant when $\texttt{c1} == \texttt{c2}$, and is then given by

$$x_2 - x_1 == (\texttt{k1} - \texttt{k2})/\texttt{c1}$$

Since iterations are numbered by integers, there is no dependence if the distance is fractional (if $\texttt{c1} == 2$ and $\texttt{k1} - \texttt{k2}$ is odd, this corresponds to one of the references stepping through the even-indexed elements, the other through the odd). There is also no reuse if the magnitude of the distance is greater than or equal to n — the sets of accessed elements do not overlap (and there are no iterations that far apart).

If k1 and k2 are not constant, we need symbolic analysis techniques to tell us first, if they are loop invariant, and second, the value of their difference.

To organize the process of symbolic dependence testing, we parse the two subscripts being compared according to loop-variance level, producing two vectors for each subscript:

- *coeffs*$[i]$, holding the constant coefficient of the level-i index variable, or for $i == 0$, any constant added part, and
- *variant*$[i]$, holding terms of the sum which are variant with level i but not linear in the index variable; or for $i == 0$, the loop-invariant symbolic terms.

In testing for dependence for the example loop, *coeffs* $== [\texttt{k1}, \texttt{c1}]$ for the definition of A, while for the use of A, *coeffs* $== [\texttt{k2}, \texttt{c2}]$. Testing for dependence carried at level ℓ then proceeds as follows:

- If *variant*$[i] \neq 0$, for $i \geq \ell$, then give up (assume dependence).

- If remaining terms of *variant* are all equal (between the two subscripts being tested), we have reduced the problem to non-symbolic dependence testing using *coeffs*.

- Otherwise, attempt to compute a dependence distance ignoring *variant*, then add in all terms of *variant* and compare against the bounds.

4.5.4 Slicing

Both the naïve value graph (obtained by unioning G_{DU}^{GSA} with the expression forest) and the collapsed value graph (obtained by collapsing copies and congruent nodes in the naïve value graph) are useful for slicing [DSvH93]. Given a program expression, chasing backwards from its value node will give everything that affects its value. While this will omit the decision whether or not to execute the expression, that omission may be useful in recognizing identical slices that can be combined.

If the slice must be converted to actual code, the naïve value graph is somewhat easier to deal with. There is still a close correspondence between values and the intermediate variables they were stored in. When using the collapsed value graph for slicing, one must replicate the entire slice back to the atomic values: constants and external values (transmitted through formal parameters and globals on entry, actual parameters and globals on return from call, and variables in `READ` statements).

4.6 Evaluation of Performance

4.6.1 Expected Complexity

Our manipulation of symbolic expressions has three major sources of expense:

- construction of the expression forest and dataflow graph,

- hashing of values during value graph construction, and

- rewriting of values during value graph construction.

The expression forest is a subgraph of the abstract syntax tree, and therefore trivially linear in the size of the program (represented by the AST). As shown in Chapter 3, construction of the GSA-form dataflow graph appears linear for all but a few outlying procedures.

It takes time proportional to the size of the keys to evaluate the hash function and, assuming a uniform and sufficiently pseudo-random hash distribution, constant time to perform the lookup in the hash table [Knu73b]. In numbering the whole value graph (assuming that we do no rewriting), we hash the operator name and input value numbers of each node in topological order, adding those which are not yet present in the table. The size of the keys is thus proportional to the in-edges, and the total number of items in the hash table is the number of nodes minus the number of identities found. Assuming effective hashing, value numbering without rewriting takes linear time.

Shallow normalization adds at most a constant amount of time to the processing of each operator. Arithmetic normalization, on the other hand, can take a linear sequence of assignments and turn them into an exponential value tree. For example, consider the fragment

$$\begin{aligned} &x_0 := 1 \\ &x_1 := x_0 * (y_1 + z_1) \\ &\ldots \\ &x_n := x_{n-1} * (y_n + z_n) \end{aligned}$$

During arithmetic simplification, each x_i will have a value graph fragment twice as large as that for x_{i-1} because nothing cancels out. Distribution and sorting will also greatly reduce sharing between the subgraphs representing each of the x_i values. The value subgraph contributing to x_n will have 2^n terms, each of length n.

We predict, without proof, that such chains of multiplying sums will be short in practice. They are not a factor at all in the current implementation, which takes the short-cut of not distributing multiplications involving sums (except for multiplication by a constant). Under multiplication by a non-constant term, sums are scored as atomic terms.

If the arithmetic expressions that we are free to manipulate have size bounded by a constant, then the simplification of all value numbers takes linear time. If large expressions occur often enough to slow the whole system, we can limit simplification to expression dags of limited height.

4.6.2 Experimental Results

Efficiency

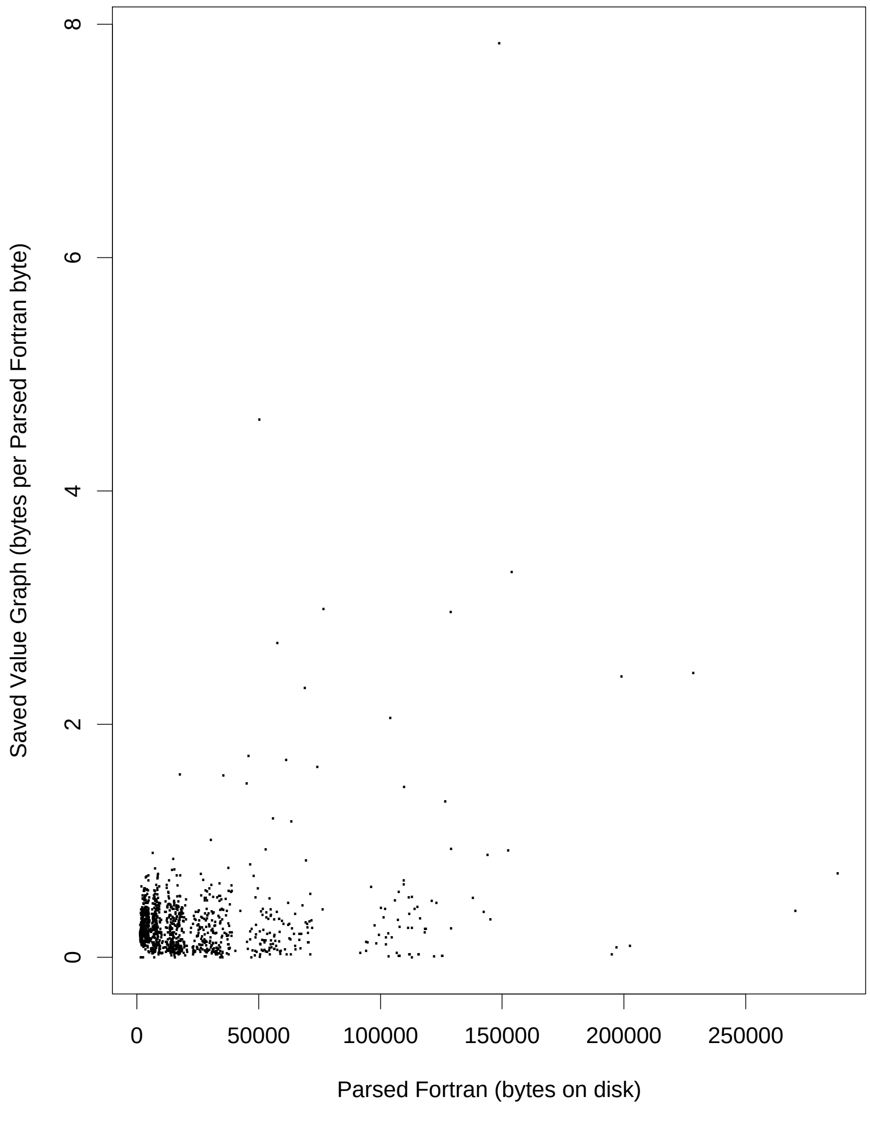


Figure 4.2 Saved Value Graph bytes per AST byte

Since the value graph is built directly from the expression forest and the SSA or GSA dataflow graph, it is generally about the same size.

Figure 4.2 charts the size of the (uncompressed) value graph built using GSA form and saved as initial information for interprocedural analysis, in bytes per byte of the saved AST. This initial symbolic information constitutes a slice of the value graph, giving every node that contributes to an interprocedurally visible value. Because the value graph file is stored in ASCII and the AST in binary, the comparison makes the value graph look larger by a constant factor. Still, except for a few outliers, the ratio of value graph size to AST size is bounded by a constant, so the value graph is roughly linear in the size of the procedure.

Effectiveness

Symbolic analysis should primarily be judged by how much it improves the precision of subsequent analyses and the results presented to the programmer. Final results can include reduced execution time of the compiled program or improved debugging abilities.

We experimented primarily with the use of symbolic expressions in dependence testing. Previous research shows that symbolic information can greatly improve dependence testing.

We ran ParaScope's dependence analysis on all the procedures in our benchmarks, then eliminated 71 procedures from consideration because of bugs in the analysis. We counted only dependences resulting from pairs of subscripted array references in the same procedure. Table 4.1 gives the cumulative counts of intraprocedural array dependences for each benchmark program under different levels of symbolic analysis. The column "Constants" gives counts when the value graph is used for constant propagation and constant folding only. The next two columns, under "Symbolic", give the absolute and percentage reduction in the edge count when symbolic pattern matching and rewriting are enabled. The final two columns, under "Arrays", give the further reductions (beyond those with scalar symbolic analysis) when pattern matching on array operations is enabled. For these dependence tests and these benchmarks, use of GSA form vs. SSA form to build the value graph made no difference.

While the overall reduction in dependence edges is small (just over three percent), for some programs the reduction is over 35 percent. These results are in the same range as those for the symbolic dependence analysis in PFC [GKT91]. The sym-

		Constants	Symbolics		Arrays	
		total deps	Δ	Δ%	Δ	Δ%
NAS:	`bt`	9890	873	8.8		
	`cg`	116	3	2.6		
	`ep`	6				
	`ft`	256	72	28.1		
	`is`	15	4	26.7		
	`lu`	2955	63	2.1		
	`mg`	75	3	4.0		
	`sp`	3437	445	12.9		
Perfect:	`adm`	1406	447	31.8		
	`arc2d`	1260	285	22.6		
	`bdna`	4076	121	3.0	51	1.3
	`dyfesm`	1262	177	14.0		
	`flo52`	527	60	11.4		
	`mdg`	2982	3	0.1		
	`mg3d`	3063	7	0.2		
	`ocean`	1358	41	3.0		
	`qcd`	3573	44	1.2		
	`spec77`	3373	834	24.7		
	`track`	547	6	1.1		
	`trfd`	233				
RiCEPS:	`boast`	46379	33	0.1		
	`ccm`	31181	498	1.6		
	`hydro`	919				
	`simple`	642	22	3.4		
	`sphot`	422				
	`wanal1`	544	58	10.7		
	`wave`	4063	603	14.8		
SPEC:	`doduc`	428	4	0.9		
	`fpppp`	8505	49	0.6		
	`matrix300`	1				
	`nasa7`	933	32	3.4		
	`spice`	19921	51	0.3	1	0.0
	`tomcatv`	43	14	32.3		
Total:		154391	4852	3.1	52	0.0

Table 4.1 ParaScope Data Dependence Edges

bolic analysis implemented in ParaScope is more extensive than that in PFC, except for the limitations of the auxiliary induction variable recognition in Section 4.4.4. Unfortunately, ParaScope's dependence testing is still incomplete [GKT91], limiting any comparison of the underlying symbolic analysis. In particular, ParaScope

- combines all distance and direction vectors between two references (forward and backward) into one vector, and
- fails to test for dependence carried by particular levels (i.e., freezing outer loops to test for inner-loop-carried dependences).

4.7 Related Work

Basic Blocks

Value numbering on basic blocks has a long history [AC72, ASU86]. It was generally employed in common subexpression elimination; that is, detecting multiple evaluations of the same operation and replacing them with one evaluation. The saved result can then be used in place of the other occurrences.

SSA Form

Global common subexpression elimination (over whole procedures) was one motivation in the development of static single-assignment form by Wegman, Zadeck, and others [AWZ88, RWZ88]. Their work inspired our use of a value graph to represent symbolic expressions. However, the method used for detecting congruence requires the whole value graph to be built before any congruences are detected. This severely limits rewriting of the graph, as it would have to be reanalyzed after rewriting to see if new congruences were exposed. A few tricks are shown for handling commutative operators [AWZ88], but they are unlikely to generalize. While this method should prove powerful in detecting exactly equivalent expressions, it is inadequate to the more general comparisons needed for dependence testing and other symbolic analysis clients.

PFC

The Parallel Fortran Converter (PFC) takes Fortran 77 and converts it to vector, parallel, or parallel/vector form [AK84, AK87]. As it evolved from a single-procedure vectorizer to a whole-program parallelization system, several symbolic analysis systems, all using symbolic expressions, were implemented.

The original symbolic analysis method relies on forward substitution and simplification of expression trees [All83]. Whenever there is a single definition of an integer variable reaching a use, the right-hand-side of the expression is substituted in place of the use — provided that none of the variables in the replacement expression are redefined between the two points. The resulting expression is then simplified.

This method has proven a very powerful tool in symbolic dependence analysis. Forward substitution pushes expression information into subscripts, where it is conveniently available to the dependence tester. Some 12 percent of all subscript pairs proven independent and 25 percent of all subscript pairs whose dependence relations are precisely computed are enabled by symbolic analysis [GKT91].

Despite its success, this method has not been copied exactly for use in interprocedural analysis within PFC or in the ParaScope system. The main reasons stem from the use of AST expressions for representing symbolic values:

- The AST is the internal representation of a single procedure, and would require significant adaptation for use in interprocedural analysis.
- Since the AST is the canonical representation of the procedure, this method implements symbolic analysis as a transformation (analysis methods without side effects are preferred).
- AST expressions are cumbersome; an expression representation tuned for symbolic analysis can be more lightweight.

When interprocedural constant propagation was added to PFC, symbolic expressions were also required to represent not-yet-constant values (that might become constant during the propagation phase) [CCKT86]. The data structures used resemble a value graph, but are built in an ad-hoc manner whose computational complexity is hard to control. Because these symbolic expressions are mainly used for representing but not for comparing values, they were not designed to maximize congruence of subtrees.

We implemented yet a third symbolic expression framework when we added array section analysis to PFC (see Chapter 2). Because the precision of array section translation and union depends heavily on the ability to compare bounds, we found the expression framework used for constant propagation to be inadequate. Because of the interprocedural nature of the analysis, we could not use AST expressions. Therefore, we implemented a value numbering method much like the one presented here, with the following differences:

- It uses simple def-use chains [Ken81] instead of GSA form. If multiple definitions, with different value numbers, reach a use, then approximate bounds on the value are built.
- The handling of potential interference from interprocedural MOD is more primitive. A list of the potential modifications affecting each saved value is kept, and if any are not disproven during interprocedural analysis, the value is marked as unknown.
- Rewriting is much more limited.
- Auxiliary induction variables are not handled.

These limitations forced us to rewrite the BLAS in a simpler but equivalent form before being able to accurately summarize their effects. While this is a valid approach for testing the power of regular sections in describing array side effects, better symbolic analysis is needed for analyzing realistic programs.

Dehbonei and Jouvelot

Dehbonei and Jouvelot describe a partial symbolic evaluation method based on symbolic expressions with guards [DJ92]. Each value is represented by a list of guards (logical expressions) followed by a list of symbolic expressions of the same length. The guards must be mutually exclusive; if the i^{th} guard would evaluate to `true` at run-time, then the run-time value is given by the i^{th} symbolic expression.

Where values are merged, the *shuffle* operator combines the two lists. Because *shuffle* can be expensive, they relax their method to leave *shuffle* unevaluated. They also use a value graph to represent symbolic expressions; because, like SSA form, it descends from the work of Reif *et al.* [RT81], it probably bears some resemblance to our value graph.

Unevaluated *shuffle* operators are closely related to the dags of γ operators built in full GSA form. They both combine some of the conditions under which an expression executes with symbolic representations of the possible values. With *shuffle*, these predicates and values are organized in lists; with γ, they are organized in a dag. Both can be used in pattern-matching. Which method is better awaits further investigation.

Thinned GSA form omits predicates affecting whether or not a program point is reached, leaving only the predicates selecting among possible reaching values. This potentially increases the congruences to be found over Dehbonei and Jouvelot's method.

Haghighat

Haghighat's symbolic analysis also relies on symbolic expressions [Hag90, HP90, HP93]. He has highly developed methods for rewriting expressions and solving recurrences, while our methods seem stronger in representing and matching patterns of operators which are poorly understood (except for being known compositional). In particular, our methods for representing and matching merged values with γ functions do not seem to have equivalents in his work.

Overall, we seem to have taken distinct but complementary approaches to the problem of symbolic expressions, and a fusion of the two methods should be profitable.

4.8 Summary

Neither value numbering nor arithmetic simplification are new; our innovations lie in combining them aggressively and using GSA form. Value numbering on GSA form lets us compare expressions from different basic blocks, as is often necessary in dependence testing. Careful structuring of simplification enables many symbolic comparisons to be performed with a very few instructions. The integrated analysis proves a potent tool in dependence testing.

Chapter 5

Symbolic Predicates

5.1 Introduction

In designing an efficient method for symbolic analysis, we have attempted to split the problem into smaller, more manageable parts. Chapter 4 treats the analysis of how values are computed; now we give the analysis of facts that hold when values are used.

In principle, we could jettison our analysis of symbolic expressions and deal in the more general world of symbolic predicates. These can include linear inequalities, bounds, and even equality relations representing the same facts as our expressions. However, our specific techniques for manipulating symbolic expressions can be more efficient than the general techniques for predicates. Furthermore, the representation of many symbolic predicates includes expressions in symbolic quantities. Therefore, we keep our symbolic expressions as part of the necessary infrastructure of a symbolic predicate method.

We include this discussion of general predicate propagation largely for completeness. It shows the synergies and incompatibilities between our symbolic expressions and predicate frameworks. We have yet to implement these techniques, except for the pairwise linear equalities which prove useful in the interprocedural analysis of Chapter 6.

5.1.1 Predicates *vs.* Expressions

When building symbolic expressions (and assigning value numbers) in Chapter 4, we focused on the operations used to compute each value. The congruence of two expressions implies that the same operations, applied to the same inputs, produce the same value for both, *if both execute.*

Despite, or because of, its efficiency, symbolic expression analysis has a few major limitations:

- While our symbolic expressions provide an exact representation for values, approximations are difficult to represent. A predicate can constrain a value, e.g. by specifying upper and lower bounds, without specifying it completely.

- Symbolic expressions are naturally built from the GSA-form def-use chains — a sparse dataflow graph, connecting definitions of each variable to its uses. GSA form and expressions are less useful in reasoning about relations among variables not introduced by definitions, such as inequality relations arising from use of a comparison in a conditional branch.

- Each definition (and use) of a variable is modeled by a single symbolic expression. More general frameworks are needed for sets of separate and simultaneous facts.

- Our symbolic expressions encode arithmetic and control flow determining the value at a definition, but omit control conditions affecting whether a definition or use executes. This makes sense, because we wish to compare expressions in different contexts by their value numbers. Computing the conditions under which a program point executes can independently improve our results.

Our division of the problem makes the last distinction particularly important; the value graph concisely represents the way values are computed, but ignores the contexts in which they are used. Predicates can combine this context information. For example, in the following fragment, the element modified is always in the lower triangle of the matrix:

```
if (i > j) then
   A[i,j] := ...
else
   A[j,i] := ...
endif
```

Tracking the inequality relation between `i` and `j` could be very useful, independent of any preceding knowledge of their values.

5.1.2 Sources of Predicates

A predicate can be any fact about a variable or a symbolic expression, or about a set of either, provided that the fact can be meaningfully communicated to a compiler. Some sources of predicates are given below.

Control conditions

When a statement's execution is guarded, either explicitly or implicitly, by the value of a comparison or boolean expression, we can assume that the test value is true.

- *conditional branches*

  These are explicit transfers of control that affect whether or not the statement is reached.

  ```
        IF (m .gt. 5) THEN
  S1        PRINT *, m
        ENDIF
  ```

  In this case, the predicate ($\mathtt{m} > 5$) holds at S1.

- *assertions*

  Some languages enforce termination if an asserted condition is false. An assertion is the same as a conditional branch to the program exit, but because the branch is only taken in abnormal termination, we may have more freedom to rearrange code around it.

  ```
        ASSERT(n .lt. 100)
  S2    DO i = 1, n
            CALL work(i)
        ENDDO
  ```

  Here, the predicate $\mathtt{n} < 100$ holds at S2 and beyond.

- *error conditions*

  These are implicit assertions, which may or may not be checked in practice, but which must hold for safe program execution.

  ```
        DIMENSION X(20)
        READ *, m
  S3    X(m) = 0
  ```

  Because the array subscript must be within the arrays bounds, the predicate $0 < \mathtt{m} \leq 20$ holds at S3 and beyond.[1]

[1] Array bounds of 1 in the last dimension of a Fortran formal parameter should be ignored, as this is an anachronistic way of representing unknown bounds.

Implications of expressions

While the symbolic expression for a value may be precise for pattern matching, it can be obscure for other purposes. Some facts are best deduced from the expressions, then propagated separately.

```
x = sin(...)
y = x + 1
```

In this case, $-1 \leq \mathtt{x} \leq 1$ and $0 \leq \mathtt{y} \leq 2$.

Merges of predicates

Propagation must start with predicates derived from the sources listed above. However, summarizing the multiple control-flow paths that can reach a merge point can require great care in the selection of a predicate framework.

- *local merges*
  The most important cases to exploit are distinct reaching definitions.

```
    IF (P) THEN
        j = 1
        k = 4
    ELSE
        j = 2
        k = 0
S4  ENDIF
```

- *call sites*
  When there are multiple calls to a procedure, constraints for all context must be merged (alternatively, the procedure can be cloned [BCHT90] and the cloned copies optimized for the different contexts).

```
SUBROUTINE bar()
    ...
    CALL foo(1, 4)
    ...
    CALL foo(2, 0)
    ...
```

```
SUBROUTINE foo(j, k)
    ...
```

In both cases above, different constants are in the variables at S4, or in the parameters for foo, depending on the path taken. The different constants in each variable can be summarized imprecisely by the constant bounds ($\mathtt{j} \in [1:2:2]$) and ($\mathtt{k} \in [0:4:4]$).[2] Each reaching pair of $(\mathtt{j}, \mathtt{k})$ values represents a point, which can be summarized at the merge by the line $(4\mathtt{j} + \mathtt{k} == 8)$.

It is generally impossible to retain precise information after a merge of predicate. We must balance retaining information against keeping a simple representation.

5.1.3 Outline of Chapter

Having given some idea of what we mean by symbolic predicates, we now sketch the rest of the chapter. Section 5.2 presents some particular varieties of symbolic predicates. The efficiency and style of analysis is affected by how many quantities are constrained by each predicate and whether the quantities modeled are variable contents or contextless values. Section 5.3 discusses how different program representations from Chapter 3 can be used for predicate propagation. Section 5.4 describes the framework of pairwise linear equalities exploited in the interprocedural analysis of Chapter 6. Section 5.5 describes how to combine symbolic predicate and expression information in dependence testing and other analysis and optimization tasks. Section 5.6 describes the extensive prior work on symbolic predicates, and Section 5.7 gives our conclusions.

5.2 Structure of Predicates

The symbolic representation of Chapter 4 produces, for each expression, either a unique name (for external inputs and other unknowable values) or a complete graph structure, whose leaves are constants or unique names, and which can be compared or combined with the value graphs for other expressions. Symbolic predicates encompass a more general and potentially vague set of facts:

- *linear equalities* — give equality relations on values without saying how any one of them is computed

[2]The triplet expression $[l : u : s]$ here, as in Chapter 2, represents the set of variables $\{i | (l \leq i \leq s)$ **and** $((i - l) \bmod s == 0)\}$.

- *linear inequalities* — give bounds on how an unknown value may vary, potentially in terms of other values
- *ranges* — for a given value, not just upper and lower bounds but stride information
- *variance* information – how a value, such as that of a loop-variant variable or of a subscripted array reference, changes with successive loop iterations or subscripts
  - ▷ is it monotonically increasing or decreasing?
  - ▷ does it ever repeat a value?

In principle, we could jettison our analysis of symbolic expressions and deal only with the more general world of predicates. In practice, symbolic expressions are a way of handling a special case cheaply and efficiently. Efficient analysis of equality predicates is generally limited to linear expressions, whereas the pattern-matching techniques of Chapter 4 apply to any complete expression — that is, any expression free of hidden, unanalyzed references or side effects.

5.2.1 Predicates on Variables or on Values?

Previous research has, with few exceptions [DJ92], generally relied either on symbolic expressions or symbolic predicates, but not both. Predicates have therefore usually been defined on the current contents of variables [CH78, Kar76, IJT91]. This complicates analysis in two major ways.

Assignments remove predicates. When a variable is assigned a new value (unrelated to its old one), then it must be removed from the predicate set for that point. If the old predicates relate the modified variable to others, the predicate sets updated can be large.

Some values are never stored. If a value is never saved to a variable, then it cannot be reasoned about directly with predicates. When a branch condition involves a comparison of linear expressions in variables, e.g., $(2j > n)$, it can be interpreted as a predicate on the variables in many frameworks. However, frameworks for handling nonlinear expressions in predicates, e.g., $(j^2 > n)$, are not as precise or efficient. Generally, nonlinear relations are ignored in a variable-based framework.[3]

Partly for these reasons, but primarily for the greatest compatibility with our symbolic expression analysis described in Chapter 4, predicates should be maintained on the values of symbolic expressions (as identified by value numbers), not variables. This provides a common basis for the deductions of expression rewriting and predicate propagation, with a relatively clean division of labor:

- Extended value numbering (pattern-matching with rewrites) identifies expression values that are computed from the same inputs with the same operators.
- Predicate propagation derives facts about these values that apply in particular contexts.

Note that the meanings of single-assignment values are not tied to context. Since they are never redefined, we never need to discard a predicate due to the side effects of some statement. Depending on the exact framework used, this can allow shortcuts to reduce the amount of propagation.

5.2.2 Self Predicates *vs.* Relational Predicates

The number of separate values constrained by a predicate is a major feature in evaluating a predicate framework. We define two major cases:

Definition: *Self predicates* constrain the value of one non-constant expression which is manipulated atomically (that is, without regard to its internal structure) by the predicate framework.

Definition: *Relational predicates* constrain multiple expressions which are manipulated independently by the predicate framework.

The same fact can be manipulated as either kind of predicate; it depends on the framework. For example, the linear equality $a + b == 2$ can be a self predicate on the value $(a + b)$ or a relational predicate on the values a and b.

[3] We also ignore them in our value-based framework, but can still manipulate nonlinear expressions as unanalyzed atomic values in linear expressions. The difference is that other frameworks only allow this treatment for values currently residing in variables.

5.2.3 Combining Predicates

There are two basic operations to be performed on sets of predicates:[4]

Composition ($\wedge$): when we discover that multiple predicates apply simultaneously, we can assume that all are true. The choice of name is influenced by the successive application of predicates as we descend past conditional branches.

Merge ($\vee$): when we discover that different sets of predicates apply on different executions of the same code, we can only assume a predicate true on every execution if it exists in all the original sets. The choice of name results from the use of this operation when propagating sets to a merge (join) point in the control flow graph.

In many predicate frameworks, it is easier to construct a precise composition than a precise merge, or even any merge at all. If each predicate set restricts some values to a convex region, then composition intersects the regions, producing a convex result. Merging unions the regions, producing a potentially non-convex result. Consider the case of constant bounds:

- $[1:5] \wedge [3:7] \Rightarrow [3:5]$
  $[1:3] \wedge [5:7] \Rightarrow \emptyset$ (i.e., contradiction)
- $[1:5] \vee [3:7] \Rightarrow [1:7]$
  $[1:3] \vee [5:7] \Rightarrow 1,2,3,5,6,7 \subseteq [1:7]$

Staying in the framework of a convex set may require an imprecise merge operation. Furthermore, merging sets can produce empty information, while composing sets can more frequently salvage something: if two sets cannot be precisely composed, we can at least pick the most informative input set as the result.

Because merge operators may be imprecise, complicated, and expensive, we consider below the effects of ignoring some or all merges.

5.3 Predicate Propagation Graphs

Chapter 3 gives us several program forms which can be useful in predicate propagation. The simplest is the control-flow graph, for which we can define a dataflow

[4]Composition is conjunction of predicates and intersection of the spaces of allowed values, while merge is disjunction of predicates and union of value spaces.

analysis framework to check the effect of every assignment and every branch on the sets of predicates. However, this may waste a great deal of effort examining statements which do not affect the predicates, or which affect predicates involving only a few variables. We show how control-dependence and dataflow graphs can provide comparable results with better expected performance.

5.3.1 Control Flow Graph

Traditional dataflow analysis methods can be used to propagate arbitrary sets of information over the control flow graph. We need only define the predicates to be propagated and appropriate functions for the effects of control-flow components [ASU86]. A *dataflow analysis framework* consists of

- A set S of values to be propagated. For us, this is the set of subsets of predicates from an appropriate universe (such as linear inequalities).
- A set of *transfer functions* from S to S, and a method of deriving a transfer function for each.
- The merge operation $\vee$, to be applied at joins in the control flow graph.

Using the control flow graph allows a more exhaustive propagation of predicates than the control dependence graph. Both merges of control conditions and merges of values can be handled.

- Merges of control conditions happen only in unstructured code, where a node may have multiple incoming control dependences.
- Merges of values happen any time there is a merge in the control flow graph, bring multiple paths together where there may be a definition after the predominator of the merge.

Predicate analysis can profit from a more detailed G_{CF}, in which all assertions and potential error points are treated as conditional aborts. The resulting CFG obscures much program structure, but will expose more control conditions. The same trick can be used to build a more detailed control dependence graph (which will look less structured because of the jumps out of nesting).

Traditional iterative methods for dataflow analysis on the control-flow are easily extended to the propagation of linear predicate vectors. Two obvious problems are the propagation of empty predicate information and that termination is not guaranteed for many predicate frameworks.

In order to guarantee termination for predicate propagation in cycles of the value graph, the lattice of predicate information should have finite depth. Each merge operation of distinct predicate sets (neither containing the other) loses information, yielding a smaller predicate set lower in the lattice. When a lattice has the finite descending chain property, a finite number of merge operations on distinct predicate sets is guaranteed to produce the empty set.

However, one can simulate the efficiency of a finite-depth lattice. Since loop-variant values (μ operators) are the reason why we may continually propagate, we need a special merge operator, called a *widening* operator, that is guaranteed to converge in a finite number of steps [CC77]. Different widening operators can be defined for each variety of predicates. One option, used on subscripts of array sections in Chapter 2, is to attach a counter to the information and take it directly to $\perp$ (no information) if the count exceeds some predefined constant.

5.3.2 Control Dependences

Control dependences, as defined in Chapter 3, encode the tests which decide whether or not statements are executed. Whereas in the control-flow graph, paths correspond to sequences of statements, in G_{CD}, paths generally correspond to increasingly restricted execution contexts. Straight-line sequences of statements become siblings in G_{CD}.

These properties make the control dependence graph particularly suited for analyzing predicates based on context. A statement's set of control dependence ancestors includes only those statements which restrict its execution. Furthermore, in block-structured code (where all control-flow paths from a branch first meet at its post-dominator), the forward control dependence graph is a tree.

Each path through the control dependence graph from START to a statement S contributes one possible control context for the execution of S. The control conditions for the branches on a path encountered are composed to give the set of predicates contributed by that path. If F_{CD} is a tree, we are done; otherwise we must approximate the MOP (meet over all paths) solution by computing predicate sets reaching each node, and merging sets from predecessors of a control flow merge.

The merge operation for many predicate frameworks is more expensive and less precise than the composition operation. Depending on the ultimate effect on the derived information, we may want to handle as many merges as possible (directing us towards the control flow graph), or no merges at all (suggesting the use of a pruned control dependence graph) or something in between. To eliminate all the merges in a control dependence graph, prune out all incoming edges to a node m with multiple incoming edges, and add a new, unlabeled in-edge from m's immediate pre-dominator in G_{CF} to m, representing that m's execution conditions are contained in those of its pre-dominator. (The resulting pruned G_{CD} will be invalid for other uses.)

Computing predicate sets on the pruned control dependence graph will give us the same answer for block-structured code (for which pruning is a null operation) and more conservative answers for unstructured code.

5.3.3 Mixing Control and Dataflow Predicates

We must be careful how we mix predicates derived in different ways. Let us make the following distinction:

Control predicates are those derived from conditional branches; e.g., from the control dependence graph.

Dataflow predicates are those derived from the form of expressions; e.g., from the value graph.

The control-flow graph can be used to propagate either or both, via traditional iterative or interval propagation methods.

If we keep these two kinds of predicates separate until we must handle a symbolic query, we can make special use of them.

- Pure control predicates involve only context information, sets of facts that hold if a statement executes. When performing dependence analysis, two statements are examined for reuse of the same memory. Because dependence holds only if both statements execute, we can compose the two sets of control predicates for use in dependence testing.

- Pure dataflow predicates involve only information present in the value graph. Therefore, they can be kept attached to the value graph and applied to multiple expressions mapping to the same value graph node.

The payoff is conceivably quite high if the same variable occurs in two subscript expressions being tested for dependence, and the context of one is much more restricted than the other. On the other hand, if the two are mixed, we require distinct predicate sets for every program point *and* we must merge predicates, losing precision, before testing for dependence.

5.3.4 Efficiency

The time and space requirements of predicate propagation are dependent on the predicate universe, the representation of the predicates, and the propagation method chosen.

Length of Lattice Chains

First, consider the depth of the predicate lattice. Self-predicates may be considered independently for each variable. Constant, variance, and stride information all have lattices with the finite descending chain property. However, constant bounds have an unbounded lattice. In order to guarantee convergence in iterative propagation, we must introduce a widening operator, applied when propagating around loops, that takes the lattice values to $\perp$ in a bounded number of operations [CH78]. The method proposed array section subscripts in Chapter 2 can be applied here; simply limit the number of times a lattice value saved at a loop header can be lowered; if the limit is exceeded, replace the lattice value with $\perp$.

Relational predicates involve multiple variables, which can multiply the lattice depth over that for a similar self-predicate framework.

Graph Propagation

Propagation of self predicates on the control-flow graph will require at least $O(N_{CF}V)$ time and space, because the dense predicate information of length $O(V)$ is stored at each of the N_{CF} control-flow graph nodes. (Here V is either the number of variables or the number of symbolic values, depending on how we decided to build our predicates.) However, if we propagate on a dataflow graph (or value graph) only, then we keep one predicate for each node (since each node represents only one value). If we wish to combine control-based predicates, we require a set of predicates for each distinct control dependence context. In the worst, case, this propagation is $O(N_{CF}V)$ like that on the control-flow graph. However, propagation of predicates on the control dependence graph is more readily optimized to the particular situation.

5.4 Pairwise Linear Equalities

Because of the efficiency and power of our value numbering techniques and the computational expense of published methods for handling relational predicates, we have omitted the latter from our *intra*procedural analysis. However, for reasons discussed in Chapter 6, it is unwise to propagate too much information about symbolic expressions across call boundaries. One simple reason is that different symbolic expressions may be passed for a parameter at different call sites, and we have no interprocedural equivalent of γ to represent the result of this merge.

In order to carry relations among variables across the interprocedural gap, we have developed methods to manipulate pairwise linear equality predicates. These are sets of predicates of the form

$$c_1x_1 + c_2x_2 == c_0$$

where the c_i are constant. Considering the x_i values as indexing two coordinate axes, such predicates limit the pairs of values to a line in the two-dimensional plane.

Note that we do define these predicate on variables, not values. This is natural, because the set of variables which can carry information across the interprocedural boundary is well defined. For each call site, we build all pairwise equalities that we can derive from intraprocedural information, and save them in a particularly compact form.

5.4.1 Construction

Pairwise linear equalities are easily built from intraprocedural value numbers. A key insight is that the relation "x and y are related by a pairwise linear equality" is transitive. Given linear equalities between x and y and between y and z, we can always derive a linear equality between x and z.

Algorithm 5.1 shows how to build a set of linear equalities for the variables visible to a call site. First of all, we only wish to bother with those variables whose values are used, so we look at the REF set for the called procedure (if USE, that is, used-before-modified, information is available, that is even better).

$Cache := \emptyset$ // initialize cache to empty
foreach $x \in$ REF, in canonical order **do**
    if (x is constant) **then** save constant value
else
    extract constant coefficient and constant added term from value for x
        $x == C_x b + K_x$
    if ($\exists p$ with $(b, p) \in Cache$) **then**
        // obtain expression $p == C_p b + K_p$
        // $C_x p - C_p x == C_x K_p - C_p K_x$ (cancel out b)
        $g := \gcd(abs(C_x), abs(C_p))$
        if ($C_x < 0$) **then** $g := -g$ // normalize parent coeff > 0
        // save relation $(c_p p + c_x x == k_x)$
        $c_p := C_x / g$
        $c_x := C_p / g$
        $k_x := (C_x K_p - C_p K_x)/g$
    else // make x the parent of subsequent vars with same base
        $Cache := Cache \cup (b, x)$

Algorithm 5.1 Converting Symbolic Expressions to Pairwise Equalities

We require a canonical order for the variables to be analyzed that is consistent across all our manipulation algorithms and all call sites to the procedure. This allows us to unambiguously make the first variable that is encountered with each linear base value be the parent (representative) of all other such variables.

Definition: A *basic term* is an expression which is not a known constant, and which does not contain multiplication by or addition of a constant at the top level. The *linear base* of an expression x is a basic term b such that $(x == cb + k)$, where c, k are constants. (Note that the linear base of b is itself.)

As we examine each referenced variable x, we extract the linear base of its value at the current call site. When the linear base b is not already cached, we add (b, x) to *Cache*. When we encounter a variable y whose base is already associated with another variable x, we use the symbolic expressions for x and y, in terms of b, to derive a direct relationship. We add this relation to the predicate set, with x as the parent variable representing the class containing x and y.

The resulting predicate set can be represented by a vector giving the parent (if any) of each variable and the three constants (c_p, c_x, k_x) relating the variable x to its parent p. The time required to build these relations is proportional to the number of referenced variables. If intraprocedural predicate propagation has been performed separately, and neither analysis subsumes the other, then we can compose the two sets of pairwise linear equalities and save that set for future use.

5.4.2 Composition

Composition of pairwise linear equalities is highly efficient. However, since we do not propagate these relations within procedures, we have no use for it in our implementation.

The relation "x and y have a pairwise linear equality" is an equivalence relation: it is reflexive, symmetric and transitive. Therefore, it partitions the set of variables into disjoint classes, with all variables in a class linearly related to the others.

To compose two sets of relations R and S, we proceed as follows:

- Examine the relation between each variable x and its parent p (if any) in R.
- If x and p are also related in S, then either the relation is the same, or else we have a contradiction. The latter can happen, for example, when code is not executable.
- If x and p are not related in S, then add the new relation to S.

The modified version of S produced represents $(R \wedge S)$. Addition of a relation to S can union two previously unrelated classes of variables; this takes almost linear time using fast UNION-FIND data structures [Tar83]. Composition can therefore be done in time almost linear in the number of variables involved.

If $C(R)$ is the number of classes in the set of pairwise linear equalities R, then $C(R \wedge S) \leq min(C(R), C(S))$. Either the number of classes is the same, and the result is the same as one of the inputs, or some of the classes has been merged. The minimum number of classes is 1 and the maximum the number of variables. The predicate sets and $\wedge$ thus form a semi-lattice in which the longest chains have length proportional to the number of variables. Propagation of these constraints is guaranteed to converge even on a graph with cycles.

```
function merge(Q, R) // merge two sets of pairwise equalities
    if (size(Q) < size(R)) then swap(Q, R)
    S := R      // make a deep copy
    foreach p ∈ REF, in canonical order do
        if (parent_S(p) == nil) then
            // p is the parent (representative) of a set of related variables in S
            q := nil
            foreach child y of p do
                if same relation (c_py p + c_y y == k_y) applies in both S and Q then
                    leave relation in result set S
                else if (q == nil) then
                    parent_S(y) := q      // make y independent of p
                    q := y
                else
                    parent_S(y) := q      // make y a child of q, independent of p
                    cancel p in old relations (c_py p + c_y y == k_y) and (c_pq p + c_q q == k_q)
                        to compute new relation (c_qy q + c'_y == k'_y)
```

Algorithm 5.2 Merging Sets of Pairwise Equalities

5.4.3 Merge

Our current algorithm for merge is, unfortunately, quadratic in the number of variables related. The problem occurs when we start out with large classes of related variables, and gradually lose information. Algorithm 5.2 gives the method.

First, we compare the two sets of predicates and make a copy of the smaller. The size of a set of predicates we take to be the number of relations between variables that it specifies; therefore, this is proportional to the sum of the squares of the sizes of its classes (recall that each class is a set of variables with the same parent, which are all mutually related by pairwise linear equalities).

We then examine the variables in the same canonical order used above. This ordering will always encounter the parent (i.e., representative variable) p of each class before any of its children. For each child x that maintains the same linear equality relation to its parent in the both Q and S, set, we keep that relation in S. Otherwise, we delete the relation. If another child q of p has already been so treated, we make x a child of q and derive the new relation from the old. Otherwise, we make x its own representative.

The running time of the algorithm is quadratic in the size of the largest class in the smaller set of predicates. The worst case occurs if we repeatedly process a class, each time extracting only its parent and selecting a new parent for the rest. Each pass over a class of original size n then reduces its size by 1, so that the total time required by the passes is $O(n^2)$. In the event that large classes of related variables are being reduced by merges at call sites, we should clone the called procedures to exploit the relationships and avoid the merge.

We omit the treatment of constants, which complicate the algorithm somewhat. The set of constant variables is treated as a class in each predicate set. If two variables are constant in both Q and S, and the constants are the same, then they are left constant in S. If the constants are different in Q and S, we can build a linear relation between the variables.

Having constants for two variables in one of Q and S while they are non-constant but related in the other allows us to keep the non-constant relation. Constants in one of Q or S which are inconsistent with a linear relation in the other force us to delete the relation.

For $C(R)$ the number of classes in the set of pairwise linear equalities R, $C(R \vee S) \geq max(C(R), C(S))$. Either the number of classes is the same and the result is the same as one of the inputs, or at least one of the classes has been split. The minimum number of classes is 1 and the maximum the number of variables. The predicate sets and $\wedge$ thus form a semi-lattice in which the longest chains therefore have length proportional to the number of variables. Propagation of these constraints is guaranteed to converge even on a graph with cycles.

5.4.4 Combination with Constant Ranges

Pairwise linear equalities combine naturally with constant ranges. First, let us consider an extended formulation of ranges that separates the information into bounds and stride information:

Definition: A *constant range* $R = [L : U : S@A]$ is the intersection of the sets of integer values given by each of two components:

- Upper and lower bounds: $\forall x \in R, L \leq x \leq U$

- Stride and alignment:[5] $x \bmod S = A$;
  i.e., $\forall x \in R, x = A + \alpha S$ for some integer α.

in which L, U, S, A are integer constants.

The beauty of the combination is that a constant range predicate on one variable x can be used to derive a valid range for any variable linearly related to x. Consider again the general form of a pairwise linear equality, with the c_i representing constants:

$$c_1x_1 + c_2x_2 == c_0$$

Given $x_1 \in R_1 == [L_1 : U_1 : S_1@A_1]$, we can derive the range $R_2 = [L_2 : U_2 : S_2@A_2]$ containing all possible values of x_2.

- L_2

  $$\begin{aligned} &== \lceil (c_0 - c_1U_1)/c_2 \rceil, \text{ if } (c_1c_2 > 0) \\ &== \lceil (c_0 - c_1L_1)/c_2 \rceil, \text{ if } (c_1c_2 < 0) \end{aligned}$$

- U_2

  $$\begin{aligned} &== \lfloor (c_0 - c_1L_1)/c_2 \rfloor, \text{ if } (c_1c_2 > 0) \\ &== \lfloor (c_0 - c_1U_1)/c_2 \rfloor, \text{ if } (c_1c_2 < 0) \end{aligned}$$

- $S_2 == (c_1S_1)/c_2$

- $A_2 == ((c_0 - c_1A_1)/c_2) \bmod S_2$

Note that there pairwise equalities imply a stride and alignment for each of the related values. So long as a variable is related to another, separate stride and alignment information is superfluous. When merges remove all relations of a variable to others, the separate stride and alignment information is no longer redundant.

[5]Using the Euclidean definition of mod and div, for which $x == S \cdot (x \text{ div } S) + (x \bmod S)$, $0 \leq (x \bmod S) < |x|$ [Bou92].

foreach $x \in$ REF, in canonical order **do**
if ($x == c$, a constant) **then** *do nothing*
else if ($parent(x) ==$ **nil**) **then** // x is a root variable
$C_x := 1; \quad K_x := 0$
else // we have ($c_p p + c_x x == k_x$) for root variable $p := parent(x)$
use Euclidean GCD method to find $g == \gcd(c_p, c_x)$ and
p_0, x_0 such that ($c_p p_0 + c_x x_0 == g$); g divides k_x
$p_1 := p_0 k_x / g; \qquad x_1 := x_0 k_x / g \quad // \; c_p p_0 + c_x x == k_x$
// Invent base variable i generating values satisfying original relation
$C_x := -c_p / g; \qquad K_x := x_1 \qquad // \; x == C_x i + K_x$
// Combine ($p == (c_x/g) i + p_1$) with prior relation ($p == C_p j + K_p$)
use Euclidean GCD method to find $g' == \gcd(c_x/g, C_p)$ and
i_0, j_0 such that ($(c_x/g) i_0 - C_p j_0 == g$); g' divides ($p_1 - K_p$)
$i_1 := i_0 (p_1 - K_p) / g'$
$p_3 := (c_x/g) i_1 + p_1 \quad$ // substitute for i yielding particular p
// New relation: ($p = \mathrm{lcm}((c_x/g), C_p) j' + p_3$), where j' varies freely
$C_p := \mathrm{lcm}(c_x/g, C_p); \quad K_p := p_3 \qquad // \; p == C_p j' + K_p$
foreach $x \in V$ in canonical order **do**
if ($x == c$, a constant) **then** $entry_x := c$
else if ($parent(x) ==$ **nil**) **then** // x is a root variable
generate primitive symbolic value $base_x$
$entry_x := C_x base_x + K_x$
else // we have ($c_p p + c_x x == k_x$) for root variable $p := parent(x)$
// substituting $entry_p$ for p: $\quad c_p(C_p base_p + K_p) + c_x x == k_x$
$C_x := -c_p C_p / c_x; \qquad K_x := (k_x - c_p K_p) / c_x$
$entry_x := C_x base_p + K_x$

Algorithm 5.3 Converting Pairwise Equalities to Symbolic Expressions

5.4.5 Usage

We use pairwise equality predicates to bride the gap between procedure bodies in interprocedural analysis. Depending on the specific case, it can be difficult or impossible to meaningfully translate a symbolic expression from the context of a call site to that of the called routine. Chapter 6 describes many reasons, but a simple one is that we lack a good way of the merge of values from different call sites as a symbolic expression.

Therefore, we build pairwise linear equalities at call sites and merge these predicate sets in the interprocedural analysis. When we want to examine values on entry, we generate expressions from the predicates that represent equivalent information.

Algorithm 5.3 gives the method for building symbolic expressions from pairwise linear equalities. Note that it would be trivial if in every pair (x, y) of variables, one (e.g., x) had unit coefficient in their relation. Then we could use y as the base, and define x by a linear expression in y. But because both can have non-unit coefficients, and integer division is awkward to manipulate, we must invent a synthetic base value b, presumed to range over all the integers (except as constrained by constant bounds on x and y), with each value generating corresponding values of x and y.

We first make a pass over all the variables, once again in the same canonical order used during construction of the predicate sets. As we encounter each root, we save coefficient 1 and addend 0, indicating that the root variable may range freely over the integers (so far as we know).

When we encounter a non-root variable x, we derive from the relationship with its root p a point (p_1, x_1) that satisfies the relation. (The Euclidean greatest-common-divisor algorithm is given in [Knu73a].) We then invent a value i, presumed to range over all the integers, and write p and x in terms of i so that each value of i corresponds to a (p, x) point satisfying their relation.

The expression $(p == (c_x/g)i + p_1)$ is equivalent to specifying a stride and alignment for p. We intersect this with the stride and alignment implied by p's linear relations with its other children. After processing all the children, therefore generating a properly limited expression for p, we process all the children again to generate their expressions in terms of the same base.

This algorithm takes time proportional to the number of variables.

5.5 Using Predicate Information

Our notion of the equivalence of scalar expressions is strongly influenced by the demands of our applications: dependence analysis, test elision, and array section manipulation.

Of particular importance in testing for dependences and in building summary array sections is the ability to compare expressions computed in different parts of the program. By maintaining predicates in terms of contextless symbolic expressions, we can treat congruent value numbers as equivalent in the expressions being compared. To combine the predicates for the expressions,

- if testing for dependence, compose the predicates, because both statements must execute for there to be a dependence; and
- if unioning sections for MOD or REF, merge the predicates, because we are describing the effect if either statement executes.

We say that two loop-invariant program expressions x and y are *equivalent* if and only if for any execution of x and any execution of y, the results of evaluating x and y are the same. Two loop-variant program expressions x and y, varying with loop level i, are *equivalent* if and only if for any execution of x_0 of x and any execution y_0 of y, either

- the results of evaluating x_0 and y_0 are the same, or
- the loop iteration count for at least one of the i outermost loops surrounding x is, for execution x_0, different from the iteration count of the corresponding loop surrounding y, for execution y_0.

In the latter case, we say that the expressions are equivalent but the particular executions are not. When comparing program expressions whose different contexts make imbedded values inequivalent, we must say that the result of the comparison is unknown.

5.5.1 Test Elision

For test elision and generation of predicates, one might be comparing a variable defined outside a controlled region with one defined inside. But at the point of comparison, both definitions must have executed (with the same iteration counts), so any equivalences are meaningful. For test elision, value numbers all by themselves are useful, and information strictly improves when predicate information is used to refine the value numbers.

5.5.2 Dependence Analysis

For dependence testing, we may be comparing subscript expressions in arbitrarily distant parts of the program. Of course, if either expression does not execute, then its associated array reference also fails to execute, so no dependence can exist. So it is valid to compare loop-invariant value numbers. In the presence of predicate information, the predicates for both references should be composed and applied to both value numbers (because both must execute for a dependence to exist) before comparing.

5.5.3 Simplifying Uses

It may be too expensive to examine every program expression for simplification of its inputs. But on occasion, a value computed under one control condition may be used under a copy of that condition.

```
if (P) then
    x := y
else
    x := z
endif
...
if (P) then
    ... := x
endif
```

In this case the use of `x` will have a symbolic expression $\gamma(\mathtt{P}, \mathrm{val}(\mathtt{y}), \mathrm{val}(\mathtt{z}))$, which simplifies to $\mathrm{val}(\mathtt{y})$ when we combine it with the controlling guard.

5.6 Related Work

5.6.1 Equality Predicates

Karr

Karr proposed a framework for analyzing linear equalities of arbitrarily many variables. The allowable values of the variables define an affine space, represented as a $V \times V$ matrix of simultaneous equations, where V is the number of variables. [Kar76].

Conjoining affine spaces corresponds to our composition operation. Conjunction involves, in the worst case, putting one $V \times V$ matrix under another and using row operations to convert to normal form. This operation takes $O(V^2)$ time.

Unioning affine spaces corresponds to our merge operation. It requires V operations on the matrices, and as each operation appears to be $O(V^2)$, the whole union operation seems to be cubic. The longest chain in the lattice induced by the merge operation — that is, the maximum number of times a stored matrix can be changed by a merge operation — is $O(V)$.

Our methods for handling pairwise linear equalities are a factor of V more efficient in the composition and merge operations, but have a lattice depth that is also $O(V)$.

Many of the benefits of linear equalities seem to be achievable using only the symbolic expressions of Chapter 4, but a full comparison would require an implementation of Karr's method.

5.6.2 Inequality Predicates

Symbolic Bounds

Harrison gives ad-hoc techniques for propagating symbolic bounds information, especially for loop induction variables [Har77]. While his methods have inspired much later work, including much of our Chapter 2, it is difficult to assess the exact complexity and precision of his techniques.

General Linear Inequalities

Cousot and Halbwachs give a general framework for handling linear inequalities. The propagate sets of predicates, each predicate of the form predicates of the form

$$c_1x_1 + c_2x_2 + \ldots + c_nx_n \leq c_0$$

or

$$c_1x_1 + c_2x_2 + \ldots + c_nx_n < c_0,$$

where the c_i are constants. Each set of such vectors limits the x_i values, taken as a set of points in n-space, to a convex polytope. Unfortunately, manipulation can be expensive.

- Composition is intersection, a linear programming problem and therefore exponential in the worst case.
- Merge is convex hull, both exponential in cost and imprecise in result.

The implementors of the PIPS system use a similar general framework, but specialize it to common cases and claim to achieve much greater efficiency in practice [IJT91, Iri93].

Simple Sections

The simple section framework of Balasundaram's Data Access Descriptors was developed to represent predicates on subscripts [BK89, Bal89]. It can also be used to represent predicates on other values (although we are unaware of prior work in extending it thus).

Simple sections represent predicates of the form $L \leq x \pm y \leq U$, *i.e.*, pairwise linear inequalities where the coefficients are constrained to be 1 or -1. The size of simple sections, and the time required for meet and intersection operations, are quadratic in the number of variables mentioned. The lattice depth is unbounded.

5.7 Summary

We consider the most common forms of symbolic predicate information to be too expensive for use in a compiler. However, in case they can be shown efficient in practice, we give methods to combine them with our symbolic expression techniques.

Constant bounds information should definitely be worth propagating. We show that pairwise linear equalities can also be an efficient method for carrying symbolic information across call boundaries. We investigate the experimental validation of the latter in Chapter 6.

Chapter 6

Whole-Program Propagation

6.1 Introduction

As programs become larger and hardware more idiosyncratic, analysis beyond procedure boundaries becomes increasingly vital to a compiler. This chapter outlines a framework for the solution of flow-sensitive program analysis problems, and shows how whole-program analysis of symbolic values can be achieved in this framework.

Generally speaking, it takes longer to perform analysis than to avoid it. The question must always be, is the gain worth the expense? Compiler analysis, when it pays off in optimization, can win in two ways:

- speeding up the compiled program, thereby winning the good will of people who care deeply about running their programs quickly, and
- speeding up the compiler (for example, by shortening the compiled program enough to offset the analysis time), thereby pleasing everyone, especially programmers.

In order to have a chance, interprocedural analyzers must not only be fast on the parts of the program they analyze, but they must check new results against old ones to determine which routines are not affected by changes and thus need not be recompiled. Without such a means of approximating separate compilation, handling of large programs would be impractical.

We show how a particular version of interprocedural symbolic analysis can be made efficient. While the jury is still out on the value of this analysis, the implementation techniques and evaluation methods are the same that must be applied if more powerful methods are to be successful.

6.1.1 Traditional Interprocedural Analyses

Many frameworks for computing interprocedural facts are described in the literature. The basic questions are simple, but solving them efficiently can be difficult. *Summary* analysis describes the side effects of a procedure to its caller; e.g., which variables does it read (or write)? *Context* analysis describes the context of a call site to the called procedure; e.g., what variables can share storage? Sometimes summary and context questions are intertwined in one problem, such as the propagation of constants returned from one call site and passed to another.

MOD and REF

The prime examples of summary information are descriptions of variables that are potentially accessed by a procedure. Given a code fragment F,

- MOD(F) is a set containing all variables that are modified by any execution of F.
- REF(F) is a set containing all variables referenced by any execution of F.

For F a call site, MOD (REF) includes the variables modified (referenced) by this call to the procedure. For F a procedure, the access sets accumulate the side effects of all call sites and other statements in the procedure body. Note that MOD and REF are conservatively large; they include all variables that may be accessed, but can include some that are not. For this reason, we call them *may-summary* problems [Ban79, Ban78].

MOD and REF are also called *flow-insensitive* problems because their precise solution does not depend on the structure of control flow [Ban78]. The side effect information for a procedure is built from the effects of its statements and calls to other procedures. When combining summaries for multiple statements (or call sites), a flow-insensitive summary is the union of the individual summaries, regardless of whether the statements occurred in sequence or on different branches.

However, even so-called flow-insensitive problems can benefit from the removal of dead code through test elision. Whenever the last read or write to a variable is proven unreachable, that variable is removed REF or MOD set, respectively, improving the information available to an optimizer.

KILL

The flip-side of MOD is the set of variables that are always modified:

- KILL(F) is a set containing only variables modified on every execution of F.

Note that KILL is conservatively small; it contains only variables that must be modified, but potentially omits variables. For this reason, we call it a *must-summary* problem. Like USE, described below, it is a *flow-sensitive* problem. The KILL summary for a sequence of statements is the union of the individual summaries, but that for statements on different branches is the intersection.

KILL analysis also benefits from elimination of unreachably dead code. This makes the KILL set smaller, not larger, by proving paths that avoid modification of a variable cannot be executed.

USE

USE is a flow-sensitive problem that isn't naturally described as a must-summary problem.

- USE(F) is a set containing all variables which may be referenced in an execution of F without first being redefined.

USE is essentially a flow-sensitive refinement of REF, and can replace REF for most purposes, such as determining if variable's value is needed by a procedure.

Removal of unreachably dead code can delete variables to the USE set, by deleting uses of a variable or deleting paths to a use which would have avoided a definition.

ALIAS

When two variable names refer to the same or overlapping storage, they are called aliases. In Fortran, static aliases are introduced by `EQUIVALENCE` statements, which are easy to analyze. Dynamic aliases are introduced through procedure calls, whereby the same variable becomes accessible by multiple names (all but one of which must be formal parameter variables).

- MAY-ALIAS(p) is a set containing all pairs of variables (x, y), where x and y share storage on some call to the procedure p.

Unlike the four other problems just described, MAY-ALIAS is a context analysis problem. It describes the storage map passed into each procedure from the outside.

6.1.2 Constant Propagation

Interprocedural constant propagation is a combination of summary and context analysis.

- RETURN-CONSTANTS(p) is a set of pairs (x, v) where x is a variable modified by procedure p and v is the known constant value produced in x on return from p.

- PASSED-CONSTANTS(p) is a set of pairs (x, v) where x is a variable referenced by procedure p and v is a known constant which is present in x at every executable call site for p.

Unlike the previous problems, effective solutions for constants require both forward and backward propagation on the call graph. Variables are frequently defined in initialization routines, then returned to a higher level before being passed back down to their uses. However, given a suitable expression framework, all the return values can be computed symbolically in one backwards pass, followed by a forward pass which computes the passed values.

6.1.3 Symbolic Interprocedural Analyses

In keeping with our division of symbolic information into expressions and predicates, and modeling our interprocedural propagation after that of constants, we divide symbolic interprocedural analysis into four parts.

- RETURN-EXPRESSIONS(p) is a set of pairs (x, v) where x is a variable modified by procedure p and v is a symbolic expression for the value in x on exit from p.

- PASSED-EXPRESSIONS(p) is a set of pairs (x, v) where x is a variable referenced by procedure p and v is a symbolic expression for the value(s) present in x at every executable call site for p.

- RETURN-PREDICATES(p) is a set of predicates, relating the values of the variables in (MOD(p) $\cup$ REF(p)) to each other, that holds on exit from p.

- PASSED-PREDICATES(p) is a set of predicates, relating the values of the variables in REF(p) to each other, that holds at every executable call site for p.

We consider PASSED-EXPRESSIONS a difficult problem to solve, because of the difficulty of detecting changes in the solution. Of the other three, we have implemented PASSED-PREDICATES (for pairwise linear equalities) and RETURN-EXPRESSIONS.

6.2 Interprocedural Strategy

Interprocedural analysis must be efficient and effective to gain acceptance. Our strategy for interprocedural analysis builds on prior work by many researchers at Rice working on the ParaScope programming environment[1]. The key features are:

- separation of analysis and transformation
  This philosophy is applied in analyzing single procedures and whole programs. Where possible, code is analyzed as written, without preliminary transformations. The products of analysis are then used to change and hopefully improve the code.

- distinct interprocedural phases
  The whole-program compilation process consists of disjoint phases of

  ▹ per-procedure gathering of initial information for interprocedural analysis and exploiting of interprocedural results

  ▹ propagation of information among all procedures

  Summary representations of procedures are used in the interprocedural propagation phase, so that multiple passes can be made without reading procedure bodies.

- recompilation analysis
  In traditional development environments, the only interprocedural tool is the linker/loader. To approach the time savings of separate compilation, we must find ways to recompile only those procedures whose old object files are invalidated by new interprocedural facts.

6.2.1 Separate Analysis and Transformation

Why have a special mechanism for analyzing multiple procedures? One can simply replace each call with the body of the called procedure, then analyze the whole program at once.

Few advocates of inlining as a substitute for interprocedural analysis would go quite that far. It is simply impossible to inline everything in a language allowing

[1] ParaScope's interprocedural analysis infrastructure is descended from the $\mathbb{R}^n$ programming environment, and forms the basis for the new D System at Rice.

recursion. And even in non-recursive Fortran, the main focus of this work, exhaustive inlining can lead to excessive growth in code size — an exponential amount of growth in the worst case [CHT91].

Inlining:

- cannot always be applied (potential array reshapes)
- can slow program execution (anomalies in register allocation)
- frequently slows down compile time (nonlinear algorithms)

These are not fatal criticisms of inlining as a program transformation. With sufficient analysis support, we can sometimes fix the obstacles to inlining and determine where inlining can enable enough optimization to make it profitable.

But to inline intelligently, we need interprocedural analysis. Analysis and transformation must be kept distinct.

6.2.2 Distinct Interprocedural Phases

Why bother with building a summary representation of each procedure, when one could just analyze the procedures in order? The problem is, which order, and how to compute it?

Visiting procedures in any interesting order requires construction of a call graph, in which nodes are procedures and directed edges represent the source procedure calling the destination. Building the call graph is itself an interprocedural problem, requiring information about call sites in every procedure.

Having built the call graph, we can compute the order in which to visit procedures for each interprocedural analysis problem. Analyses of side effects, such as MOD and REF, describe called procedures to their callers — and therefore require a reverse topological ordering of visits (in the absence of recursion, this would be a postorder walk of the call dag). Analyses of context, such as ALIAS and passed constants, describe the context of call sites to the called routine — and therefore require a topological ordering of visits (in the absence of recursion, a preorder walk).

Solving these four problems would require three separate traversals, because those for ALIAS and constants cannot be combined. Reading every procedure on every pass would take enormous amounts of time. Therefore we combine passes over the call graph into phases. Each interprocedural phases is preceded by an initial analysis phase, which attaches interesting information about a procedure to its call graph node, and is followed by an exploitation phase, which uses interprocedural information to detect further facts and enable transformations.

6.2.3 Recompilation Analysis

Compilers are a central program development tool, and their efficiency is a large determinant in the efficiency of the programming process. For programs of any significant size, the coding task is dominated by the time spent recompiling after a change, then testing — the proverbial "edit–compile–link–run" cycle. Progress would be near impossible without separate compilation, which permits the recompilation of only the modified procedures, leaving only a simple linker to stitch together the executable.

Programmers will not give up separate compilation. The difference, for example in recompiling the ParaScope environment, is one of minutes to recompile one procedure and relink, versus the bulk of a day to recompile everything. This obstacle would seem to prohibit interprocedural analysis.

The answer is recompilation analysis. We dare not promiscuously propagate information across procedure boundaries, but must instead define solutions which can be compared from one program compilation to the next. We then need only recompile a procedure when a interprocedural fact is used in compiling once and that fact changes on the next compilation.

This is essentially incremental compilation at the procedure level. We cannot in general expect this to be quite as cheap as separate compilation, but with careful design hope to keep the cost low enough that the optimization benefits justify its use.

We have not implemented recompilation analysis, but have been careful to make our methods ready for this extension. We show how to test a new interprocedural symbolic solution against an old one to see which procedures must be recompiled.

6.3 Annotated Call Graph

The call graph is the basic structure of interprocedural analysis. Initial information is attached to the call graph, then the interprocedural phase propagates to find solutions, which are also attached to the call graph for use in compiling each procedure.

6.3.1 Procedure Summaries

The annotations for many flow-insensitive problems are extremely simple. For MOD and REF, each procedure has a set of variables potentially written and another of variables potentially read, without attention to control flow.

For flow-sensitive problems, the context of each variable access matters as much as its existence. A variable must be written on every path through the procedure to be in the KILL set. If on any path it can be read before being written, the variable is in the USE set. These and other flow-sensitive problems rely on some control-flow information.

How much information should be attached to the call graph node for each procedure? In prior work, this has ranged from a relatively complete representation [Mye81] to one highly tailored for KILL and USE [Cal88]. Concurrently with other researchers at Rice [HMBCR93], we have decided to take a middle road of a lightweight control-flow graph for each procedure, with annotations to the control-flow nodes as needed for each particular problem.

The core procedure-summary control-flow graph (G_{CF}^{PS}) consists of five kinds of nodes:

- a unique `entry` node, with no in-edges, representing the start of the procedure after being invoked by some call site[2]
- a unique `exit` node, with no out-edges, serving a merge point for all procedure exits before control is returned to the caller
- for each call site, a `call` node, representing the state and execution context immediately before jumping to the called routine
- for each call site, a `return` node, representing the state and execution context immediately after returning from the called routine
- `plain` nodes, which may further be distinguished as
  - ▷ `branch` nodes, with multiple out-edges
  - ▷ `merge` nodes, with multiple in-edges

[2]In many language, including Fortran, procedures can have multiple entry points. We will either model these as plain nodes, branched to from `entry` depending on the entry name invoked, or else prohibit them.

a `plain` node can be both a `branch` and a `merge`

We maintain a mapping between each pair of `entry` and `exit` nodes and the corresponding call graph node, and between each pair of `call` and `return` nodes and the corresponding call graph edge, along with information about the variables through which values can cross the procedural interface. Otherwise, the G_{CF}^{PS} nodes are empty and the edges connecting them unlabeled except as additional information is provided through the annotation mechanism. This allows the basic framework to be lightweight and efficient, while supporting more complicated analysis as necessary.

6.3.2 Propagation

Interprocedural propagation proceeds by one or more distinct phases, each preceded by an initial phase gathering information from each procedure and followed by a post phase exploiting interprocedural solutions in local problems such as code generation. When there are multiple interprocedural phases, the post-phase for one and the pre-phase for the next can be combined, so that we talk of three-phase or five-phase analysis, typically.

The separation of phases avoids the problem of multiple accesses to the bulky representations used in analyzing and optimizing single procedures.

Directions

Within each interprocedural phases, we may have multiple passes, each typically, but not necessarily, propagating information in one of two directions:

- *Forward* passes push information from routines with call sites to the called routine. The facts propagated can generally be called *context* information, because they describe the outside world surrounding a call site.

- *Backward* passes pull information from called routines to each routine containing a call site. The facts propagated can generally be called *side-effect* information, because they describe changes in the program state due to the call.

Modes

In flow-sensitive analysis, we may focus a lot of attention on the G_{CF}^{PS} for particular procedure. It is then useful to distinguish the true interprocedural mode of analysis from the subprocedural mode.

To control recompilation, we must be able to measure the information crossing procedure boundaries and detect when a new solution is different from an old one. For this reason, subprocedural propagation will often look very different from the true interprocedural propagation.

6.4 Initial Information

6.4.1 Prerequisites

We need the G_{CF}^{PS} so that we can attach our annotations to particular nodes, and a mapping between `call/return` nodes and call graph edges. Having the results of interprocedural MOD analysis is very helpful, but its use can be postponed through the use of dummy predicates that check the yet-to-be-computed MOD information.

This analysis would be unacceptably crude were we to assume that all potential aliases did occur. Three ways that our analysis can handle aliases are:

- Assume they never exist (this is supported by the Fortran standard, but may be violated by common practice)
- Assume they do not exist in the initial analysis, then conservatively degrade the analysis based on aliases later discovered.
- Compute an approximation to alias information first.

Initial analysis can be much simpler and faster with interprocedural MOD and alias information than without. Using them in this initial phase puts us in the realm of five-phase analysis, with two interprocedural phases.

We think that our method of two interprocedural passes will be acceptable for several reasons. Primarily, we believe that flow-insensitive interprocedural analysis is trivial in analysis costs and stable in its solutions.

- The classical flow-insensitive problems of MOD, REF and ALIAS have extremely efficient solutions [CK88b, CK89].
- MOD, REF and ALIAS can be expected to change infrequently across program edits — adding the first and deleting the last references of a variable are rare decisions, and aliases are rare both in their occurrence and in their insertion and deletion.

- More modern languages than Fortran frequently include MOD and REF information as declarations, and encourage greater use of call-by-value parameters, which cannot be aliased. (These features are especially helpful for the scalar integer variables on which our analysis concentrates.)

Therefore, the initial phase for symbolic analysis, as described here, will rely on a previous pass of MOD, REF and ALIAS alias analysis. If such results are not available, we can still use this method with imprecise but safe assumptions (or unsafe assumptions, which are preferred in the case of ALIAS information); or we can use a more complicated method described later.

To limit recompilation, our strategy restricts interprocedural propagation to vectors of predicates on scalar values. No symbolic facts will be propagated between procedures except those representable as binary linear equality predicates or constant ranges for globals and formal parameters at ENTRY and EXIT, and for globals and actual parameters at CALL and RETURN. However, because we represent some control flow within procedures, we have the opportunity to do some subprocedural analysis in more detail.

6.4.2 Important Symbolic Expressions

We keep one separate value graph for each procedure, as an annotation on its call graph node. This graph is (ideally) pruned down from the full value graph to only that part needed to represent important symbolic expressions. It therefore represents a union of the slices for these expressions.

- passed values: the values in global variables, actual parameters, and static variables reaching each call site. Ideally, we save these only for the variables referenced via the call (in its IP REF set).

- returned values: the values in global variables, formal parameters, and static variables reaching the end of the procedure. Ideally, we save this information only for variables modified in this routine (in its IP MOD set).

- branch conditions: the values determining which out-edge from a each branch node will be taken. For this to be useful, each G_{CF}^{PS} edge must also be labeled with the branch condition value corresponding to its execution.

- expressions interesting to other analyses: These include array subscripts and bounds (for array section analysis), loop bounds (for performance estimation), and other facts for other uses that we can't necessarily predict.

6.4.3 Important Symbolic Predicates

With the current structure of this analysis, symbolic predicates are only used at procedure boundaries and are freely converted to and from symbolic expressions. These two-variable linear equalities are not used in subprocedural analysis, and do not need to be gathered in the initial analysis phase.

However, if more complicated predicate sets are used, they may require subprocedural propagation. They can be built from a full predicate propagation graph with exit branches for assertions and error conditions, then used to annotate a normal G_{CF}^{PS} to support test elision without excessive clutter.

At procedure boundaries, symbolic predicates must be in terms of variables, but within subprocedural propagation, they must be in terms of symbolic expressions (value numbers).

6.4.4 Related Annotations

These initial annotations for symbolic analysis are made to the same G_{CF}^{PS} that can be used for other IP analysis problems. By placing annotations for variable modifications and references on G_{CF}^{PS} nodes, improved KILL and USE analysis can be achieved when evaluation of tests enables code elimination. This can even benefit MOD and REF, although at the expense of making them dependent on symbolic analysis.

6.5 Propagating Symbolic Information

6.5.1 Backwards Pass

Algorithm 6.1 gives the method by which return values from called procedures are incorporated into the return-value descriptions of their callers. Note the usage of `transReturned` and `getPassed` to manage the interprocedural jump between rewriting expressions from the caller and rewriting ones from the callee.

```
foreach node (procedure) p in reverse topological order on call graph do
    load value graph OVals for p
    build new value graph NVals
    foreach interprocedural variable v ∈ MOD(p) do
        NVals.Map(v, End) := NVals.update(OVals, OVals.Map(v, End))

NVals.update(OVals, oval)
    if oval is a return value then
        (VarName, CgEdge) := oval.site_info
        nval := NVals.transReturned(OVals, VarName, CgEdge)
    else
        allocate nvalStruct in NVals
        foreach field f of oval with input value number v do
            nvalStruct.f := NVals.update(OVals, v)
        nvalStruct.op := oval.op
        nval := NVals.lookup(nvalStruct)
    return nval

NVals.transReturned(OVals, VarName, CgEdge)
    RVals := CgEdge.called.vals
    oval := RVals.Map(VarName, cgedge.called.exit)
    return NVals.translate(OVals, RVals, oval, CgEdge)

NVals.translate(OVals, RVals, oval, CgEdge)
    if oval is a passed value (used actual/global/static)
        VarName := oval.name
        nval := NVals.getPassed(OVals, VarName, CgEdge)
    else
        allocate nvalStruct in NVals
        foreach field f of oval with input value number v do
            nvalStruct.f := NVals.translate(OVals, RVals, v, CgEdge)
        nvalStruct.op := oval.op
        nval := NVals.lookup(nvalStruct)
    return nval

NVals.getPassed(OVals, VarName, CgEdge)
    oval := OVals.Map(VarName, CgEdge.site)
    return NVals.update(OVals, oval)
```

Algorithm 6.1 Translating a Return Value

6.5.2 Forwards Pass

The forward propagation pass uses symbolic expressions in propagating information subprocedurally and pairwise linear equalities to jump procedure boundaries. During subprocedural propagation, call sites are handled using the return-value descriptions saved in the backward pass.

When visiting a call-graph node, we build new value numbers for entry values to express their linear relationships.

We then visit `call` nodes of G_{CF}^{PS} in topological order (ignoring loop back-edges). We build the two-variable linear equality relations among all variables in the REF set of the called routine, and attach this predicate set to the call graph edge. We then proceed to the next call site.

When we need the return value from a call site (to build a value passed to a subsequent call site), we translate again the saved return symbolic expressions for the callee.

6.6 Exploiting Symbolic Information

6.6.1 Procedure Compilation

The main benefits of interprocedural symbolic information are expected to be in analysis of arrays and of conditional branches. For arrays, improved subscript analysis should give better dependence information, both by traditional methods and indirectly by improving array section information needed for privatization [TP93]. For conditional branches, the tests may sometimes be evaluated symbolically, allowing the deletion of unexecuted code. While the direct speedups from reduced code size and saving the test would be small, indirect benefits can be large due to reduced uncertainty. This is especially true in library code, where different combinations of values enable very different functionality.

6.6.2 Cloning and Inlining

Apply goal-directed methods of [BCHT90]. First, a backwards propagation to determine which sets of values are important. An initial approximation is the REF set; a better one is the set of variables used in the slices for array subscripts, conditional branches and array and loop bounds. Then a forward propagation of the symbolic information.

6.6.3 Recompilation Analysis

Tracking the use of symbolic information within a procedure would be complicated, and probably more effort than it's worth. Whatever information is exposed to the procedure compiler, we should assume that a change in that information compels recompilation of the procedure. We should expose all the symbolic information on some set of interprocedural variables, either:

- all the variables directly referenced in the procedure (that is, referenced besides through a contained procedure call), or
- all variables marked as contributing to interesting values (e.g., conditional branches and array bounds and subscripts).

Given that our initial phase for symbolic analysis relies on solutions of REF, MOD, and ALIAS, we also need to test for reanalysis (repeating the initial analysis for the symbolic information). The tests for reanalysis and recompilation are as follows for each procedure:

- Repeat initial symbolic analysis if this routine has been edited or if any MOD, REF, or ALIAS set of a called routine has changed.
- Repeat interprocedural symbolic analysis if any initial symbolic analysis has been repeated.
- Recompile any procedure reanalyzed in the first step above, plus any routines whose `entry` predicate set, or any `return` symbolic expressions, have changed (for variables whose information was exposed, as described above, in the previous compilation).

6.7 Other Interprocedural Problems

The annotated call graph is useful in other flow-sensitive problems. Symbolic information is vital to array section analysis. Symbolic evaluation of conditional branch predicates can improve all interprocedural solutions, and with carefully built initial information, we can converge to a stable solution in one interprocedural phase.

6.7.1 Kill and Use

Callahan's Program Summary Graph has separate sets of `entry`, `exit`, `call` and `return` nodes for every variable [Cal88]. To model the PSG, we must attach these variable reference nodes to their corresponding procedure-summary control-flow graph nodes. Dataflow analysis is performed in the initial phase, with `entry` and `exit` nodes treated as using and defining all formal parameters and global variables. Call and Return nodes are treated as using and defining all their actual parameters and global variables. These can be refined if flow-insensitive REF and MOD results are available. If they are not, we can still reduce the number of edges by treating all global variables not directly referenced in the procedure as aliased to a single dummy variable.

6.7.2 Array Section Side Effects

In Chapter 2 we showed how array side effect analysis relies on symbolic information. The initial sections are built at the same time as the initial value graphs. We must collect the list of direct array references, building value numbers and, where possible, constant bounds. Merges should be postponed until after the interprocedural symbolic propagation. The lists of sections are saved with embedded references to the value graph.

After symbolic and other interprocedural analyses are complete, the array sections are read back in and their subscript value numbers updated to reflect the results of interprocedural symbolic propagation. All the sections for a procedure, including those for call sites, are then merged to produce one each of MOD and USE summaries for each variable. The summary sections are propagated up the call graph, with a similar translation used for symbolic subscripts as is used for the symbolic return values in Algorithm 6.1.

6.7.3 Test Elision

Test elision is the process of evaluating conditional branches, then deleting the the control-flow edges and nodes proven unreachable. We know of no interprocedural problem that cannot potentially be improved.

Call Graph Construction. Call sites can be ambiguous in the presence of procedure variables (formal parameters which are bound to procedure names). If test elision removes the last call site binding a particular procedure to a procedure variable, then the ambiguity of all call sites using that procedure variable is reduced.

Alias Sets. Test elision may remove the call sites that introduce an alias (either passing a variable twice, or passing a global as a parameter).

MOD and REF. Test elision may delete the last definition or use of a variable by a procedure.

KILL. Test elision may delete the last definition-avoiding path, adding the variable to the set.

USE. Test elision may remove the last definition-free path to a use, removing the variable from the set.

Array section side effects. MOD, REF, USE and KILL may all be improved for array sections as well, by the same mechanisms as for the scalars versions.

If we have kept sufficient information about these problems in the initial information, then we can repeat these analyses in the same interprocedural phase as the symbolic analysis. All we require is that the variable access and call binding information be attached the G^{PS}_{CF}. If we have also kept enough information about the effect of these problems on symbolic information, we can iterate the analyses until convergence (or until some set number of repetitions is reached) within one interprocedural phase.

6.8 Evaluation

The current implementation provides two forms of interprocedural analysis:

Return value analysis translates expressions saved for called procedures into the context of their callers.

- A symbolic exit expression is computed for each variable modified in each procedure.
- Symbolic expressions can be computed to support other analysis, such as by representing subscripts of array section side effects.

		Constants		Linear
		integers	floats	Equalities
NAS:	`bt`	23	20	
	`cg`	9		
	`ep`	1		
	`ft`	16		
	`is`	5		
	`lu`	21	20	
	`mg`	11		
	`sp`	35	20	
Perfect:	`adm`	4		2
	`arc2d`	25		16
	`bdna`	2		1
	`flo52`	11		12
	`mdg`	4		5
	`mg3d`	3		1
	`ocean`	3		1
	`qcd`	1		
	`track`	11		
	`trfd`	1		1
RiCEPS:	`simple`	7		
	`sphot`			
SPEC:	`matrix300`	11		
Total:		204	60	39

Table 6.1 Interprocedural Predicates on Entry

Passed predicate analysis computes relations among variables on entry to each procedure. These relations are used to write interdependent symbolic expressions dependence testing and other uses.

6.8.1 Passed Predicates

Table 6.1 summarizes our results on the benchmarks of Appendix A. Some programs are omitted due to bugs in the implementation which prevented running the interprocedural analysis to completion.

The first numeric column gives the total, over all procedures in each program, the number of variables in each procedure's REF set that were found to have integer constant values. The next column gives the number of referenced variables which were found to have non-integral floating-point constant values in the intraprocedural analysis. Because fractional values can be changed by conversion to and from floating-point format, they should be manipulated via the original textual representation, not as numbers. The current implementation does not support textual propagation; however, variables with equal fractional values were marked as equal for predicate propagation.

The third data column gives the number of linear equalities. All classes of related *and referenced* variables found contained two or three variables; the former were counted as one linear equality, the latter as three.

Our value numbering infrastructure has also been used in other tests of interprocedural constant propagation [GT93].

6.8.2 Effect on Dependence Testing

Return value analysis alone did not eliminate any dependence edges in the current implementation. While the incomplete state of symbolic analysis in ParaScope may limit the improvements, the dominant factors seem to be:

- Return value propagation only reaches those procedures which called the returning procedure. This misses the common case of initialized values returned by one procedure, then passed to others for use.

- Some potential gains of symbolic analysis come from auxiliary induction variable recognition, but our current implementation is ineffective in rewriting cyclic value numbers into induction variables.

The combination of passed predicates analysis with returned value analysis does somewhat better. (Since we use symbolic expressions for returned values in building initial symbolic predicates, we did not try passed predicate analysis by itself.) Three procedures showed reductions in their dependence graphs: one from constants alone, one from constants and symbolic predicates, and one from symbolic predicates alone.

Procedure `arc2d/etadif`

There are six dependences to be disproven in `etadif`, generated by two similar sets of array references. Two are disproven through constant information, two because of symbolic pairwise equalities, and two are not disproven, but could be through propagation of asserted constant bounds. The potential dependences are generated from two sets of array references like the ones labeled `A`, `B`, and `C` below.

```
         subroutine etadif(jdim,kdim,x,y,xy)
            common /base/ jmax, kmax, ..., jbegin, jend,
        +                 kbegin, kend, ..., klow, kup, ...
            dimension xy(jdim,kdim,4)
            do 21 j=jbegin,jend
               do 20 k=klow,kup
A                 xy(j,k,2) = ...
20             continue
               k = kbegin
B              xy(j,k,2) = ...
               k = kend
C              xy(j,k,2) = ...
21          continue
            return
         end
```

The main program `arc2d` reads common block variables `jmax` and `kmax` setting the size of the problem data. Tracing assignments to other variables and parameter bindings at two levels of procedure call, we obtain these predicates on entry to `etadif`:

- `jend` $==$ `jmax` $==$ `jdim`
- `kend` $==$ `kup` $+ 1 ==$ `kmax` $==$ `kdim`
- `klow` $== 2$

- kbegin $==$ jbegin $== 1$

(Relations on kmax and jmax were not counted in the statistics for etadif because they are not referenced.)

These facts enable the following proofs of independence:

- No reuse between A and B; because kbegin is $1 \notin [2 : \texttt{kup}]$ (the bounds of k.
- No reuse between A and C; because kend $==$ kup $+ 1$, so kup $\notin [2 : \texttt{kup}]$.

But these methods failed to prove independence between B and C, because comparison of kbegin (1) with kend failed. This could be fixed were constant bounds information computed; if we assume that arrays span at least two in each dimension, then the lower bound of kend is 2.

Procedure spice/load

Five dependences are disproven in load due to our propagation of pairwise inequalities. The array references below account for two of the potential dependences.

```
          do 50 j=2,jl
             do 45 i=2,il
A               p(i,j) = ...
45           continue
B            p(1,j)    = p(il,j)
C            p(i2,j)   = p(2,j)
50        continue
```

Propagation form multiple call sites in flo52q gives us these relations on entry, which are converted to value numbers for use by dependence testing.

- j2 $==$ jl $+ 1 == 2 *$ jj2 $+ 2$
- i2 $==$ il $+ 1 == 2 *$ ii2 $+ 2$

A loop-independent output (write-write) dependence from A to C is disproven by a symbolic bounds check. Since (i2 $==$ il $+ 1$), and il is the upper bound of i, we know that (i2 $>$ i).

A loop-independent anti-dependence (read-write) from B to C is disproven by a comparison of loop-invariant symbols. Again, i2 $>$ il is the important fact.

6.9 Related Work

6.9.1 Flow-Sensitive Analysis

Banning

Banning introduced the distinction between flow-sensitive and flow-insensitive interprocedural dataflow problems [Ban79, Ban78]. For flow-insensitive problems, the control structure inside a procedure is irrelevant to summarizing the effects of imbedded calls. For example, a variable is in the MOD set of a procedure if it is potentially modified via any statement or call site. In contrast, flow-sensitive problems require control structure for their computation. A Variable is in the KILL set of a procedure if every path through the procedure passes through a call site or other statement which kills (must define) the variable.

Banning gives efficient methods for solving flow-insensitive MOD (may write) and REF (may read) problems. He also gives efficient approximations to a few flow-sensitive summary problems. A variable can be easily proven in the KILL set of a procedure if a direct definition to the variable lies on every path, or if a call to another procedure killing the variable occurs on every path.

Banning's work has inspired linear-time algorithms for and commercial implementations of flow-insensitive interprocedural analysis [CK88b, LM94]. While his approximate solutions for flow-sensitive problems are a valuable starting point, more precise methods may be profitable today.

Myers

Myers addresses flow-sensitive problems by fusing the internal control flow graphs of each procedure into one program *supergraph* [Mye81]. There is no essential difference between the supergraph and our annotated call graph. Myers refers to our `call` and `return` nodes as CPOINT and RPOINT, respectively, and does not explicitly support as rich a set of annotations.

Myers gives algorithms for flow-sensitive problems, including:

- MUST-ALIAS(p): the set of sets of variables accessible by p that are aliased on some potential execution path to p (ignoring the direction of conditional branches).

- LIVE(t): the set of variables which exist at program point t and may later be used (in or after the current procedure ends) without first being redefined or going out of scope.

Myers shows how to solve LIVE precisely (except for ignoring the directions of conditional branches) using the MUST-ALIAS sets. Under his analysis, the MUST-ALIAS set for a procedure is potentially a power set of the formal parameters and globals visible to the procedure — the set of all possible sets of them. Myers also NPstates that LIVE is NP-complete, giving a proof that LIVE is NP-hard by reducing 3-satisfiability to it.

However, Myers's analysis is overly pessimistic. Even though nesting of procedures is allowed in his framework (and therefore formals can be global variables), global variables which are not formal parameters can only be aliased to formal parameters. Let

- g_p be the number of non-formal globals visible to a procedure p,
- h_p be the number of formal parameters to nesting parents of p which are visible as globals to p, and
- f_p be the number of formal parameters to p.

Then the set of variables which may be freely aliased to each other has size $(h_p + f_p)$, and its power-set, the largest possible set of MUST-ALIAS sets, has size $2^(h_p + f_p)$. The g_p non-formal globals can be aliased to the other variables, but not to each other, so there are exactly zero or one non-formal globals in each final alias set. Therefore the total number of alias sets is $O((g_p + 1)2^(h_p + f_p))$

This bound is not excessive if common usage of procedure nesting and formal parameters is considered. Common practice does not have arbitrarily nested procedures, so we can assume that the nesting depth is bounded by some small constant, c_n. Many interesting languages, such as Fortran, C, and C++, do not allow nesting of procedures ($c_n == 1$). Furthermore, the number of formal parameters f_p to a procedure is generally bounded by some small constant c_p (such an assumption is made in [CK88b]). The global variables that can be aliased to each other arise only when formal parameters are exposed as globals to nested procedures, so h_p is bounded by $(c_n - 1) * c_p$. Therefore, the total number of MUST-ALIAS sets is bounded by $(g_p + 1)2^(c_n * c_p)$, or, removing the constants, $O(g_p)$.

With bounded nesting depth and formal parameters, LIVE is no longer -hard. Myers's complexity analysis involves no exponential components except in the size of the alias sets, which by the above analysis are linear in the number of non-formal globals. His experiments suggest that his methods may be practical.

The improved complexity analysis of Myers's methods for flow-sensitive analysis suggest further experiments using our annotated call graph. Aliases are usually rare in practice, so the MUST-ALIAS computation should not be much more expensive than the MAY-ALIAS computation [CK89]. An initial interprocedural phase could gather alias information and allow a pruned supergraph, something like the PSG described later, to be used for more efficient solution. The solutions to flow-sensitive analysis could be improved through symbolic evaluation of conditional branches.

The example of Myers's supergraph also shows that algorithms which appear exponential in theory may be efficient on programs encountered in practice.

Callahan

Callahan's Program Summary Graph (PSG) is a pruned version of the supergraph tuned to the solution of the flow-sensitive summary problems KILL and USE [Cal88]. For a variable v visible in procedure p,

- $v \in$ KILL(p) if every path through p redefines v
- $v \in$ USE(p) if there exists a path through p which encounters a use of v before encountering a redefinition of v

Using the PSG, `kill` and `use` can be defined as simple graph connectivity problems.

The nodes of the PSG are essentially the interprocedural interface nodes of G_{CF}^{PS} replicated for each interprocedural variable. The edges are built by dataflow analysis on the the procedure's control-flow graph, treating each `entry` and `return` node as a definition of its variable, each `call` and `exit` as a use, and direct definitions and uses normally.

Where an `entry` or `return` is a reaching definition for a `call` or `exit`, a PSG is added. Edges are also added from `entry` or `return` to a special `use` node, if they reach any direct use of the variable. The `entry`, `exit`, `call`, `return`, and `use` nodes, and the edges connecting them, are retained for each procedure. They are stitched together to build the full PSG by adding `call–entry` and `exit–return` edges corresponding to call sites.

Callahan's methods for solving `kill` and `use` are linear on the size of the PSG. The only significant weaknesses of the method are:

- Aliases are handled imprecisely, by performing the analysis by variable name and then considering a set of aliased variables to be killed or used only if they are all killed or used individually.
- Given g global variables and e call sites, the number of nodes in the PSG is at least $g * e$, which can be excessive.

These problems can be avoided if the alias, MOD, and REF problems are solved before building the initial information for the PSG. This would also solve Callahan's difficulties Callahan with using the PSG to solve the LIVE problem.

FIAT

FIAT is an acronym of Framework for Interprocedural Analysis and Transformation, an interprocedural analysis system descended from the $\mathbb{R}^n$ program compiler and available in the ParaScope programming environment [HMBCR93, Hal90]. Its main data structure, the annotated call graph is fundamentally a variation of Myers's supergraph [Mye81]. However, FIAT supports a much richer set of annotations than Myers proposes, and provides tools for interprocedural analyses *and transformations*.

Our scheme for symbolic and flow-sensitive analysis has been developed contemporaneously and in collaborative discussion with the FIAT authors. Our implementation of interprocedural symbolic analysis builds on the FIAT implementation in ParaScope. When the full FIAT design, including procedure-summary control-flow graphs, has been implemented, this will provide an ideal platform for studying the effects of test elision during interprocedural analysis. When the deleted code includes accesses to variables, we should see improvements in the results of side effect analysis (such as flow-insensitive MOD and REF, flow-sensitive USE and KILL).

The lack of procedure-summary control-flow graphs (G_{CF}^{PS}) in the current version of the system led us to postpone analysis of the effects of test elision within the interprocedural phase.

FIAT is capable of supporting multiple phases of interprocedural analysis, a feature which should be exploited in future experiments.

6.9.2 Symbolic Analysis

Our work constitutes a decomposition of symbolic analysis into independent handling for expressions and predicates. They are exploited together for various purposes and converted back and forth in interprocedural propagation, but they are kept fundamentally separate. We hope in this way to gain the benefits both of pattern-matching arbitrary expressions and of manipulating simple linear relations.

Prior work has relied either solely on symbolic relations [Iri93] or symbolic expressions [HP93], or else has combined them in one representation [DJ92]. Combining our analysis with the first two methods could improve all. We believe that the third method produces a unified representation that is too difficult to manipulate for the exploitation of symbolic facts.

Ours is also the first work to explicitly treat the issue of recompilation analysis for symbolic information. While symbolic predicates on variables can easily be checked for changes, symbolic expressions for returned values must be implemented carefully. Precise recompilation analysis on symbolic expressions for passed values is very difficult; minor changes can produce new expressions without conveying any worthwhile information.

PIPS

The PIPS system provides interprocedural analysis of side effects and symbolic predicates [IJT91, Iri93]. The symbolic information is exploited in several ways, including analysis of side effects to array regions (based on a previous implementation by Triolet [TIF86, Tri85]).

Relying on predicates as the main form of symbolic information can have significant disadvantages. The simplest form of predicates to manipulate, linear equalities and inequalities, miss many interesting cases, such as the equality of identical nonlinear expressions. While it may be possible to handle more complicated predicates, the linear ones are expensive enough. A set of predicates on n variables is represented as an n-dimensional polyhedron, with convex hull as the merge operation and linear programming required for composition. These algorithms take exponential time.

The PIPS researchers write that they have found efficient approximate methods for manipulating linear predicates. Comparison of the precision and expense of our methods and theirs is an important area for future work. We expect significant benefits could be achieve by combining our pattern matching for arbitrary expressions with their linear inequality analysis.

Dehbonei

Dehbonei and Jouvelot describe a partial symbolic evaluation method for interprocedural semantic analysis [DJ92]. Their symbolic representation of a program value is a pair of symbolic expressions; the first representing the conditions under which the program expression executes and the second giving the value if it executes.

The symbolic value for a variable after a control flow merge is built using the *shuffle* function, which produces a list representing the possible control conditions (each mutually exclusive predicate in the list is OR'd to give the complete execution conditions) and a list of the possible values. The value assumed is that corresponding in list order to the predicate that evaluates to true.

Use of a symbolic value unifying control conditions and value content is the important difference between their method and ours. Their unified representation should be simpler to implement. But with regard to precision and efficiency, the unified representation seems to combine the worst features of predicates and expressions.

- Powerful tools can be used to manipulate some kinds of predicates, such as linear relations. But the predicates here are just expressions; no single representation of the relations on all variables is built.

- Symbolic expressions are difficult to manipulate (except when linear), but can tested for equality by pattern matching. Combining control conditions in the symbolic value makes identical values in different control contexts appear different.

Dehbonei and Jouvelot build their symbolic values from a value graph that sounds like a relative of SSA form, being also derived from the earlier work of Reif *et al.* [RT81]. Full GSA form could perhaps also be employed in this construction; thinned GSA form omits some of the predicates they are interested in propagating.

Haghighat

Haghighat's symbolic analysis relies purely on symbolic expressions [Hag90, HP90, HP93]. His methods for expression manipulation are more highly developed than ours, while we have more efficient (or at least more explicitly described) techniques for building an initial expression from dataflow information.

Haghighat uses symbolic expressions to describe the output values of procedures, much as we do, but he also uses them to describe actual parameter values to a called procedure. He does not address the recompilation issue, but his method faces several challenges:

- Interprocedurally propagated expressions must be built carefully to reduce the occurrence of apparent but meaningless changes in the solutions.
- When there are multiple calls to a procedure, expressions for actual parameter values must be merged to produce an expression for each formal parameter. The representation of this merge function should not depend on ephemeral details of the call graph or of the calling procedures.

As discussed above, we convert information about passed values from symbolic expressions into equality relations for these reasons.

6.10 Summary

We have devised an efficient scheme for interprocedural symbolic analysis which allows recompilation analysis. However, the direct improvements in dependence information are marginal.

Return expression analysis is value for its indirect effects via array section analysis [Tsa94] and constant propagation [GT93]. Constant range propagation should be added. The forward propagation of pairwise equalities should be reconsidered, and either jettisoned, replaced with a richer predicate framework, or replaced with a passed expression framework, if one can be made compatible with recompilation analysis.

Chapter 7

Conclusions and Future Work

Computer hardware and software are two of the fastest-evolving areas of modern technology. Designers of optimizing compilers must keep pace with these changes while answering the unchanging demand to produce efficient machine-specific executables from maintainable portable source code.

7.1 Contributions

We have given a solid framework for symbolic analysis within procedures and demonstrated its efficiency and effectiveness on substantial Fortran benchmarks. Our extension of this technique to interprocedural analysis provides information needed to support dependence testing and is compatible with recompilation analysis.

Array Section Analysis. We have demonstrated an accurate and efficient technique for summary side-effect analysis on subarrays.

Thinned GSA Form. We give new algorithms are given for constructing this variant of SSA form. They are both simpler than those given for original GSA form and better suited to value numbering on the fly.

Global Value Numbering. Combination with simplification produces a symbolic analysis method strictly superior to that in PFC.

Interprocedural Symbolic Analysis. We propagate symbolic expressions for return values and linear equality relations on passed values while remaining compatible with recompilation analysis.

Robust Implementation. Other researchers are already using this implementation to support array section analysis and slicing analysis.

7.2 Evaluation

Our implementations and experiments in the course of this research have revealed significant differences in the implementation effort, compile-time expense, and run-time effectiveness of different program analysis methods. Here we give our recommendations as to which analyses should be included in a modern compiler. The columns in each table give the following information:

Resources: Gives the expected complexity of the analysis over the programs encountered in practice. For example, methods with entry X are asymptotically linear or almost linear in X, or else have exhibited linear behavior in extensive experiments. Methods with entry $X+$ are usually linear in practice, but have significant potential for nonlinearity with unusual programs.

Implement: An estimate of the programming effort needed to add the analysis to a compiler, provided that the major prerequisite analyses are already available.

- *Simple* implementations can be finished by one skilled programmer, working from published algorithms, in a fraction of a year.
- *Moderate* implementations require a large portion of a year and more interpretation of the published methods.
- *Hard* implementations require many prerequisite analyses, much interpretation of the published work, and, especially for a production compiler, substantial performance tuning not yet addressed in the literature.

Payoff: This assumes that the compiler is testing for array dependences and substantially restructuring the program's loops for parallelism and locality.

- *Enabler* analyses have little benefit on their own, but provide an infrastructure for useful techniques.
- *High* payoff is achieved if many loop-parallelization or other profitable transformation opportunities will be enabled, and some kind of improvement will be seen in most programs.
- *Moderate* payoff is achieved if multiple examples of code improvement have been found, but widespread benefits have not yet been shown.
- *Small* payoff is reported when few or no profitable transformations of realistic programs have been experimentally derived by this method.

Recompile: For interprocedural analysis methods, this refers to the difficulty of assessing changes in solutions, so as to decide which procedures must be reanalyzed or recompiled after an edit.

- *Simple* recompilation problems require only the comparison of bit vectors and constant values.
- *Moderate* problems require the comparison of symbolic expressions that can be written in a normal form.
- *Hard* problems require symbolic information that is difficult to isolate to the context of one procedure.

Recommended (or "Rec'd") gives our conclusions on which analyses are justified for a modern production compiler. A check mark indicates that the analysis has payoff justifying its expense. "Required" indicates that the analysis is part of the enabling infrastructure for many profitable analyses and transformations. Blank entries are not recommended, either because the cost is high or their proven benefits are particularly small.

7.2.1 Intraprocedural Analysis

Table 7.1 gives our assessments of analyses conducted on a single procedure. Here, N represents the size of the control flow graph (nodes and edges) and V represents the number of variables or symbolic expressions for which facts are being propagated at any point.

	Resources	Implement	Payoff	Recommended
Tarjan intervals, dominators	N	simple	enabler	required
Control dependence graph	N	simple	enabler	$\surd$
SSA form	N	simple	enabler	required
Constant propagation	N	simple	high	$\surd$
Value numbering	$N+$	moderate	high	$\surd$
Constant ranges	N	simple	high	$\surd$
GSA form	$N+$	moderate	small	
Relational predicates	$NV^2+?$	hard	high?	

Table 7.1 *Intra*procedural Analysis Evaluation

Dominators are essential to the construction of control dependences and SSA form, as are Tarjan intervals if we wish to know which edges are loop-carried. These four analyses provide an inexpensive basic infrastructure for a compiler. Once these are available, constant propagation and constant range analysis are trivial.

Some method for symbolic expression manipulation is needed for dependence testing. Our value numbering techniques fill this need, while also supporting common subexpression elimination and test elision. Complicated code inserted by advanced transformations, such as run-time preprocessing of loops, can also benefit from recognition of redundant computations [DSvH93].

Gated single-assignment form does not show many direct benefits in our experiments. New analysis and transformations may exploit its unique properties in the future, however [Wol92]. Both GSA form and value numbering show generally linear behavior in our experiments. However, the worst-case bounds are decidedly nonlinear, as borne out by a few procedures with tangled control flow.

The worst case for exhaustive propagation of relational predicates ranges from cubic for pairwise linear equalities to exponential for general linear inequalities. While these seem too time-consuming for use in a compiler, some implementors have had good experience with propagating such information only on demand of dependence testing [Iri93]. Limiting analysis to cases where we know (from preliminary checking) that it may be useful will be increasingly important to producing powerful and fast compilers.

	Resources	Recompile	Implement	Payoff	Rec'd
MAY-ALIAS, MOD, REF	nV	simple	simple	high	required
Constant propagation	nv	simple	moderate	high	√
KILL, USE, LIVE	Nv	simple	moderate	high	√
Constant ranges	nv	simple	moderate	high	√
Return expressions	depends	moderate	moderate	enabler	√
Array MOD, REF	nv	moderate	moderate	moderate	√
Array KILL, USE, LIVE	Nv	moderate	hard	moderate	
Pairwise equalities	nv^2	simple	moderate	small	
Passed expressions	depends	hard	hard	?	

Table 7.2 *Inter*procedural Analysis Evaluation

7.2.2 Interprocedural Analysis

Table 7.2 gives our recommendations on the use of interprocedural analysis. Here, n represents the size of the call graph (nodes and edges) and N represents the combined size of the call graph and control flow graphs for each procedure. V represents the number of interprocedurally interesting variables, while v represents the intersection of the interprocedural variables with the MOD or REF sets of the procedure, as appropriate.

Basic alias and flow-sensitive side-effect solutions are prerequisites to more extensive analysis. MUST-ALIAS, KILL, USE and even LIVE solutions can be approximated much more efficiently than the worst case given by Myers and can have substantial payoff [Mye81, Cal88].

Constant propagation can be very useful, but requires at least an intraprocedural symbolic representation [CCKT86, GT93]. Propagation of constant ranges is not quite so simple or effective as that of pure constants, but can have a significant effect in proving loop bounds to be positive.

Return expressions have little direct effect, but the same representation and translation infrastructure is essential to propagation of array section information and to analysis of passed values. Array side effect analysis is important to parallelization, although the KILL and USE information needed for privatization are flow-sensitive.

Analysis of passed values can be very important to dependence testing, especially when a program makes extensive use of dynamic problem sizes. While pairwise linear equalities are an efficient framework, their effectiveness has not been fully confirmed. Merely using symbolic expressions for passed values introduces difficulties in representing values computed in other procedures, and knowing what changes to those values do and do not invalidate previous analysis, forcing a recompilation.

The complexity of propagating return expressions and passed expressions depends on the size of the expressions. Since building a monolithic whole-program value graph is ruled out by recompilation issues, symbolic expressions must be copied during propagation from one procedure to another. In practice, we may require a limit on the size of expressions propagated.

7.3 Future Work

Several directions suggest themselves for further implementation in ParaScope.

Constant ranges. Our experiments have already exposed cases where constant bounds information will improve dependence testing. Because array addressing frequently runs from a constant lower bound to some symbolic upper bound (varying with the size of the data), it will pay to determine minimum values for these upper bounds. The analysis techniques of Chapter 5 are well-suited to the computation of these ranges within procedures. The interprocedural propagation framework of Chapter 6 can be used to propagate range information between procedures, where it combines profitably with the linear equality predicates already handled.

Test elision. Our implementation provides most of the framework needed to delete unexecutable code based on the values of conditional tests. When compiling a single procedure (with or without a preceding interprocedural analysis), it is sufficient to build the symbolic expression for the test. If the simplified expression returned is a constant such as `true` or `false`, then we delete all control flow edges from the branch save the one so labeled. Test elision may also be employed during interprocedural analysis to improve the information used by other interprocedural analyses, as described in Chapter 6.

More sophisticated predicates. The value of affine inequalities on arbitrary numbers of variables (or value numbers) should be investigated. Efficient implementations exist for composing inequalities into a convex set [Pug91]. However, merging such sets appears to be more difficult [CH78]. The best approach may be to have a backwards pass of analysis that determines which values are particularly interesting, then restrict the forward analysis to compute predicates only on the interesting values (like goal-directed cloning) [BCHT90].

More array analysis. The same pattern-matching techniques applied in value numbering scalar computations can be applied to array operations. We already support value numbering of subscripted array references treated as `access` and `update` functions [DSvH93]. These techniques can be further improved by noticing when arrays are accessed with only constant subscripts, then analyzing each of those elements as separate scalars.

7.4 Conclusion

We have presented techniques for symbolic analysis, realized them in an efficient implementation, and shown their effectiveness on a variety of publicly-available Fortran applications.

The use of a consistent framework and the avoidance of ad-hoc techniques make it possible for programmers using the system to know which constructs are hard to analyze and which can be handled by the system. In particular, the perceived cost of procedure calls should be reduced, encouraging the use of modular design. Continued progress in this area brings us continually closer to the goal, yet so far away, of automatic parallelization of scientific programs.

Bibliography

[AC72] F. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[ACK86] Randy Allen, David Callahan, and Ken Kennedy. An implementation of interprocedural analysis in a vectorizing Fortran compiler. Technical Report TR86-38, Dept. of Computer Science, Rice University, May 1986.

[AH82] Marc Auslander and Martin Hopkins. An overview of the PL.8 compiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, June 1982.

[AK84] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.

[AK87] J. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[All83] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, April 1983.

[ASU86] Alfred V. Aho, Ravi I. Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 1–11, San Diego, CA, January 1988.

[Bal89] Vasanth Balasundaram. *Interactive Parallelization of Numerical Scientific Programs.* PhD thesis, Rice University, July 1989. Available as Rice COMP TR89-95.

[Bal90] Vasanth Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: the Data Access Descriptor. *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[Ban78] J. Banning. *A Method for Determining the Side Effects of Procedure Calls*. PhD thesis, Stanford University, August 1978.

[Ban79] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, January 1979.

[Ban86] U. Banerjee. A direct parallelization of CALL statements – a review. CSRD Rpt. 576, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, April 1986.

[Bar77] J. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the Fourth ACM Symposium on the Principles of Programming Languages*, Los Angeles, January 1977.

[BBDS93] David Bailey, Eric Barszcz, Leonardo Dagum, and Horst Simon. NAS parallel benchmark results. Technical Report RNR-92-002, NASA Ames Research Center, February 1993.

[BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, June 1986.

[BCHT90] P. Briggs, K. Cooper, M. W. Hall, and L. Torczon. Goal-directed interprocedural optimization. Technical Report TR90-147, Dept. of Computer Science, Rice University, December 1990.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 311–321, San Francisco, California, June 1992.

[BGNP93] Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors. *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, Portland, OR, August 1993. Springer Verlag.

[BK89] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism-enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[BKK+89] V. Balasundaram, K. Kennedy, U. Kremer, K. S. McKinley, and J. Subhlok. The ParaScope Editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[BMO90] R. Ballance, A. Maccabe, and K. Ottenstein. The Program Dependence Web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, pages 257–271, White Plains, New York, June 1990.

[Bou92] Raymond T. Boute. The euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, April 1992.

[Cal87] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, March 1987.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, pages 47–56, Atlanta, GA, June 1988.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on the Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CCKT86] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 1986.

[CF89] Robert S. Cartwright and Matthias Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, Oregon, June 1989.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CFS90a] R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. In *Proceedings of the SIGPLAN '90 Conference on*

Program Language Design and Implementation, pages 337–351, White Plains, New York, June 1990.

[CFS90b] R. Cytron, J. Ferrante, and V. Sarkar. Experience using control dependence in PTRAN. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth ACM Symposium on the Principles of Programming Languages*, pages 84–96, 1978.

[CHT91] K. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[CK85] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6*, July 1985.

[CK88a] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[CK88b] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN 88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 49–59, Austin, Texas, January 1989.

[CKB93] Philip L. Campbell, Ksheerabdhi Krishna, and Robert A. Ballance. Refining and defining the Program Dependence Web. Technical Report TR 93-6, Department of Computer Science, University of New Mexico, March 1993.

[CKPK90] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer performance evaluation and the Perfect benchmarks. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[CKT86] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[Cou81] Patrick Cousot. Semantic foundations of program analysis. In S. S. Muchnick and M. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 303–342. Prentice-Hall,New Jersey, 1981.

[DBMS79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, 1979.

[DJ92] Babak Dehbonei and Pierre Jouvelot. Semantical interprocedural analysis by partial symbolic evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Francisco, California, June 1992.

[DSvH93] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In Banerjee et al. [BGNP93], pages 152–168.

[Fie92] John Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, San Francisco, California, June 1992.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GKT91] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[GS90] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[GT93] Dan Grove and Linda Torczon. Interprocedural constant propagation: A study of jump function implementation. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 90–99, Albuqueque, NM, June 1993.

[Hag90] Mohammad Reza Haghighat. Symbolic dependence analysis for high performance parallelizing compilers. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1990. Also available as CSRD Rpt. No. 995.

[Hal90] Mary Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, October 1990.

[Har77] W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[Hav93] Paul Havlak. Construction of thinned gated single-assignment form. In Banerjee et al. [BGNP93], pages 477–499.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[HKMC90] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[HMBCR93] Mary W. Hall, John M. Mellor-Brummey, Alan Carle, and René G. Rodríguez. Fiat: A framework for interprocedural analysis and transformations. In Banerjee et al. [BGNP93], pages 522–545.

[HP90] M. Haghighat and C. Polychronopoulos. Symbolic dependence analysis for high performance parallelizing compilers. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[HP93] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In Banerjee et al. [BGNP93], pages 567–585.

[HPR88] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 146–157, San Diego, CA, January 1988.

[IJT91] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[Iri93] Francois Irigoin. Interprocedural analyses for programming environments. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*. Elsevier Science Publishers, 1993.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[Ken81] K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnick and M. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 1–54. Prentice-Hall,New Jersey, 1981.

[KMT91a] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KMT91b] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[Knu73a] Donald E. Knuth. *The Art of Computer Science Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, 1973.

[Knu73b] Donald E. Knuth. *The Art of Computer Science Vol. 3: Searching and Sorting*. Addison-Wesley, Reading, Massachusetts, 1973.

[Li90] Zhiyuan Li. Private communication, October 1990.

[LM94] Jon Loeliger and Robert Metzger. Developing an interprocedural optimizing compiler. *SIGPLAN Notices*, 29(4):41–48, April 1994.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1:121–141, 1979.

[LY88a] Z. Li and P.-C. Yew. Efficient interprocedural analysis for program parallelization and restructuring. In *ACM SIGPLAN PPEALS*, pages 85–99, 1988.

[LY88b] Z. Li and P.-C. Yew. Interprocedural analysis and program restructuring for parallel programs. CSRD Rpt. No. 720, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, January 1988.

[LY88c] Z. Li and P.-C. Yew. Program parallelization with interprocedural analysis. *The Journal of Supercomputing*, 2:225–244, 1988.

[Mas92] Vadim Maslov. Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 152–161, San Francisco, June 1992.

[Mye81] E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth ACM Symposium on the Principles of Programming Languages*, pages 219–230, January 1981.

[Por89] Allan Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989. Available as Rice COMP TR88-93.

[Pug91] William Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[RL86] John H. Reif and Harry R. Lewis. Efficient symbolic analysis of programs. *Journal of Computer and System Sciences*, 32(3):280–313, 1986.

[Ros90] C. M. Rosene. *Incremental Dependence Analysis.* PhD thesis, Rice University, March 1990. Available as Rice COMP TR90-112.

[RT81] J. H. Reif and R. E. Tarjan. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, 11(1):81–93, February 1981.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 12–27, San Diego, CA, January 1988.

[Sel92] Rebecca P. Selke. *A Semantic Framework for Program Dependence.* PhD thesis, Rice University, 1992.

[Tar74] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia,Pennsylvania, 1983.

[TIF86] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 176–185, Palo Alto, CA, July 1986.

[Tor85] L. Torczon. *Compilation Dependences in an Ambitious Optimizing Compiler*. PhD thesis, Dept. of Computer Science, Rice University, May 1985.

[TP93] Peng Tu and David Padua. Automatic array privatization. In Banerjee et al. [BGNP93], pages 500–521.

[Tri85] R. Triolet. Interprocedural analysis for program restructuring with Parafrase. CSRD Rpt. No. 538, Dept. of Computer Science, University of Illinois at Urbana-Champaign, December 1985.

[Tsa94] Hariklia Tsalapatas. Interprocedural array side effect analysis. Master's thesis, Rice University, March 1994.

[Uni89] J. Uniejewski. SPEC Benchmark Suite: designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.

[Wol92] Michael Wolfe. Beyond induction variables. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, California, June 1992.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[X3J89] X3J3 Subcommittee of ANSI. *American National Standard for Information Systems Programming Language Fortran: S8 (X3.9-198x)*. American National Standards Institute, New York, NY, 1989.

[YHR89] Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical Report 840, Computer Sciences Department, University of Wisconsin-Madison, April 1989.

Appendix A

Experimental Benchmarks

We have worked hard to make our system robust for realistic programs. In this process, we have examined its performance and stability on a number of relatively standard benchmarks. Minor changes were made to the source files to eliminate constructs poorly handled by ParaScope, such as statement functions and alternate entries, and to convert some `DATA` statements to assignments to make them visible to the symbolic analysis (which does not yet extract constants from `DATA` initializers).

In cases where a program was available in slightly different versions from multiple benchmark suites, we chose one version arbitrarily. Procedures never called were omitted. The line count is the number of Fortran records; the statement count is the number of executable statements for which we build G_{CF} nodes (not counting extra nodes which correspond to no statements).

A.1 Perfect Club Benchmarks

The Perfect Club Benchmarks are a set of scientific applications collected for evaluation of complete supercomputer compiler and hardware systems [CKPK90]. The copies of these programs used in our experiments were obtained around August 1991.

A.2 SPEC Benchmarks

The Standard Performance Evaluation Corporation (SPEC) benchmark suite is the result of another effort to build a set of complete programs for measuring system performance [Uni89]. We examine only the Fortran programs in the benchmark, which were chosen to stress floating-point performance. The programs studied are from the first set of the benchmarks, `SPECfp89`, which we obtained around March of 1991.

A.3 RiCEPS

The Rice Compiler Evaluation Program Suite (RiCEPS) is a set of programs chosen to stress compiler analysis and transformation for advanced architectures. This collection has been stable since about February 1990, and is available by anonymous FTP from `cs.rice.edu` in the `public/riceps` directory. Some programs from RiCEPS

Program	procedures	lines	statements
`adm`	90	5733	3277
`arc2d`	39	3904	1796
`bdna`	43	3808	2702
`dyfesm`	70	6998	1578
`flo52`	35	2009	1711
`mdg`	18	1250	743
`mg3d`	30	2851	2247
`ocean`	38	4353	1506
`qcd`	36	2321	1299
`spec77`	68	3915	2533
`track`	41	3706	1345
`trfd`	7	323	241
Total	515	41141	21795

Table A.1 Perfect Club Benchmarks

Program	procedures	lines	statements
`doduc`	41	5334	4469
`fpppp`	38	2729	1615
`matrix300`	5	431	124
`nasa7`	17	1117	609
`spice`	126	18417	13466
`tomcatv`	1	195	129
Total	228	28223	20414

Table A.2 SPEC Benchmarks

have been used, along with other programs gathered at the same time, in other compiler experiments at Rice [Por89, CHT91].

Program	procedures	lines	statements
boast	49	7562	5578
ccm	146	23574	7581
hydro	36	12474	1483
simple	8	1313	736
sphot	7	1146	625
wanal1	11	2099	1123
wave	94	7840	4478
Total	351	56008	22037

Table A.3 RiCEPSPrograms

A.4 NAS Benchmarks

The NAS (Numerical Aeronautical Simulation) benchmark suite from NASA Ames Research Center was *not* designed for evaluation of compilers [BBDS93]. These codes were instead written as reference implementations solving problems of interest to supercomputer users at NASA Ames. Vendors benchmarking their machines are free to use different implementations or even completely different algorithms to get maximal performance, so long as the numerical results are within tolerance.

These reference versions of the NAS benchmarks can be considered examples of codes solving interesting problems, but which have not been aggressively tuned. They therefore make an interesting contrast to the mature codes selected for RiCEPSand Perfect, in which hand optimization sometimes creates obstacles to analysis.

We obtained our copy of the NAS benchmarks from the anonymous FTP site at Syracuse University in September 1992.

Program	procedures	lines	statements
`bt`	19	4457	1448
`cg`	13	857	256
`ep`	4	265	110
`ft`	11	773	319
`is`	6	305	128
`lu`	17	3285	978
`mg`	14	598	291
`sp`	26	3516	1139
Total	110	14056	4669

Table A.4 NAS Benchmarks