Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form

Michael P. Gerlek
B.S., Computer Science, Merrimack College, 1991

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

January 1996

The thesis “Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form” by Michael P. Gerlek has been examined and approved by the following Examination Committee:

Michael Wolfe
Professor
Thesis Research Adviser

James Hook
Associate Professor

Dino Oliva
Senior Research Associate

for Kara

Acknowledgements

First and foremost, I must thank Dr. Michael Wolfe for introducing me to the realm of academic research and the field of high performance compilers. For four years he has served as a superior teacher and advisor to me; I owe him an immeasurable debt. The road to this thesis turned rocky at the end, and I am grateful for his continual support of my efforts and my work.

Dr. James Hook has provided me with a role model for academic and professional quality and has been a good friend who supported and encouraged me when I needed it most. Outside of the realm of compilers, the most important lessons I learned at OGI I have learned from Jim.

Thanks also go to the others I have worked with at OGI. The members of the Sparse Group – Eric Stoltz, Akiyoshi Wakatani, Priyadarshan Kolte, and Tito Autrey – all helped contribute to this thesis by questioning my claims and freely sharing their own views with me. Eric in particular helped shaped much of the ideas and implementations presented here. And, of course, thanks and gratitude go also to my stress-break partner, Jef Bell, who does a pretty good triple for a functional programmer.

Mike Wolfe has literally taught me everything I know about compilers, SSA, and induction variables. Nonetheless, the responsibility for any problems or errors within this thesis is entirely mine.

This research was supported by ARPA under grant F3062-92-C-135 and has been previously published under the same title in ACM Transactions on Programming Languages and Systems, *vol. 17, no. 1 (January, 1995).*

Contents

List of Tables

List of Figures

Abstract

Beyond Induction Variables:
Detecting and Classifying Sequences
Using a Demand-Driven SSA Form

Michael P. Gerlek, M.S.
Oregon Graduate Institute of Science & Technology, 1996

Supervising Professor: Michael Wolfe

Linear induction variable detection is usually associated with the strength reduction optimization. For restructuring compilers, effective data dependence analysis requires that the compiler detect and accurately describe linear and nonlinear induction variables as well as more general sequences. In this thesis we present a practical technique for detecting a broader class of linear induction variables than is usually recognized, as well as several other sequence forms, including periodic, polynomial, geometric, monotonic, and wrap-around variables. Our method is based on Factored Use-Def (FUD) chains, a demand-driven representation of the popular Static Single Assignment (SSA) form. In this form, strongly connected components of the associated SSA graph correspond to sequences in the source program: we describe a simple yet efficient algorithm for detecting and classifying these sequences. We have implemented this algorithm in Nascent, our restructuring Fortran 90+ compiler, and we present some results showing the effectiveness of our approach.

Chapter 1

Introduction

The process of detecting and classifying induction variables is usually associated with strength reduction; the most common candidates for this optimization, and therefore the most important induction variable candidates, are array address expressions in inner loops. Techniques for detecting and classifying linear induction variables have a long history and are well known [1, 2, 3].

Many modern compilers now include advanced loop transformations such as *loop distribution* and *loop interchanging* [17] that have proven useful on a wide variety of systems, ranging from uniprocessor workstations to vector multiprocessors and massively parallel processors. These transformations require analysis of array subscripts to determine data dependence relations within loops. Current methods to test and characterize data dependence relations for subscripted array references require the subscript expressions to be a linear combination of induction variables in the enclosing loops.

Research into data dependence testing has found that variables used in subscript expressions are not necessarily linear induction variables, but can often take the form of polynomial and geometric induction expressions, periodic sequences, monotonic sequences, and wrap-around variables: Eigenmann et al note one scientific code where a speedup of eight was obtained by recognizing a geometric sequence [9]. Current compilers recognize specific forms of these expressions by ad hoc pattern recognition algorithms.

Consider the following Fortran loop:

```
np1 = n + 1
j = n
```

```
do i = 1, n
    B(i) = (A(j) + A(i)) / 2.0
    j = i
    C(i) = B(i) + B(np1)
enddo
```

In the absence of any aggressive analysis, two problems arise. First, `j`, used as an array subscript, is not an induction variable but a wrap-around variable — only after the first iteration does it follow an induction sequence. This restricts the compiler's ability to, for example, vectorize the assignment. Second, the value of `np1` may not be known in the loop; a dependence must be assumed between the assignment to `B` and the second use of `B`. However, after forward substituting and peeling off the first iteration of the loop, the loop is transformed:

```
np1 = n + 1
j = n
B(1) = (A(n) + A(1)) / 2.0
j = 1
C(1) = B(1) + B(n+1)
do i = 2, n
    B(i) = (A(i-1) + A(i)) / 2.0
    j = i
    C(i) = B(i) + B(n+1)
enddo
```

Now, having removed `j` from the first assignment in the loop, the statement may vectorize. Within the loop, the second use of `B` will not be affected by the first assignment, since the subscript is in terms of `n`.

This thesis presents an important new technique for recognizing and classifying such *sequence variables* as the wrap-around variable and the linear induction variables in the program above, implicitly performing global symbolic forward substitution as well. The technique is based on a demand-driven interpretation of the Static Single Assignment

(SSA) form of the source program [8, 19]. We present an algorithm for detecting and classifying strongly connected components within the SSA representation, corresponding to various forms of these sequences. Symbolic expressions are used to represent the value of each expression within the program, and these expressions are then propagated across the program [23].

The method presented here is simple and intuitive. In its simplest form, it represents a new approach to SSA-based constant propagation; taken further, it provides a method for detecting linear induction variables as a precursor to traditional strength reduction. In the extreme, it provides a fast and efficient method for the classification and symbolic representation of complex subscript expressions needed for advanced dependence analysis in high-performance compilers. This technique has also proven useful as a basis for other optimizations, such as array bounds check analysis, which can make use of the ability to determine linear induction expressions for subscripts. We have implemented this technique in Nascent, our experimental, restructuring Fortran 90+ compiler, and present experimental results showing the effectiveness of our approach and its usefulness in dependence analysis.

In the next chapter, we describe and present examples of the different types of sequences addressed by our technique. In Chapters 3 and 4, our SSA-based framework is introduced, and the algorithm for detecting and classifying sequences is presented. Issues dealing with loop structures are addressed in Chapter 5, and in Chapter 6 some experimental results are shown. Chapters 7 and 8 conclude with a discussion of related work and a summary of the key aspects and relevance of this new approach.

Chapter 2

A Bestiary of Sequence Forms

The focus of this thesis is a scheme to recognize certain sequences and the variables that define them, which will potentially aid in the subsequent analysis and optimization of programs. Such sequences include arithmetic series, periodic functions, and monotonically increasing series, among others.

Definition 1 *Given a statement s within the body of a loop l assigning some arbitrary expression e to a scalar, integral variable v:*

```
   ...
l: loop
      ...
s:    v = e
      ...
   endloop
   ...
```

If expression e contains an occurrence of v, then v is a basic sequence variable *in l, and e is the associated* sequence expression*.*

Definition 2 *Given a statement s within the body of a loop l assigning some arbitrary expression e to a scalar, integral variable v:*

```
   ...
l: loop
      ...
s:    v = e
      ...
```

```
    endloop
    ...
```

If expression e does not contain an occurrence of v but does contain an occurrence of some sequence variable w, then v is a derived sequence variable *in l.*

The first definition states, intuitively, that a basic sequence variable is a variable "carried around the loop." A derived sequence variable is defined in terms of another sequence variable (or variables), as opposed to being defined in terms of itself.

Note that there may be more than one assignment to a sequence variable within a loop. Also note that trivial cases such as `i = 3*i - 3*i + 1` will be simplified; although `i` occurs on the right-hand side (rhs) of this assignment, the multiplications will be removed when simplifying, yielding `i = 1`. In general, after simplification the sequence variable must occur exactly once on the rhs.[1]

Definition 3 *Associated with each loop l is a basic loop counter, h_l, whose value is zero on the first iteration of the loop and is incremented by one at the end of each subsequent iteration, i.e.,*

```
    ...
    $h_l$ = 0
 l: loop
        ...
        $h_l$ = $h_l$ + 1
    endloop
    ...
```

(We will omit the subscript l where the meaning is clear.)

The basic loop counter will be used to provide closed-form expressions for certain sequences. We will reserve the term *induction variable* (IV) for the specific classes of sequence variables with well-defined closed forms.

[1]Geometric induction variables are an exception, as we will show.

2.1 Linear Induction Variables

The most important candidates for strength reduction and a common form of array subscript expression derive from linear induction variables. Intuitively, a *basic linear induction variable* is a variable that is assigned in a loop and incremented by a constant amount on every iteration [1].

More generally, a linear IV can be defined in terms of itself and some linear combination of constants and other linear induction variables. This loop shows a few different types of linear IVs:[2]

```
i = 0
j = k = 1
loop
    i = i + 2
    j = k + n
    k = j + 1
    l = t + 4*i
    A(l+1) = ...B(l*2) ...
endloop
```

The assignments within the loop to variables `i`, `j`, `k`, and `l` each define a linearly increasing series, which may be expressed in terms of the basic loop counter, h:

$$\begin{array}{rcll} h & = & \{0, 1, 2, \ldots\} & \\ \mathtt{i} & = & \{2, 4, 6, \ldots\}, & 2h + 2 \\ \mathtt{j} & = & \{1 + \mathtt{n}, 2 + 2\mathtt{n}, 3 + 3\mathtt{n}, \ldots\}, & (\mathtt{n} + 1)h + \mathtt{n} + 1 \\ \mathtt{k} & = & \{2 + \mathtt{n}, 3 + 2\mathtt{n}, 4 + 3\mathtt{n}, \ldots\}, & (\mathtt{n} + 1)h + \mathtt{n} + 2 \\ \mathtt{l} & = & \{8 + \mathtt{t}, 16 + \mathtt{t}, 24 + \mathtt{t}, \ldots\}, & 8h + \mathtt{t} + 8. \end{array}$$

The variable `i` is a basic linear IV: its initial value within the loop is 2, and on each subsequent iteration it is incremented by 2. The variables `j` and `k` are *mutual* IVs [3], since they are defined in terms of each other: `j` and `k` have initial values of $\mathtt{n} + 1$ and

[2] In example loops, we will usually omit loop termination tests and exits.

$\mathtt{n}+2$, respectively, and both are incremented by the loop-invariant value $\mathtt{n}+1$ on each subsequent iteration. The variable `l` is not incremented on each iteration, but since it is assigned a linear function of another induction variable, `i`, and a loop-invariant value, `t`, it is a derived IV. This implies the subscript expressions of `A` and `B`, `l+1` and `l*2`, are induction expressions as well.

The sequence expression of a basic linear IV may include a subtraction operation, provided the right operand is not the induction variable itself, i.e., `m = n - m` is not linear.

2.2 Polynomial Induction Variables

Traditionally, induction variables recognized by compilers are linear functions of a loop index, formed by the addition of loop-invariant values. When the term added to the induction variable is a linear IV, however, a *polynomial* IV may result. In the program,

```
i = 0
j = k = 1
loop
    i = i + 1
    j = j + i
    k = k + j + 1
endloop
```

the linear IV `i` is used to define the polynomial IVs `j` and `k`:

$$\begin{aligned} h &= \{0,1,2,3,\ldots\} \\ \mathtt{i} &= \{1,2,3,4,\ldots\}, && h+1 \\ \mathtt{j} &= \{2,4,7,11,\ldots\}, && \tfrac{1}{2}h^2+\tfrac{3}{2}h+2 \\ \mathtt{k} &= \{4,9,17,29,\ldots\}, && \tfrac{1}{6}h^3+h^2+\tfrac{23}{6}h+4. \end{aligned}$$

A sequence variable whose expression contains an addition of a polynomial IV, e.g., `k`, yields a polynomial of a correspondingly higher degree. The degree of the closed-form expression is equal to $d+1$, where d is the degree of the IV added (for linear IVs, $d=1$). Thus, `j` and `k` are polynomials of degree 2 and degree 3, respectively.

For polynomial IVs, the sequence expression may include subtraction, again provided the right operand is not the IV being defined.

2.3 Geometric Induction Variables

In addition to linear and polynomial sequences, geometric sequences may also be described. These arise when the sequence variable is multiplied by some loop-invariant value. In the program,

```
l = 1
loop
    l = 2*l + 1
endloop
```

the variable `l` is a geometric IV defining the sequence

$$\begin{aligned} h &= \{0, 1, 2, 3, \ldots\} \\ \texttt{l} &= \{3, 7, 15, 31, \ldots\}, \quad 2^{h+2} - 1. \end{aligned}$$

For geometric sequences, the sequence expression must contain a multiplication of the sequence variable by a loop-invariant value; this multiplicative factor defines the base of the geometric term in the sequence expression. The rhs may also consist of additions or subtractions of loop-invariant values, linear IVs, and polynomial IVs.

2.4 Wrap-Around Variables

Wrap-around variables occur when a variable is assigned a value from outside the loop on the first iteration, and then takes on the pattern of another sequence variable (typically a linear IV) for the remainder of the iterations [17]. Such forms are typically encountered when the elements of an array are to be "wrapped around" a cylinder, as in this example:

```
im1 = n
i = 1
loop
```

```
    i = i + 1
    A(i) = A(im1) + ...
    im1 = i
endloop
```

Here `im1` is a wrap-around variable at the assignment to `A(i)`; at the use, it has value `n` on the first iteration and then follows a linear sequence:

$$\begin{aligned} h &= \{0,1,2,3,\ldots\} \\ \mathtt{i} &= \{2,3,4,5,\ldots\}, \quad h+2 \\ \mathtt{im1} &= \{\mathtt{n},2,3,4,\ldots\}, \quad \langle \mathtt{n}, h+1\rangle_{wrap}. \end{aligned}$$

The notation $\langle i_1,\ldots,i_d,f\rangle_{wrap}$ indicates the sequence has values $i_1,\ldots,i_d$ on the first d iterations and follows the sequence defined by f thereafter.

Compilers typically recognize such forms with a separate pattern-matching phase run after induction variable detection: wrap-around variables may be identified as derived sequence variables that are used in the loop before being assigned. When wrap-around variables are detected, the first iteration may be peeled off the loop, and the wrap-around variable may be treated as an induction variable, as shown in the introduction.

Wrap-around variables may be cascaded: if the sequence being taken on after the first iteration is another wrap-around variable, the order of the variable being defined is one greater.

```
im1 = n
im2 = n2
i = 1
loop
    i = i + 1
    A(i) = A(im1) + A(im2) + ...
    im2 = im1
    im1 = i
endloop
```

Here the use of `im2` as a subscript expression takes on the value `n2` on the first iteration, `n` on the second, and then follows the sequence $\{2,\ 3,\ 4,\ \ldots\}$, which is represented as $\langle \mathtt{n2}, \mathtt{n}, h\rangle_{wrap}$.

2.5 Periodic Sequences

A method used in some relaxation codes is the generation of a "new" matrix of values from the matrix of "old" values. A simple way to implement such a scheme is to represent the matrix with an array having one extra dimension of size two; this array holds both the "old" and "new" matrices. A *flip-flop* variable is then used to swap between them:[3]

```
k = 1
kold = 2
loop
    A(i,j,k) = ...A(i,j,kold) ...
    ktemp = k
    k = kold
    kold = ktemp
endloop
```

At its use in the subscript expression, the variable `k` is a *periodic variable* with a period of 2:

$$\begin{aligned} \mathtt{k} &= \{1,2,1,2,\ldots\}, \quad \langle 1,2\rangle_{per} \\ \mathtt{kold} &= \{2,1,2,1,\ldots\}, \quad \langle 2,1\rangle_{per}. \end{aligned}$$

(The notation $\langle i_1,\ldots,i_d\rangle_{per}$ indicates that the sequence cycles through the values $i_1,\ldots,i_d$.) An optimizing compiler may profit by determining that, on any given iteration, `k` and `kold` have distinct values.

Periodic variables may follow more complex sequences as well, if arithmetic operations occur in the pattern [13]:

[3]An alternate form of the swap operation involves integer subtraction, e.g., `k = 3 - k`. In this form, `k` may be recognized as a geometric IV with base -1.

```
k = 1
kold = 2
loop
    A(i,j,k) = ...A(i,j,kold) ...
    ktemp = k + m
    k = kold + n
    kold = ktemp
endloop
```

In this example `k` is a *nonconstant* periodic variable. At its use as a subscript, it follows the sequence:

$$\begin{aligned}\texttt{k} &= \{1, 2+\texttt{n}, 1+\texttt{m}+\texttt{n}, 2+\texttt{m}+2\texttt{n}, 1+2\texttt{m}+2\texttt{n}, \ldots\}, \quad \langle 1, 2\rangle_{per(\texttt{n},\texttt{m})} \\ \texttt{kold} &= \{2, 1+\texttt{m}, 2+\texttt{m}+\texttt{n}, 1+2\texttt{m}+\texttt{n}, 2+2\texttt{m}+2\texttt{n}, \ldots\}, \quad \langle 2, 1\rangle_{per(\texttt{m},\texttt{n})}\end{aligned}$$

The "main" value of `k` alternates between 1 and 2, but at each time step the value increments by (alternately) `n` and `m`. The notation for nonconstant periodics is extended to $\langle x_1, \ldots, x_n\rangle_{per(y_1,\ldots,y_n)}$. For a period-2 variable like `k` in this example, the ith value of the sequence is x_1 when $i = 1$, and, when $i > 1$, the value is

$$x_m + \left\lceil \frac{i-1}{2} \right\rceil y_1 + \left\lceil \frac{i-2}{2} \right\rceil y_2$$

where $m = 2 - (i \bmod 2)$.

2.6 Monotonic Variables

Variables that are conditionally incremented or decremented cannot normally be represented as a function of the basic loop variable, but recognition of these forms provides useful information for data dependence solvers. A common example of a conditional induction variable occurs in code to "pack" the values in a vector into another vector, based on some test:

```
k=1
do i = 1, n
```

```
      if (A(i) > 0.0) then
         B(k) = A(i)
         k = k + 1
      endif
   enddo
```

Although the sequence of values for `k` cannot be represented in closed form (since the conditional assignment precludes it from being a linear IV), a compiler can profitably recognize that each store to array `B` refers to a distinct element.

Monotonic sequences arise when a variable is conditionally incremented by a known constant value. Four classes can be distinguished, depending on the constant: *monotonically increasing*, *monotonically decreasing*, *monotonically strictly increasing*, and *monotonically strictly decreasing*. In the previous example, `k` is monotonically increasing; within the body of the conditional, it is monotonically strictly increasing.

It may be possible for a compiler to represent the bounds of monotonic variables. For an increasing sequence, the minimum value of the sequence is the value on the first iteration, and the maximum value is the minimum value plus the maximum increment times the trip count of the loop.

2.7 Strengthening Sequences

The range of sequence expressions can be expressed as a lattice, ordered by set containment, as in Figure 2.1. In this lattice, $\top$ represents "no expression," and $-$ represents "all expressions." The class containing wrap-around variables is represented below all other classes except $-$, since in the limit, a wrap-around variable can be cascaded through an infinite set of values, and therefore represent any sequence of n values in a loop.

It is interesting to note that certain sequences will degenerate to simpler forms, allowing the compiler to strengthen the classification of a sequence variable. As implied earlier, the assignment `i = 3*i - 2*i + 1` is really just `i = i + 1` when simplified algebraically; the former expression is not a recognizable form, since the sequence variable occurs twice in the sequence expression, but the latter is easily determined to be a linear induction

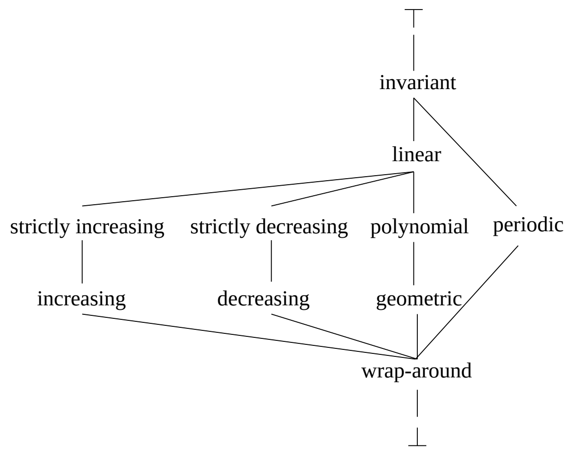


Figure 2.1: Sequence variable lattice

variable. Strengthening an expression will raise its classification in the lattice.

Two other examples of such strengthening are found in monotonic variables and wrap-around variables. The sequence variable `i` defined by

```
loop
    if (p) then
       i = i + c
    else
       i = i + c
    endif
endloop
```

might be classified as monotonic since it is conditionally incremented. Since both branches of the `if` increment `i` by the same amount, a compiler may recognize that `i` is actually a linear IV. If it were to be determined that `c` is equal to zero, `i` would degenerate to a loop-invariant variable.

If the initial value of a wrap-around variable fits the pattern of the subsequent sequence, the compiler can strengthen the classification to a variable of that sequence type. For example, a wrap-around variable determined to have the pattern $\langle 2, 2h+2\rangle_{wrap}$ can be represented as a linear IV, $2h+2$.

Chapter 3

Detecting Cycles in SSA Graphs

There are two steps involved in determining symbolic expressions for sequence variables. First, the sequence variables must be found; this is accomplished by partitioning a graph representation of the program in SSA form (a data-flow graph, as opposed to a control flow graph) into strongly connected components. Each strongly connected component (SCC) corresponds to either a loop-invariant value (which may be viewed as a trivial sequence), a proper sequence form (one of the types described in the previous chapter), or an unknown sequence form. Second, the nodes in each component (sequence) are assigned symbolic expressions describing the sequence form, such as the closed forms in terms of h seen in the previous chapter.

The sequence type and expression for a given component are dependent on the sequence types and expressions of those variables they use. This leads to an approach that combines these two steps: any given component will first "demand" the classification of any components it requires for its own classification. This demand process is performed by using Tarjan's well-known algorithm for detecting SCCs in directed graphs [21]. Tarjan's algorithm has the property that SCCs are visited only after visiting all "descendant" components in the graph; intuitively, a DAG of components is formed and processed in postorder during a depth-first traversal. Thus, we are able to classify and assign expressions in a demand-driven fashion, processing each sequence in the program exactly once.

In this chapter, we describe our intermediate representation and our SSA form, the key SSA graph concepts, and our implementation of Tarjan's algorithm for detecting components in the proper order. The algorithms for classifying these components and assigning expressions will be presented in the following chapter.

3.1 Intermediate Representation

Control flow is represented in the usual way as a graph, $G_{CFG} = \langle V, E, Entry, Exit \rangle$ where V is a set of nodes representing the basic blocks in the program; E is a set of edges representing sequential control flow; and *Entry* and *Exit* are distinguished nodes representing the unique entry and exit points in the program. All nodes are assumed to be reachable from *Entry*.

Natural loops are detected using dominator information [1]; CFG nodes within loop bodies are associated with their corresponding loops. The *header node* of a loop is defined as that node which dominates all nodes in the loop body.[1] It is convenient to insert nodes into the CFG such that the header has exactly two predecessors, one from within the loop and one from without. To provide this property, the compiler distinguishes two sets of edges entering the header: the loop back edges and the loop entry edges. A *preheader* node is inserted such that it provides a target for the loop entry edges. A *postbody* node is inserted such that it provides a target for all loop back edges. Two edges are then inserted from the preheader and postbody to the header. For each edge exiting a loop, a *postexit* node is inserted outside of the loop, between the source of the exit edge (within the loop body) and the target (outside the loop). The postexit node is used for last-value expressions for variables assigned within the body of a loop.

The *tripcount* of a loop l is defined as the number of times control flow passes through the loop header node; it is equal to the value of $h_l + 1$ after the loop has completed, where h is the basic loop counter described previously.

Each basic block in the program is represented by a list of tuples of the form $\langle op, left, right \rangle$, where *op* is an operation code, and *left* and *right* are pointers to the tuples serving as operands for the operation. Operations include fetch, store, addition, subtraction, etc. Depending on the type of the operator, *left* or *right* may be unused, e.g., unary minus. Only scalar, integer-valued operations are considered here. In particular, floating-point operations and indexed or indirect fetches and stores are not considered.

[1]We will consider only reducible loops. Since natural loops are reducible, each loop will have a distinct header.

```
i = 1
loop
    if (i > n) break
    i = i + c
    if (p) then
       k = i + 2
    endif
endloop
j = i + k
```

Figure 3.1: Sample program

```
$i_0$ = 1
loop
    $i_1$ = $\phi$($i_0$, $i_2$)
    $k_1$ = $\phi$($k_0$, $k_3$)
    if ($i_1$ > $n_0$) break
    $i_2$ = $i_1$ + $c_0$
    if ($p_0$) then
       $k_2$ = $i_2$ + 2
    endif
    $k_3$ = $\phi$($k_1$, $k_2$)
endloop
$j_0$ = $i_1$ + $k_1$
```

Figure 3.2: Sample program in SSA form

Subscript operators are considered only in the context of serving as indexing functions for other memory operations.

3.2 Demand-Driven SSA Form

Our approach is based on a form derived from the SSA form [7, 8]. SSA form is essentially a sparse representation of "def-use" chains [1], produced by renaming variables such that every use of a variable has exactly one corresponding reaching definition. Where distinct definitions of a variable merge at confluence points in the CFG, operators called *ϕ-functions* are introduced to merge each of the reaching definitions at that point. The ϕ-function in turn serves as a definition point. Figures 3.1 and 3.2 show a loop before and after conversion to SSA form. As is customary, we represent unique definitions of a variable by subscripting.

Rather than the traditional def-use chains, demand-driven SSA form uses factored use-def (FUD) chains [19]. In our implementation, fetch operators and ϕ-functions have pointers to the corresponding definition for the variable in question; note that no "renaming" of variables actually occurs, as is sometimes implied in the literature. By providing each use with a link to the reaching definition, unique definitions are provided with little additional overhead in representation. In the tuple representation of operators, fetch operators have an additional field, the *ssalink*, and each ϕ-function has an *ssalink* for each reaching definition. (Note that ϕ-functions and fetch operators do not use their *left* and *right* links.)

Merge operators that occur at loop headers are distinguished from those occurring as a result of forward branching. Within loop headers, merges of multiple reaching definitions of a variable are handled by μ-functions, akin to the μ-functions introduced by Ballance et al in their Gated Single Assignment form, but without predicate information [5]; all other merges are handled by ϕ-functions as normal. Thus, the semantics of the μ are essentially the same as the ϕ, but with two convenient differences. First, because preheader and postbody nodes are added to each loop, the arity of a μ-function is always two. Second, of the two reaching definitions at the μ, one will always be from within the body of the loop (the *internal ssalink*), and the other will always be from without (the *external ssalink*).

The *SSA graph* is an abstraction representing the operations within the SSA form of the program.[2] We define $G_{SSA} = \langle V, E \rangle$, where V corresponds to the set of operation tuples, and E corresponds to the set of *left*, *right*, and *ssalink* pointers. In Figure 3.3 the CFG and SSA graphs are shown for the program in Figure 3.2. The *ssalinks* are distinguished from the *left* and *right* links by dashed lines, and for clarity only the *ssalinks* for variable `i` are shown. Note that the ϕ-functions in the loop header position are represented by μ-functions.

The use-def chain form, as opposed to the traditional def-use chain form, has the advantage that the reaching definition at a given use is found by following the links from the use's node backward, against the direction of data flow. For the propagation-based

[2]Unless stated otherwise, for the remainder of this thesis "SSA" will refer to our demand-driven form.

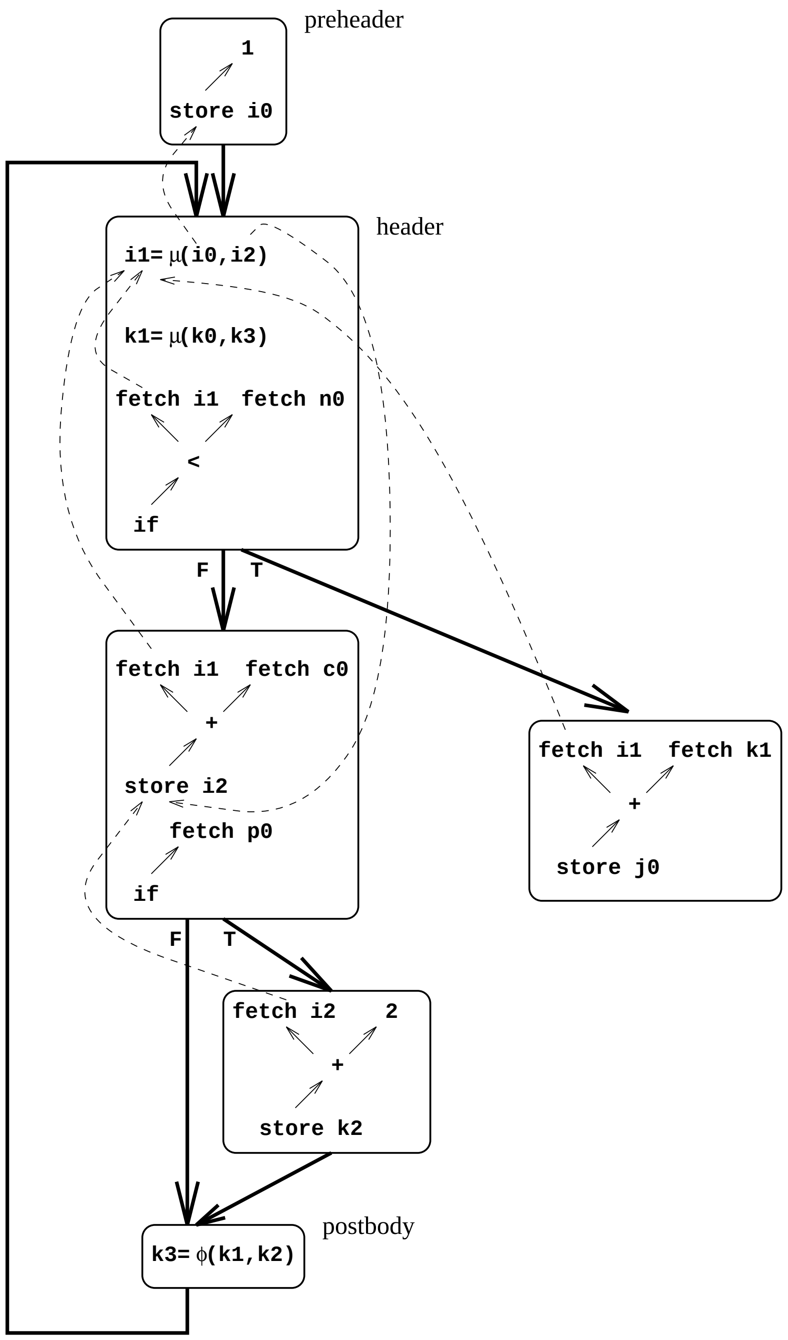


Figure 3.3: Demand-driven SSA graph of sample program.

problems we consider here, this is exactly the property desired: on a recursive traversal of the SSA graph, each use "demands" the value of the earlier definition. At the operation level, each operand is evaluated by first demanding the values of its operands, via the *left* and *right* links. These demand-driven SSA graphs provide the framework for detecting sequence variables.

The algorithms described here are demand-driven in their operations on the SSA graph. They are not *lazy* in their determination of sequence forms, however: such a system might, for example, classify a variable *only when it is required* for dependence analysis. The system described here attempts to classify all scalar, integer values, but could easily support true lazy evaluation.

3.3 Cycles in SSA Graphs

Consider the SSA representation of `i` in Figures 3.2 and 3.3. Starting at the μ defining $\mathtt{i}_1$, the external *ssalink* defines the value of $\mathtt{i}_1$ on the first iteration of the loop. On subsequent iterations the value of $\mathtt{i}_1$ is defined by the internal *ssalink* to the store defining $\mathtt{i}_2$ at the statement $\mathtt{i}_2 = \mathtt{i}_1 + \mathtt{c}_0$. The right-hand side of this update statement fetches the current value of `i`, defined by the *ssalink* from the fetch of `i` to the μ. These edges in the SSA graph define a cycle, representing the "flow" of `i` around the loop: the variable `i` is a sequence variable, since it is defined as a function of itself on a previous iteration. This observation leads to the two key properties used in our technique.

Property 1 *The nodes and edges of an SSA graph corresponding to the definition of a sequence variable form a nontrivial strongly connected component.*

The definition of a sequence variable will clearly make use of other operations, e.g., the fetch of $\mathtt{c}_0$, but these are not included in the SCC. Only the operations in the SCC have the behavior of the sequence under consideration.

Property 2 *A nontrivial SCC in the SSA graph will contain at least one μ-function.*[3]

[3]This only holds for reducible control flow graphs. SCCs not containing a μ-function should be handled separately.

These properties of SSA graphs are easily understood. Consider a statement in a loop defining a sequence of i. Assume there is only one definition of i within the loop and that i occurs only once on the right-hand side, as in the fragment:

```
      i = ...
      loop
$s$:      i = ...i...
      endloop
```

As statement s creates a definition of i within the loop, there will be a μ-function in the loop header to merge the definitions from outside and inside the loop. The internal *ssalink* of the μ will reach the last definition of i in the body, at s. There is a path from the store to the fetch on the right-hand side. The fetch, in turn, has an *ssalink* to the previous reaching definition, the μ-function. Thus, in our SSA form

```
      $i_0$ = ...
      loop
          $i_1$ = $\mu(i_0, i_2)$
$s$:      $i_2$ = ...$i_1$...
      endloop
```

there is a cycle in the corresponding SSA graph: the internal *ssalink* of the μ-function is linked to the store of i_2 which is linked to the expression on its right-hand side, which in turn contains a link from the use of i_1 to its corresponding reaching definition, the μ-function defining i_1. For more complex sequences, such as those with intermediate stores and fetches

```
loop
    j = ...i...
    ...
    i = j
endloop
```

these properties still hold, as a result of the insertion of *ssalinks* between the intermediate stores and fetches.

By detecting SCCs, a compiler can examine the operations within the component to determine the nature of the sequence. In many cases, as we will show, the variable(s) used in the component can then be classified as functions of the tripcount of the loop. In the example from Figure 3.2, `i` is a linear induction variable whose initial value is 1 and whose value on each iteration increases by `c`, a loop-invariant constant. We can define the sequence expression for `i` as a linear function of the basic loop counter, h: variable $\mathtt{i}_2$ is equal to $\mathtt{c}_0 h + 1$. Not all SCCs correspond to known sequences; a criterion will be shown below.

3.4 Tarjan's Algorithm

Associated with each node (operation tuple), t, in the SSA graph the following fields are used:

- *t.Type*: the type of operation (**fetch**, **add**, etc.).
- *t.Lowlink*: used (with global variable *Number*) within the implementation of Tarjan's algorithm.
- *t.Status*: one of {**notyet**, **onstack**, **done**}, denoting whether node t has been visited.
- *t.Loop*: (innermost) loop containing node t, or $\emptyset$ if node t is outside any loop.
- *t.HasLeft*, *t.HasRight*, *t.HasSSA*: *true* iff t has *left*, *right*, *ssalink* fields.
- *t.Left*, *t.Right*, *t.SSA*: the node corresponding to the *left*, *right*, *ssalink* fields of t.

To maintain the stack of nodes currently being searched, a global variable and two functions are needed:

- *StackTop*: node on top of stack.
- *Push_Stack(t)*: puts t on the stack.
- *Pop_Stack()*: pops the stack, returning the top node.

Each loop, l, has one field, $l.Ops$, which contains the list of operations within each basic block in the body of the loop. To determine the nesting of one loop with respect to another, a function is used:

- $Contains(l_1, l_2)$: returns *true* iff (loop l_2 is contained within loop l_1) or ($l_1 = \emptyset \wedge l_2 \neq \emptyset$)

Algorithm 1 (SSA Graph Component Detection) *Given l, a loop in the program, this algorithm finds the SCCs in the SSA graph of the operations in l.*

The algorithm uses procedure **Find_Components**, *called for each loop in the program in innerloop-first order, to visit each node in a loop. The procedure* **Visit_Node** *first visits all of a node's SSA graph successors within the loop, and then processes that node. Depending on whether the node is a trivial component or the root of a nontrivial component, one of two classification procedures is used. The procedures used —* **Find_Components**, **Visit_Node**, *and* **Visit_Descendent** *— are shown in Figures 3.4, 3.6, and 3.5.*

Note that SSA graph edges that cross loop boundaries are carefully respected by this algorithm. If the SSA successor node is in a loop that is outside of the loop currently being processed, the successor node is not recursively visited in **Visit_Descendent**. This allows the compiler to treat a value from *outside* the current loop as *invariant*, and is critical to the characterization of loop-based sequences. In the case where the SSA successor node is in a loop that is *inside* the current loop, there will be an η-function *gating* the exit value from the inner loop, preventing the algorithm from revisiting those nodes in the inner loop. (The treatment of η-functions and exit values is explained in Chapter 5.) Strictly speaking, we can say that, because loop boundaries are not crossed, the loops may be visited independently in any order. As a practical matter, however, inner loops are visited first.

Given that the goal is to provide expressions for nodes that may be candidates for reduction in strength or that may provide information needed for data dependence, some optimization to the above algorithm is possible. If the goal is to characterize subscript expressions, the **Find_Components** procedure could be specialized in the loop at line 6 to visit only subscript operators; the search procedure would demand sequence expressions

```
procedure Find_Components($l$)
1: $Number = 0$
2: $StackTop = \emptyset$
3: for $t \in l.Ops$ do
4:     $t.Status =$ notyet
5: endfor
6: for $t \in l.Ops$ do
7:     if $t.Status =$ notyet then
8:         Visit_Node($t$, $l$)
9:     endif
10: endfor
endprocedure
```

Figure 3.4: Procedure **Find_Components**

```
procedure Visit_Descendent($t$, $l$)
1: if Contains($t.Loop$, $l$) return $Number$
2: if $t.Status =$ notyet then
3:     Visit_Node($t$, $l$)
4:     return $t.Lowlink$
5: elseif $t.Status =$ onstack then
6:     return $t.Lowlink$
7: endif
8: /* $t.Status =$ done */
9: return $Number$
endprocedure
```

Figure 3.5: Procedure **Visit_Descendent**

```
procedure Visit_Node($t$, $l$)
1:  $t.Status$ = onstack
2:  $low = this = Number = Number + 1$
3:  $t.Lowlink = low$
4:  Push_Stack($t$)
5:  if $t.HasLeft$ then
6:      $low = \min(low,$ Visit_Descendent($t.Left$,$l$))
7:  endif
8:  if $t.HasRight$ then
9:      $low = \min(low,$ Visit_Descendent($t.Right$,$l$))
10: endif
11: if $t.HasSSA$ then
12:     $low = \min(low,$ Visit_Descendent($t.SSA$,$l$))
13: endif
14: $t.Lowlink = low$
15: if $this \neq low$ return
16: if $StackTop = t \wedge t.Type \neq \mu$ then
17:     Classify_Trivial($t$, $l$)
18:     Pop_Stack()
19:     $t.Status$ = done
20: else
21:     $Component = \emptyset$
22:     repeat
23:             $StackTop$ = Pop_Stack()
24:             $StackTop.Status$ = done
25:             $Component = Component \cup StackTop$
26:     until $StackTop = t$
27:     Classify_Sequence($Component$, $l$)
28: endif
endprocedure
```

Figure 3.6: Procedure **Visit_Node**

for precisely those operations in the SSA graph needed to classify the subscript, and time would be saved by not attempting to visit unneeded nodes, e.g., floating-point operations.

Chapter 4

Classifying Sequence Variables

In the previous chapter an algorithm for partitioning the SSA graph into components was presented. The next step is to determine the type of sequence of each given component (if any). This will be performed by the procedures **Classify_Trivial** and **Classify_Sequence**.

We will first consider the case of trivial components. A trivial component in the graph is assigned an expression according to the type of operation it represents and its operands, e.g., a subtraction operator whose operands have expressions `m+6` and `n-4` will be assigned the expression `m-n+10`. We introduce a simple lattice framework and a demand-driven propagation mechanism to perform this.

We will next consider the case of nontrivial components and trivial components consisting of only a μ-function. First, the class of sequence must be determined — linear, monotonic, etc. This is based on the types of the operations (and their operands) in the component. Second, expressions for the nodes in the components are created based on the type of sequence and the expressions of the descendent nodes.

4.1 Demand-Driven Propagation

The framework used here is based loosely on the idea used in demand-driven constant propagation [20]. In the constant propagation algorithms, constant integers may be assigned to variables at their definition points, and these values are propagated to their uses, using simple constant folding to combine variables known to be constant wherever possible. The propagation is performed "against" the data flow path, i.e., along the *left*, *right*, and *ssalink* edges: the value of a variable being fetched is determined by "demanding" the value at its definition point; the value at an addition operation is determined by demanding the values of the addition's operands, and so forth.

The lattice used in the constant propagation framework contains only the class of constants, representing the set of integers. To perform strength reduction, a compiler should be able to represent unknown variables as symbolic quantities and linear functions of the loop index variable as symbolic expressions in h, the basic loop counter. For the classification of the full range of sequences under consideration here, the set of symbolic expressions must be still larger. The lattice of classes for these expressions was shown in Figure 2.1. The top element in the lattice ($\top$) represents "no expression"; this is the value assigned initially to all variables. The bottom element ($-$) represents "unknown" and is assigned when no sequence expression can be determined for a variable. The class *invariants* contains the set of integers ($\{\ldots, -1, 0, 1, \ldots\}$), a set of invariant symbols, and combinations of these, e.g., expressions such `k+10` or `n*i/2`, where `k`, `n`, and `i` are invariant with respect to some loop. All the classes of sequences described in Chapter 2 are also represented: linear, polynomial, and geometric IVs, as well as wrap-around, periodic, and monotonic sequence variables.

In determining sequence forms, variables are assigned sequence classes at their definition points. Each variable being defined inherits the sequence class of the rhs of the definition; each operation on the rhs is assigned a class based on its operands, e.g., variable fetch operations inherit the class of the variable being fetched.

To represent the type of sequence being described, i.e., a member of the sequence class, symbolic expressions are associated with each variable definition in a similar manner. The compiler must be able to represent these members and perform algebraic operations upon them. For the purposes of sequence expressions, such a system need only be extended to accommodate operations on the complex sequence forms. For the sequences that have closed forms in terms of h (linear, polynomial, and geometric IVs), this is accomplished by explicitly representing the closed-form equations, similar to the forms shown in Chapter 2. For the wrap-around and periodic sequence classes, some means of representing the constituent parts of the expressions describing them is needed. For monotonic forms, objects representing the type of monotonicity are required.

Some freedom is allowed in the complexity of the range of a symbolic evaluation system. While the ability to perform certain operations on certain types will be required, e.g., addition of integral invariants, a system need not support the more complex cases, e.g., division of a periodic sequence by a geometric sequence. In these latter cases, a compiler may choose to evaluate the expression as $\emptyset$, with lattice value $-$.

A set of transfer functions T_{op} is used to assign each operation in the program a sequence expression. Associated with each operation n are two values: *Class(n)*, the lattice value, and *Expr(n)*, the appropriate expression of that class. Based on the type of operation, a lattice transfer function is used to determine *Class* while the compiler's algebra of types is used to determine *Expr*. The definition of each transfer function depends in the natural way on the arity and type of operation. In general, for some operation n of operator type *op* with two operands n_l and n_r,

$$\begin{aligned} Class(n) &= T_{op}(Class(n_l), Class(n_r)) \\ Expr(n) &= f_{op}(Expr(n_l), Expr(n_r)). \end{aligned}$$

Algorithm 2 (Propagation for Trivial Strongly Connected Components) *Given t, an operation, and l, its loop, this algorithm determines the lattice value and expression for t, t.Class and t.Expr. The procedure is shown in Figure 4.1.*

The procedure shows the cases for addition, fetches, and ϕ-functions. Other cases are handled similarly; μ-functions will not be processed here, since they are treated as sequence forms within nontrivial SCCs.

The operator T_+ represents the transfer function for addition on the lattice, as shown in Table 4.1. The operator f_+ represents the compiler's algebra for addition on the symbolic forms. Note that the table includes $\top$ as a possible operand, but such an operation will never occur. By the demand-driven nature of the algorithm, expressions will have been classified (to something other than $\top$) before they will be used in an operation.

The algebra T_+ presented for addition is implementation dependent in two ways. First, addition corresponds properly to the greatest lower bound in the lattice in Figure 2.1, but in practice operations may simply go to $-$ rather than become complex wrap-around

```
procedure Classify_Trivial($t$, $l$)
case $t.Type$
    $+$:     $t.Class = T_+(t.Left.Class, t.Right.Class)$
             $t.Expr = f_+(t.Left.Expr, t.Right.Expr)$
    fetch:   $t.Class = t.SSA.Class$
             $t.Expr = t.SSA.Expr$
    $\phi$:  if $t.SSA_1.Expr = t.SSA_2.Expr = \ldots = t.SSA_n.Expr$ then
                 $t.Class = t.SSA_1.Class$
                 $t.Expr = t.SSA_1.Expr$
             else
                 $t.Class = -$
                 $t.Expr = \emptyset$
             endif
    ...
endcase
endprocedure
```

Figure 4.1: Procedure **Classify_Trivial**

variables. Second, the lattice value selected by T_+ may be optimistic and later adjusted by f_+. This happens, for example, in the addition of linear IVs and monotonically increasing sequences: the result is monotonically increasing only if the linear sequence is known to have a positive increment.

4.2 Classifying Sequences

The class of a component depends on the mix of operations within it: the number and type of μ-functions, ϕ-functions, and arithmetic operations. This determination is performed by the **Classify_Sequence** procedure. Using the counts of the types of operations that make up the component, the class of the component is determined. All nodes in the component are assigned this lattice value (sequence class).

Table 4.2 shows the conditions used by **Classify_Sequence**. These conditions represent an approximation: types may be simplified, as mentioned previously, and there are some stipulations about the nature of operands, e.g., that a component for linear IVs may include subtraction provided the right operand is not the sequence variable. Further, notice that the criteria for the induction variables (linear, polynomial, and geometric) are identical. This is because the distinction comes not from the operations in the component

T_+	$\top$	*inv*	*wrap*	*lin*	*poly*	*geom*	*per*	*inc*	*sinc*	*dec*	*sdec*	$-$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$-$
inv	$\top$	*inv*	*wrap*	*lin*	*poly*	*geom*	*per*	*inc*	*sinc*	*dec*	*sdec*	$-$
wrap	$\top$	*wrap*	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
lin	$\top$	*lin*	$-$	*lin*	*poly*	*geom*	$-$	$-$	$-$	$-$	$-$	$-$
poly	$\top$	*poly*	$-$	*poly*	*poly*	*geom*	$-$	$-$	$-$	$-$	$-$	$-$
geom	$\top$	*geom*	$-$	*geom*	*geom*	*geom*	$-$	$-$	$-$	$-$	$-$	$-$
per	$\top$	*per*	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
inc	$\top$	*inc*	$-$	$-$	$-$	$-$	$-$	*inc*	*inc*	$-$	$-$	$-$
sinc	$\top$	*sinc*	$-$	$-$	$-$	$-$	$-$	*inc*	*sinc*	$-$	$-$	$-$
dec	$\top$	*dec*	$-$	$-$	$-$	$-$	$-$	$-$	$-$	*dec*	*dec*	$-$
sdec	$\top$	*sdec*	$-$	$-$	$-$	$-$	$-$	$-$	$-$	*dec*	*sdec*	$-$
$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$

Table 4.1: Example Algebra for Addition
note: *inv*=invariant, *wr*=wrap-around, *lin*=linear, *poly*=polynomial, *geom*=geometric, *per*=periodic, *(s)inc/dec*=(strictly) increasing/decreasing

sequence class	μ	ϕ	*arith*
linear	1	0	> 0
polynomial	1	0	> 0
geometric	1	0	> 0
wrap-around	1	0	0
constant periodic	> 1	0	0
nonconstant periodic	> 1	0	> 0
monotonic	1	> 0	≥ 0
$-$	*otherwise*		

Table 4.2: Classification of SCCs Based on Frequency of Operations

but by examination of their operands. The process of determining the component's class must actually examine operands as well as operations.

In the remainder of this chapter, we describe the sequence classes with respect to their criteria and the "solvers" used for assigning expressions. Other implementations may choose to use different solvers, or a combination of solvers. One alternative possibility is to use interpolation to combine the IV solvers into one general algorithm. The advantage to the technique presented here is that the common cases (linear IVs) are detected and classified quickly with only one pass through the component.

4.2.1 Linear Induction Variables

For linear IVs, the operations within the component may consist of fetches, stores, and additions or subtractions of loop-invariant values or other linear variables. There must be exactly one μ-function and no ϕ-functions in the component. Note that the SCC of the SSA graph defining the sequence will actually be a simple cycle, since the induction variable may only appear once on the right-hand side of the expression.

Solver. Our technique is to step through each of the operations in the cycle, in lexical order (opposite data flow) starting at the μ, and assign each operation an expression corresponding to its value at that point. Initially, the μ assumes the expression from its external *ssalink*. When all the operations in the cycle have been considered, their expressions represent the initial values at those points. The total increment or decrement per iteration to the induction variable will be represented by the value assigned the internal *ssalink* of the μ.

Example. Consider the variable `i` in the following loop, shown in SSA form with sequence values:

```
i₀ = 1
loop
    i₁ = μ(i₀, i₃)     ; {1, 6, 11, ...}
    i₂ = i₁ + 2        ; {3, 8, 13, ...}
    ...
    i₃ = i₂ + 3        ; {6, 11, 16, ...}
```

```
      ...
      l = t + 4*i_3      ; {t+24, t+44, t+64, ...}
   endloop
```

The search procedure finds the component for this variable and classifies it as a linear IV based on the operations within the cycle. The μ has an initial value of 1 inherited from outside the loop. The first assignment assumes the value 3 (from $1 + 2$), and the second assignment assumes the value 6 ($3 + 3$). The total increment in the cycle is 5. The sequence expression for the μ-function is therefore $5h + 1$. The first and second stores of i have expressions of $5h + 3$ ($5h + 1$ from the μ, plus 2) and $5h + 6$ ($5h + 3$ plus 3).

Variable `l` is not in a nontrivial SCC; its expression is formed from the propagation of $\mathtt{i}_3$, the multiplication by four, and the addition of `t`: $20h + \mathtt{t} + 24$.

4.2.2 Polynomial Induction Variables

For polynomial IVs, the operations within the component may consist of fetches, stores, and additions or subtractions of loop invariants, linear IVs, or other polynomial IVs. There must be exactly one μ-function and no ϕ-functions.

Solver. Our technique for discovering the sequence expression of a polynomial IV is to solve a matrix equation to determine the coefficients of an expression fitting the sequence, as follows. For a degree-d polynomial (recall Section 2.2), let $\vec{x}$ be the first $d + 1$ values of the induction variable, which the compiler can determine by symbolically executing the statements making up the SCC for $d + 1$ iterations. Let A be a matrix such that $A_{ij} = i^j, 0 \leq i, j \leq d$. Let $\vec{s}$ be the $d + 1$ unknown coefficients of the induction expression. Letting $A\vec{s} = \vec{x}$ the compiler can discover the desired coefficients by computing $\vec{s}$, which is $A^{-1}\vec{x}$. The expression for the induction variable can then be represented by $\sum_{k=0}^{d} s_k h^k$.

Example. Consider `k` in the loop:

```
   i = 0
   j = k = 1
   loop
```

```
        i = i + 1
        j = j + i
        k = k + j + 1
    endloop
```

Variable k is the sum of a degree-2 polynomial IV, a linear IV, and a constant; therefore $d = 3$:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \end{bmatrix}$$

and

$$\vec{x} = [4 \quad 9 \quad 17 \quad 29]^T$$

yielding

$$\begin{aligned} \vec{s} &= A^{-1}\vec{x} \\ &= [4 \quad \tfrac{23}{6} \quad 1 \quad \tfrac{1}{6}]^T . \end{aligned}$$

Thus, the sequence expression for k is $(1/6)h_3^3 + h_3^2 + (23/6)h_3 + 4$.

Since d will be known at compile-time, the number of unknowns is fixed. If the values of x_k are integers, the resulting values of s_k will be rational. In general, however, x_k may be a symbolic expression (possibly with rational coefficients), so a compiler may need to perform symbolic matrix operations.

Since matrix A is constant, A^{-1} may be precomputed by the compiler up to any desired degree or created on demand.

4.2.3 Geometric Induction Variables

For geometric IVs, the operations within the component must contain a multiplication by a loop-invariant value; this multiplicative factor defines the base of the geometric term in the sequence expression. The other operations in the component may consist of fetches, stores, and additions or subtractions of loop-invariant values, linear IVs, or polynomial

IVs. There must be exactly one μ-function and no ϕ-functions. The sequence variable may occur more than once (after simplification) on the rhs in a geometric IV.

Solver. The method we use to solve geometric components is the same as that for polynomials, but with the addition of a geometric term. The matrix A is now $d+2$ by $d+2$, such that

$$A_{ij} = \begin{cases} i^j, & 0 \leq i \leq d+1, 0 \leq j \leq d \\ b^i, & 0 \leq i \leq d+1, j = d+1 \end{cases}$$

where b is the base of the geometric term. This formulation accounts for induction expressions that have both geometric and polynomial terms. The base corresponds to the constant by which the induction variable is multiplied. If the variable occurs only once on the right-hand side this is trivially found by examining the operations and operands within the SCC.

If the sequence variable occurs more than once on the rhs, e.g., `g = 5*g - (2 + g)`, the compiler may determine this multiplicative constant by symbolically evaluating the expression, collecting factors of the induction variable in the expression tree of the right-hand side of the assignment. Starting at the μ-function, in reverse order each node in the SCC is visited and assigned a *factor value*. For a *fetch* in the SCC (which must be a fetch of the induction variable), a factor value of 1 is assigned. For addition operators, the factor value assigned is the sum of the factor values of the parents, using a value of zero if the parent is not in the SCC. For multiplication operations, we similarly assign a factor value equal to the product of the parents' values: one parent will be in the component, and the other must be a constant, which is assumed to have a factor value equal to the constant. The factor value assigned the μ-function is the factor value of the internal *ssalink* of the μ in the SCC. This factor is then used as the base in the geometric term.

The list of nodes in the SCC is not necessarily a simple cycle, as in the previous forms, since the sequence variable may occur more than once on the rhs. To set the factor value of the μ in only one pass through the component, each node in the component can be assigned a factor value only after each of its immediate successors in the component has been assigned a factor value. The μ-function is visited last.

This solver assumes implicitly there will be only one geometric term in the resulting

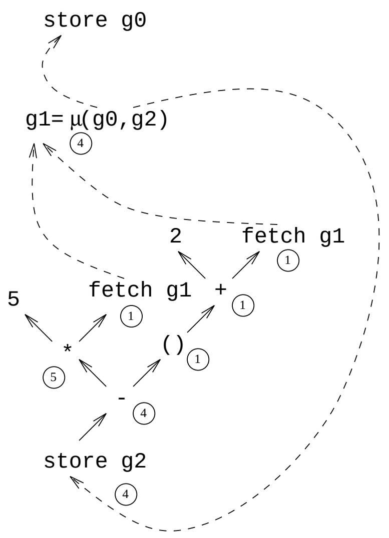


Figure 4.2: SSA graph with factor values

sequence expression. To handle the general case, the technique may be extended in a straightforward way.

Example. In Figure 4.2 we show the SSA graph for the loop

```
g₀ = 1
loop
    g₁ = μ(g₀, g₂)
    g₂ = 5 * g₁ - (2 + g₁)     ; {2, 6, 22, 86, ...}
endloop
```

The factor values assigned each node are shown circled next to the operations. The final value produced is 4, which is correct since `5*g-(2+g)` $\equiv$ `4*g-2`. The determination of the actual sequence expression for `g` follows similarly to the example provided for polynomial IVs, producing the expression $(4/3)4^h + 2/3$.

4.2.4 Wrap-Around Variables

In the SSA graph, a wrap-around variable will occur as a μ-function in an SCC by itself. Wrap-around variables are the only sequence forms whose component is a single node.

Solver. If the internal *ssalink* of the μ has a value of some expression p, then the variable defined by the μ is a first-order wrap-around variable of type p. If p is itself a wrap-around variable of order n, then the variable defined by the μ is a wrap-around variable of order $n+1$ and type p.

Example. Here `j` is used as a wrap-around variable:

```
j = n
i = 0
loop
    ...= ...j ...
    j = i
    i = i + 1
endloop
```

In SSA form, the only nontrivial SCC contains the operations in statements s_1 and s_2, defining the linear IV `i`:

```
j$_0$ = n
i$_0$ = 0
loop
   j$_1$ = $\mu$(j$_0$, j$_2$)
$s_1$: i$_1$ = $\mu$(i$_0$, i$_2$)
   ...= ...j$_1$ ...
   j$_2$ = i$_1$
$s_2$: i$_2$ = i$_1$ + 1
endloop
```

Variable j_1 is a wrap-around variable, $\langle \mathtt{n}, h+1\rangle_{wrap}$: the external *ssalink* of the μ has expression `n`, and the internal *ssalink* is linear (j_2 is linear, inherited from the fetch of i_1).

4.2.5 Periodic Sequences

An SCC with more than one μ may correspond to a periodic variable. The number of μ-functions defines the period of the sequence. If the component contains only fetches and stores, the sequence will be a *constant periodic*. If the component contains arithmetic operators, the component may define a *nonconstant periodic* variable.

Solver. For constant periodic variables, the values making up the sequence may be determined simply by examining the values of the external *ssalinks* of the μ-functions.

For nonconstant periodic forms, the compiler must symbolically execute the statements in the component for one period. The arithmetic operations in the component will define the nature of the nonconstant sequence. If the arithmetic operations are linear in nature, e.g., addition or subtraction of loop invariants, this is trivial: these invariants will define the increments or decrements to the sequence.

Example. Consider the nonconstant periodic variable defined by

```
jo = 1
j = 10
loop
    jt = jo + 1
    jo = j
    j = jt
endloop
```

Within the loop, variable j follows the sequence $\{2, 11, 3, 12, 4, 13, \ldots\}$, represented as $\langle 2, 11\rangle_{per(1,0)}$.

4.2.6 Monotonic Variables

If the SCC contains a ϕ-function then the variable carried around the loop may be a *monotonic* IV. All four classes can be distinguished: *monotonically increasing*, *monotonically decreasing*, *monotonically strictly increasing*, and *monotonically strictly decreasing*.

Solver. When the compiler has determined that an SCC may be monotonic due to the presence of one or more ϕ-functions, the component must be analyzed to determine the nature of the monotonic behavior. Starting at the μ-function, the nodes in the component may be evaluated symbolically, in lexical order, and assigned symbolic values at each point. Regardless of the values at any particular point, if the compiler can determine an ordering between the value reaching the μ from within the loop and the value from outside the loop, a monotonic classification may be assigned.

Example. Consider the pack example from Section 2.6, expressed in SSA form for variable `k`:

```
k$_0$ = 1
do i = 1, n
     k$_1$ = $\mu$(k$_0$, k$_3$)
     if (A(i) > 0.0) then
        B(k$_1$) = A(i)
        k$_2$ = k$_1$ + 1
     endif
     k$_3$ = $\phi$(k$_1$, k$_2$)
enddo
```

The compiler will first determine that k_2 is strictly increasing relative to the initial value of k_1 by recognizing the addition of a *positive* constant. At the ϕ the compiler effectively computes the meet of both arguments: intuitively, the merge of a monotonically strictly increasing value ($k_1 + 1$) and a constant (k_1) is a monotonically increasing value. Thus, the value reaching the μ from the bottom of the loop is always greater than or equal to the initial value. Clearly, if the compiler could not determine the sign of the invariant added to k_1, no ordering may be determined, and no sequence expression may be assigned.

Chapter 5

Extensions for Loop Structures

Several researchers studying parallelizing compilers have pointed out specific types of *nested* induction variable forms that, while relatively infrequent, can lead to significant parallelism if the subscript expressions can be classified [9, 13]. A typical example is a triangular loop containing a polynomial induction variable, similar to the Fortran program and its SSA form shown in Figures 5.1 and 5.2.[1] Here `k` is a polynomial in the outer loop because it is linearly incremented in the inner loop, although this polynomial behavior is not apparent from the perspective of the outer loop. The effect of the inner loop on `k` must be summarized and exposed to the outer loop. This will be accomplished by the η-functions.

When considering a loop nest, we must consider any reference from outside of the loop as invariant because we wish to define induction variables in the context of the current loop only. The inner loop of the example program is:

```
L2: do
        j3 = μ(j1, j2)
        k2 = μ(k1, k3)
        j2 = doseq(1, i1)
        A(j2+1) = ...
        k3 = k2 + 1
```

[1]In our representation of Fortran `DO` loops, the loop index variable in the loop header position provides the reaching definition of the variable and not the μ-function. The index variable is defined by the `doseq` operator. The careful reader will note the μ-function for a loop index variable, although never used as a reaching definition, is technically a wrap-around variable.

```
L1: do i = 1, n
L2:      do j = 1, i
             A(j+1) = ...
             k = k + 1
         enddo
         ...= A(k)
    enddo
```

Figure 5.1: Triangular loop nest

```
        enddo
```

Tarjan's algorithm may be applied to the nodes of the SSA graph of the loop `L2` in any order. Assuming the only nontrivial SCC in the graph is identified first, `k` is classified as a linear IV. In particular, $\mathtt{k}_2 = h_2 + \mathtt{k}_1$ and $\mathtt{k}_3 = h_2 + \mathtt{k}_2 + 1$, where $\mathtt{k}_1$ is a loop-invariant symbolic expression. The loop index, $\mathtt{j}_2$, is not in an SCC, but a compiler may easily recognize this special case: `DO` loop indices are by definition linear induction variables — in this case $\mathtt{j}_2 = h_2 + 1$. The μ-function at $\mathtt{j}_3$, a wrap-around variable, has value $\mathtt{j}_1$ on the first iteration and takes the value of $\mathtt{j}_2$ on each subsequent iteration. The subscript expression for `A(j+1)` is a linear expression, since it contains a use of $\mathtt{j}_2$.

5.1 Tripcounts

The tripcount of Fortran `DO` loops can be determined at compile-time, although the result may be symbolic and need not be constant: by definition, the Fortran loop

```
    do i = init, last, step
        ...
    enddo
```

will have a tripcount equal to the maximum of zero and $(last - init + step)/step$. Thus, the tripcount of the example inner loop will be $\mathtt{max}(0, (\mathtt{i}_1 + 1 - 1)/1)$ which is simplified to $\mathtt{max}(0, \mathtt{i}_1)$ in the internal representation in the compiler.

Our studies show roughly three-quarters of the loops in scientific Fortran programs are (syntactically) `DO` loops. For more general loops our strategy to determine the tripcount is to examine the condition controlling whether or not the loop's CFG exit edge is taken. If

```
L1: do
        $i_2$ = $\mu$($i_0$, $i_1$)
        $j_1$ = $\mu$($j_0$, $j_3$)
        $k_1$ = $\mu$($k_0$, $k_4$)
        $i_1$ = doseq(1,$n_0$)
L2:     do
            $j_3$ = $\mu$($j_1$, $j_2$)
            $k_2$ = $\mu$($k_1$, $k_3$)
            $j_2$ = doseq(1, $i_1$)
            A($j_2$+1) = ...
            $k_3$ = $k_2$ + 1
        enddo
        $j_3$ = $\eta$($j_2$)
        $k_4$ = $\eta$($k_2$)
        ...= A($k_4$)
    enddo
    $i_3$ = $\eta$($i_1$)
    $j_5$ = $\eta$($j_1$)
    $k_5$ = $\eta$($k_1$)
```

Figure 5.2: Triangular loop nest in SSA form

an induction expression for that condition may be found, e.g., `i<n` may be treated as `i-n`, the number of times the loop executes may be determined. For loops with multiple exits, we do not currently attempt to determine a tripcount. Obviously, tripcounts for loops with simple exit conditions like `if (i<n)` where `i` is not an induction expression cannot be determined either. Our compiler does not currently consider complex exit expressions such as `if (i<n OR j<m)`.

5.2 Evaluation of Loop Invariants

The tripcount is an important factor for determining symbolic expressions for variables in loop structures. The effect of a loop on a given sequence variable in that loop can be summarized and propagated out of the loop, provided (1) a closed form for the sequence variable can be found and (2) the tripcount of the loop is known. It is advantageous to simplify such expressions as much as possible. In the preceding case, if the value of i_1 is a known constant, the `max` expression may be removed from the tripcount. Determining the constant value, however, requires some modifications to the algorithms presented earlier.

Normally, the search procedure will not cross loop boundaries, so fetches of loop-invariant values are represented symbolically within the inner loop, even if the variable is assigned a constant value immediately outside. One of the reasons for this restriction is that the reaching definition may turn out to be a function of the component currently being classified and thus result in an SCC which spans loop boundaries. Three approaches, varying in complexity and power, may be taken to resolve external fetches:

1. When it would be advantageous to resolve a particular symbol, a special search routine may be invoked to follow the *ssalink* of the fetch outside the current loop. If a simple expression, e.g., a constant, can be determined at the reaching definition, that value is returned. Otherwise, the expression assigned the fetch must remain a symbolic invariant. Note this method does not actually classify or assign sequence expressions to any operations outside the loop.

2. The search procedure may be modified such that when the graph successor of a node is visited, if it is outside the current loop the node will be nonetheless visited and classified *provided* it can be determined that the reaching definition will be "safe," i.e., will not require any solution from the current component. After classifying the external path, the value is propagated into the inner loop; the fetch is classified; and the algorithm continues with the current loop as normal.

3. The search procedure may be modified to handle loop boundaries. Tarjan's algorithm is applied to any node in any loop, and if the solution to a node requires information from outside its loop, the path outside the loop is searched after recording the current node number. If the resulting search attempts to visit a node currently on the stack with a node number less than the number recorded, there exists an interloop cycle, and the search procedure must backtrack.

The search procedure used within Nascent takes the second of these approaches, since the first is not sufficiently general, and the third requires substantial engineering that detracts from the simplicity of our approach. A simple heuristic is used to determine safety: the path is “safe” only if the path does not contain a μ or ϕ.[2] In practice, we have found the number of nodes searched to be less than three on average and the “success” rate between 60 and 70 percent.

In the current example of Figure 5.2, the value of $\mathtt{i}_1$ is known (by this second approach) to be a linear induction variable of the outer `DO` loop. Quick inspection reveals this outer `DO` loop has a lower limit of 1. In general, the compiler can assume that if the inner loop has executed, the value of $\mathtt{i}_1$ must be at least 1. The `max` operator may then be optimized away, setting the tripcount of the inner loop to be $\mathtt{i}_1$.

5.3 Exit Value Expressions and η-Functions

Once the tripcount and the nodes in the inner loop have been assigned expressions, the compiler moves to the next loop level. At this level, however, the SCCs for variables `j` and `k` span loop boundaries. The effects of the inner loop on these variables should be treated as fixed, however, and for this reason we *gate* the *exit value* of each variable assigned within the loop and restrict the walk of the SSA graph from passing through these gates. We are essentially collapsing the effect of the loop body into this exit value gate expression.

Ballance et al introduced η-functions in their Gated Single Assignment (GSA) form with loop predicate information to determine under what conditions the value being gated would be used [5]. Here we adopt the η-function but use it simply as a convenient placeholder for the exit value.

Where a use of a variable has as its reaching definition a definition inside an inner loop, an η-function is inserted in the postexit nodes of the loop. The η-function takes the place of the reaching definition inside the loop and has an *ssalink* to the reaching definition. Thus, the insertion of an η for some variable `x` is essentially an insertion of the assignment `x = x` at the loop's postexit node.

The goal is to use η-functions as placeholders for an expression representing the exit value of a variable assigned within a loop. The exit value will be either a constant or a symbolic expression, depending on the classification of the variable, its value prior to

[2]Or η, as introduced next.

the loop, and the ability of the compiler to determine the tripcount of the loop. If the variable of the η is an induction expression, its exit value is a function of the tripcount of the loop, and the compiler performs symbolic algebra to "solve" the induction expression for the tripcount. If the tripcount is unknown, the exit value is left undefined ($-$). It is important to note that the exit value of a variable will be an expression in terms of the current loop (or the outermost level), not from within the loop the η is gating.

As stated earlier, when performing the search of the SSA graph, if an η-function is encountered, the *ssalink* is not followed into the inner loop. Instead, at that point the exit value of the gated variable is derived and translated into its symbolic representation as operators in our intermediate form. The resulting tree is placed as the target of the η-function's *right* link. The search (via Tarjan's algorithm) then resumes, walking up the η's *right* link to this tree. By translating the expression from the symbolic representation to operators in the intermediate form, the search procedure (Tarjan's algorithm) can treat the η expression like any other operator in the program. The implementation would be considerably more complex if this translation were not performed, since the search procedure would have to incorporate symbolic expressions in the SCCs.

In the example, when the η for $\mathtt{j}_2$ is reached, the compiler must determine the exit value of $\mathtt{j}_2$ after the iterations of `L2`. The exit value for $\mathtt{j}_2$ is calculated as shown. We use the notation $a@b$ to mean "the value of expression a after b iterations": in particular, since h starts at zero, $h@n = n - 1$. We define tc_l to be the number of times a loop l is executed, the maximum value of h_l. Note that by the definition of `DO` loops, the header is always executed at least once, so the assignment to $\mathtt{j}_2$ is actually performed $tc_2 + 1$ times.

$$\begin{aligned}
\mathtt{j}_3 &= \mathtt{j}_2@(tc_2 + 1) \\
&= (1 + h_2)@(\mathtt{i}_1 + 1) \\
&= (1@(\mathtt{i}_1 + 1)) + (h_2@(\mathtt{i}_1 + 1)) \\
&= 1 + (h_2@(\mathtt{i}_1 + 1)) \\
&= \mathtt{i}_1 + 1
\end{aligned}$$

The exit value expression for $\mathtt{j}_2$ is then $\mathtt{i}_1 + 1$. This is translated into a fetch of $\mathtt{i}_1$ added to the integer constant 1 and connected to the η-function at $\mathtt{j}_3$. By a similar process, $\mathtt{k}_4$

```
L1: do
        $i_2$ = $\mu$($i_0$, $i_1$)
        $j_1$ = $\mu$($j_0$, $j_3$)
        $k_1$ = $\mu$($k_0$, $k_4$)
        $i_1$ = doseq(1,$n_0$)
        ...
        $j_3$ = $\eta$(1 + $i_1$)
        $k_4$ = $\eta$($k_1$ + $i_1$)
        ...= A($k_4$)
    enddo
```

Figure 5.3: Outer loop, with exit value expressions

is set as the exit value expression $\mathtt{k}_2@(tc_2 + 1)$, which reduces to $\mathtt{i}_1 + \mathtt{k}_1$.

The overall cost of η-functions is comparatively small in both time and space, and no new phases have been added to the compiler for η node insertion in the SSA graph. There is some cost in creating exit value expression trees, but this is dominated by the cost of the induction variable procedure as a whole. Note the η expressions are only required for variables that are live after the loop.

5.4 The Outer Loop

After the insertion of exit values, the outer loop has been transformed into Figure 5.3. The compiler searches the SSA graph for the body of this loop, classifying $\mathtt{i}_1$ as a linear IV, $\mathtt{i}_2$ as a wrap-around variable, and $\mathtt{j}_3$ as a derived linear IV.

The SCC containing variable k, however, contains an addition of a linear induction expression ($\mathtt{i}_1$). The compiler determines k to be a polynomial IV, and the compiler simulates the first 4 iterations of the loop to determine the values of the μ defining k1. Initially, $\mathtt{k}_1$ is $\mathtt{k}_1@0 = \mathtt{k}_0@0 = \mathtt{k}_0$. On the next iteration, $\mathtt{k}_1$ is $\mathtt{k}_1@0{+}\mathtt{i}_1@1$, which simplifies to $\mathtt{k}_0 + 1$. The next two iterations produce $\mathtt{k}_0 + 3$ and $\mathtt{k}_0 + 6$. These expressions are then used to solve a system of polynomial equations, producing the expression $h_1^2/2 + h_1/2 + \mathtt{k}_0$ for $\mathtt{k}_1$. The induction expression for the subscript of A is defined by $\mathtt{k}_4$, so the subscript is represented by the polynomial equation $h_1^2/2 + h_1/2 + \mathtt{i}_1 + \mathtt{k}_0$. Since $\mathtt{i}_1$ is equal to $h_1 + 1$, the polynomial is simplified to $h_1^2/2 + 3h_1/2 + \mathtt{k}_0 + 1$.

The example concludes by determining the η values for `i`, `j`, and `k`. Variable `i` exits with the value $\mathtt{max}(\mathtt{1}, \mathtt{n}_0 + \mathtt{1})$ and `k` with an expression polynomial in h_1. The exit value of `j` is unknown since the induction expression for $\mathtt{j}_5$ is a wrap-around variable. The tripcount tc_1 is not known to be greater than 1, so no simpler expressions can be produced. The compiler may attempt to use the same lower-limit information we described earlier to determine if the tripcount indicates whether the initial value or the subsequent induction expression in the wrap-around variable is being used, and also in the `max` expression, but in this case no information about $\mathtt{n}_0$ is known.

Chapter 6

Experimental Results

In this chapter we present a look at the types of sequences found in scientific Fortran programs. Unsurprisingly, the majority of (scalar, integral) variables are classified as invariant, linear, or indeterminate ($-$). This data also presents some insight into the nature of the data dependence problem. Data dependence tests vary greatly in efficiency and accuracy. The GCD test considers only the coefficients of the loop indexes: if their GCD divides the constant term, there is an integer solution to the dependence equation, and a dependence may exist (depending on the loop bounds). If the dependence equation contains unknown variables, this test may not be used. Other tests have other constraints. It is important to examine the types of expressions that occur in the subscript expressions that produce the dependence equations [24].

Our algorithm for classifying induction variables was run on the Perfect Club benchmark suite [6], the RiCEPS suite, the Mendez benchmarks from the NBS collection, and a few other common scientific packages such as EISPACK and the Livermore Fortran Kernels. Together these programs consist of approximately 140,000 lines of Fortran.

The (lexical) occurrences of each expression in the programs were recorded by sequence class and by sequence expression: *invariant*, *linear*, *indeterminate*, etc. Note that a linear expression with all coefficients equal to zero is considered invariant. Indeterminate expressions are those values for which the compiler cannot determine a sequence expression. Table 6.1 presents the frequency of sequence classes for scalar, integral fetches for a sampling of the benchmarks. The data is presented as a percentage of the total number of fetches; empty entries signify no occurrences of that class were found, and ϵ entries signify a percentage less than 0.1 but greater than zero. While the ratios vary greatly across some

	qcd	spec77	spice	ocean	trfd	perfect	riceps	eispack	lfk
invariant	38.3	30.3	28.0	51.6	59.3	38.9	46.8	41.7	42.1
linear	20.1	61.4	5.2	18.9	26.6	27.1	33.8	45.6	31.6
indeterm	40.5	8.1	65.4	28.6	7.8	32.9	18.2	11.4	24.6
poly					2.4	ϵ	ϵ	ϵ	
geom	0.5		ϵ	0.7		0.1	0.2		0.3
wrap	0.2	0.4	0.7	0.2	3.9	0.5	0.6	1.0	0.4
periodic							0.3		
monotonic	0.3	0.3	0.7			0.3	0.1	0.7	0.9
total refs	1480	3400	15888	1859	794	47704	31013	8694	1991

Table 6.1: Subscript Classifications in Innermost Loops

programs, in all cases the majority of fetches were indeterminate, invariant, or linear.

More interesting are frequencies of the classes of expressions used as subscripts. Because the class assigned an expression is based on the nest level of the expression, however, the data in Table 6.1 provide too coarse a view. Loop transformations require information about the invariant terms that may actually be dependent on the basic loop variables of the enclosing loops. In the following fragment, for example,

```
L3: do i = 1, n
        k = k + i
L4:     do j = 1, i
            A(k) = ...
        enddo
    enddo
```

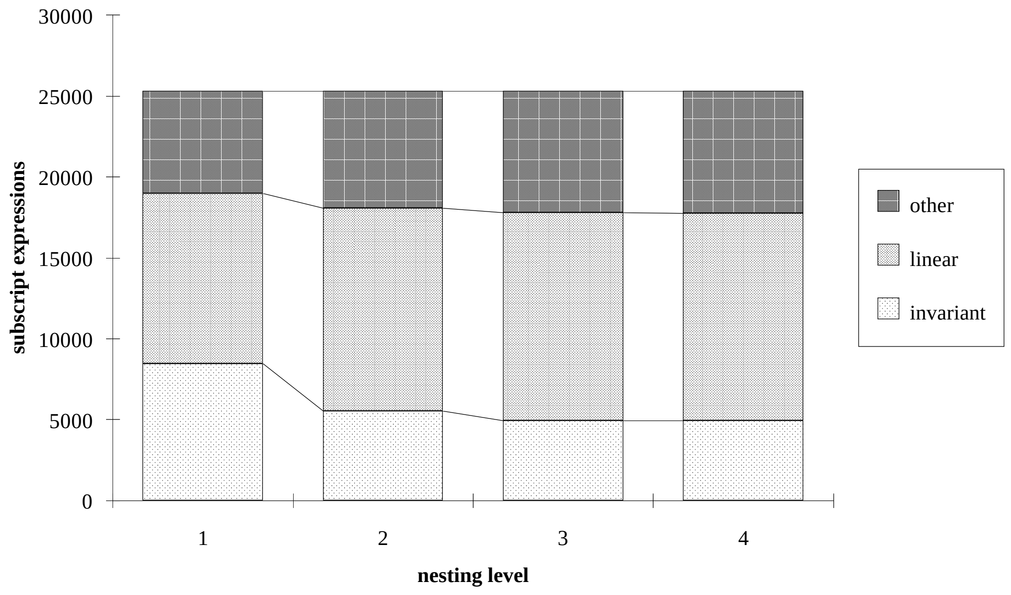
the subscript expression is classified as invariant in its immediate enclosing loop, `L4`. The sequence expression, however, is polynomial if considered in terms of its two enclosing loops, `L3` and `L4`: $1/2h_1^2 + 3/2h_1 + 2$.

In the first chart of Figure 6.1, the frequencies of classes of expressions for subscript expressions are presented for the Perfect benchmarks; subscript expressions are presented in terms of the innermost enclosing loop, the two innermost enclosing loops, and so on. (Since the frequencies of the more complex sequences (polynomials, geometrics, etc.) are small, they are grouped together as "other.") As the level of nesting considered is increased, the number of invariant expressions decreases, and the number of linear expressions increases. Beyond three loop levels, little difference is shown since only 4% of the total number of subscript expressions occur nested inside four or more loops.

Figure 6.1 also provides some information on the most common forms of subscript expressions encountered (in the Perfect benchmarks). In the second chart, c represents an integer constant; v represents any unknown variable; and k represents an integer not equal to 0 or 1. "Complex" expressions represent expressions containing nonlinear forms, including `max` operations. Nearly 15% of subscript expressions are constants; 45% are simple linear forms of one basic loop variable; and 22% cannot be determined. Of the linear coefficients not equal to ± 1, the majority are small integers, making tests for integer solutions easier in some dependence tests [18]. Relatively few classifiable subscript expressions contain unknown variables.

Up to one-quarter of the subscripts have expressions which cannot be classified. In general, nodes in the SSA graph are classified as indeterminate due to fetches of noninteger values and nontrivial SCCs not matching one of the sequence variable forms. There are three principal causes for this. First, variables used as procedure arguments are necessarily considered to be indeterminate; interprocedural analysis may be able to determine if these variables are not modified within the procedure, resulting in some gain. Second, many codes contain indexed array references, such as `A(I(k))`. In SPICE, for example, which has the most indeterminate expressions, 60% of the indeterminate expressions are caused by indexed references. Without user assertions, a compiler cannot hope to resolve these cases. Third, if a variable is conditionally incremented in a loop by a constant with unknown sign, then the compiler cannot determine if the sequence is increasing or decreasing.

Other researchers have recently examined various aspects of subscript expressions with similar results [12, 18]. These results represent a first approximation to help determine what types of data dependence tests should be applied. Further study is needed, particularly in analyzing pairwise comparisons of references to the same array at all levels in each loop nest.



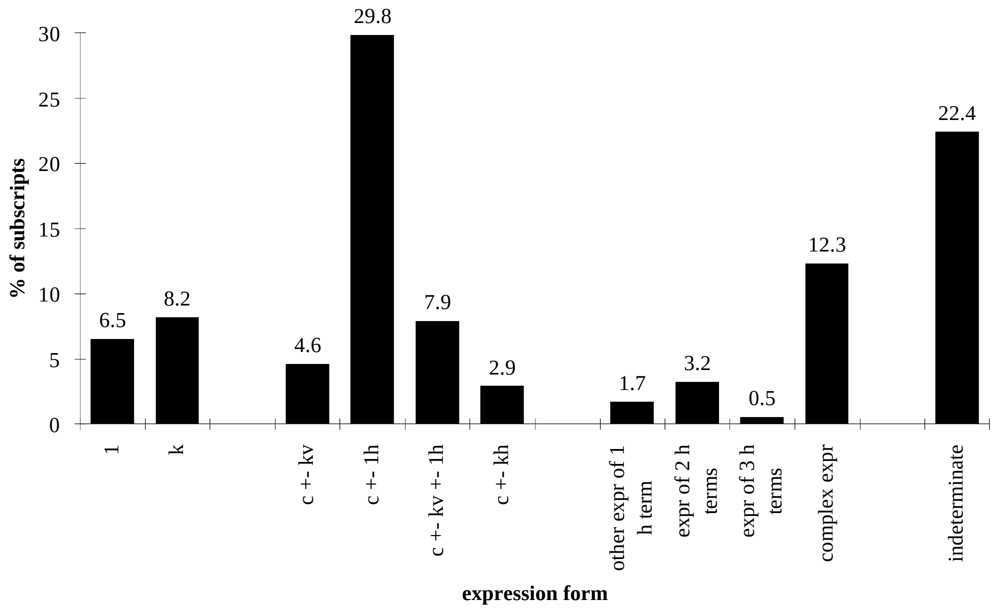


Figure 6.1: Subscript expression data

Chapter 7

Related Work

7.1 Linear Induction Variables and Strength Reduction

An early paper on program dependence graphs by Ferrante and Ottenstein suggests that basic linear IVs may be found by identifying strongly connected components of subgraphs corresponding to loops in the program [10]. Unsurprisingly, their representation of program graphs has many features in common with SSA form, e.g., merge nodes, data dependence edges. Still earlier, Loveman and Faneuf noted that in their *p-graph* representation variables introduced by strength reduction "have two generations and one merge, and all uses are generated by the merge," which is closely related to our property of μ-functions [16]. Although developed independently, our work may be viewed as a (much expanded) current treatment of these early ideas.

Classical linear induction variable detection for strength reduction, particularly of array expressions, is well covered in the literature. The usual approach is to use reaching-definition information within a loop body and search for assignments of the form `i = i` $\pm$ `c`, where `c` is loop-invariant. This defines `i` as a *basic induction variable.* Other assignments of the form `j = c*i + k`, where `c` and `k` are loop-invariant (possibly zero), associate with `j` the tuple $\langle \mathtt{i}, \mathtt{c}, \mathtt{k} \rangle$, putting `j` in the *family* of `i` [1].

Mutually defined linear IVs cannot be found by this algorithm, however, since the "other" variable on the right-hand side is not known to be in any family of induction variables. In the PTRAN compiler, such cases are solved by an optimistic data-flow technique which initially assumes all variables are linear induction variables until a contradiction exists [2]. Allen et al present a comprehensive treatment of strength reduction by recognizing more general linear cases [3].

Tu and Padua have used our SSA technique for linear induction (and monotonic) variable detection in order to perform symbolic analysis of array bounds and subscript expressions within conditionals and loops for array privatization [22]. They have found that in order to accommodate conditional expressions fully by symbolic analysis, however, more detailed predicate information is needed. Other current work by Havlak and others may address the use of γ-functions for this purpose [14, 20].

7.2 Sequence Detection

Recently, other researchers have considered various ways of extending the class of sequences usually detected. Gupta and Spezialetti have extended the traditional data-flow approach to classify "monotonic" statements for such diverse applications as run-time array bounds checking, dependence analysis, and run-time detection of access anomalies [11]. Their technique uses an iterative algorithm to detect "basic" and "dependent" monotonic statements in loops which are not nested, including both *regular* (arithmetic, geometric) and *irregular* monotonic sequences. Our demand-driven treatment is at least as powerful, and although it is still iterative for certain classes of sequences, we feel it may be, in general, more efficient and certainly simpler. Further, our method lends itself readily to an implementation of the expressions for each sequence, a problem not addressed in their paper.

Much research has been performed on the symbolic interpretation of programs. Attempting to compute the values for variables along all paths of a program, however, entails computing and maintaining path conditions, and it is difficult to compute solutions to the resulting recurrences that correspond to sequences. Ammarguellat and Harrison have presented a formalism in which abstract interpretation is used to associate a symbolic expression with each variable assigned within a loop; these symbolic expressions are then compared against known patterns (templates) representing sequences [4]. However, the inefficiency of general symbolic systems has precluded it from use in optimizing compilers [15].

7.3 Parafrase-2

Haghighat and Polychronopoulos have described an advanced symbolic analysis system for the Parafrase-2 compiler that recognizes many of the same sequences as our method [13]. To our knowledge, this is the only other scheme proposed that recognizes in an efficient manner the wide range of sequence forms required by high-performance compilers. The Parafrase approach differs from ours in five key respects:

1. The Parafrase compiler attempts to determine an *abstract model* for each loop in the program, in effect collapsing the loop into a set of statements that summarize the effect of the loop on the variables assigned within it. This collapses the CFG into a DAG. Global forward substitution is used to propagate symbolic expressions. The symbolic interpreter used does not maintain path conditions, but instead uses a join function to determine equivalence along incoming edges (similar in spirit to our transfer function for ϕ-functions): the join function finds variables that have the same symbolic values on different incoming edges. While this does cut down on the amount of information the Parafrase compiler must maintain, the system is not fundamentally based on a sparse representation, e.g., SSA, and cannot forward expressions to only those points in the program that require them.

2. Sequence forms are recognized in Parafrase by solving the recurrences in the abstract model's statements. For increasing values of d, the compiler evaluates symbolically the first $d + 2$ iterations, assuming symbolic values for the initial iteration, and attempts to interpolate a d-degree polynomial that fits the recurrence. In Nascent, the degree is known right away, since it is determined from the operations within the SCC. Within Nascent, therefore, linear IVs (by far the most common case) are recognized directly and require only one pass around the operations making up the induction sequence; in Parafrase, the entire loop body is evaluated at least twice.

3. Parafrase recognizes wrap-around variables by repeating the interpolation process at subsequent starting points. That is, having failed to find a fit for the values produced by iterations $1..n$, the compiler attempts to find a fit for the values produced by iterations $2..n$, and so on. Our method recognizes wrap-around variables

immediately based on the nature of the component in the SSA graph — it must be a solitary μ-function.

4. The Parafrase compiler does not recognize monotonic forms. The detection of monotonic forms requires the ability to distinguish incoming symbolic values at join points. Since the Parafrase compiler's join function detects only equivalent values, the opportunity to make this distinction is lost. The mechanism we use to detect monotonic variables includes the join in the component, and the comparison of incoming values is easily accomplished.

5. Nascent uses a strengthening process to simplify sequences. For example, a linear IV with an increment of zero should be represented as an invariant. Parafrase captures these cases immediately, based on the nature of the sequence.

Haghighat and Polychronopoulos have presented a chart listing 10 forms of induction expressions and symbolic substitutions felt to be essential to parallelizing programs [13]. Four state-of-the-art parallelizing compilers were compared using a test-suite containing these forms: Parafrase-2 recognizes all 10; the Titan compiler from Stardent recognizes 2; and both the KAP compiler from Kuck and Associates and the VAST-2 compiler from Pacific Sierra Research recognize none.[1] The methods presented here allow for the recognition of all 10.

In the limit, the power of the Parafrase and Nascent approaches are close, the differences in technique are important. Most importantly, our approach uses a sparse representation that minimizes the data needed and then solves sequences directly within the same framework.

[1]The versions of the compilers tested are not stated.

Chapter 8

Conclusions

This thesis has discussed an SSA-based algorithm for detecting sequence variables within programs, including classical linear induction variables. Similar to demand-driven symbolic interpretation, our technique is based on a generalization of (sparse, unconditional) demand-driven constant propagation. We have presented a simple and intuitive algorithm for classifying several types of sequences based on strongly connected components of the associated SSA graph. This form of analysis has several advantages over previous methods. It is clearly a more general solution than traditional pattern matching, and we have shown in our implementation that it can recognize the same types of expressions as other proposed schemes.

The most obvious application of the algorithms presented here is strength reduction. All linear induction expressions can be detected very quickly with only one pass over the SSA graph. With as much support added for symbolic forms as desired, compilers that require dependence information for subscripts will be able to detect linear dependences as well as more complex forms. Our technique can be readily incorporated into existing compilers that use internal representations similar to SSA.

Information about variables, such as lower limits and induction variable forms, may be profitably used in optimizing compile-time and run-time checks. With the additional use of assertions derived from the symbolic expressions of variables, in a demand-driven SSA context, for example, a compiler may be better able to eliminate array bounds checks in loops.

The primary motivation for this work, however, is an advanced classification of expressions in support of data dependence testing. Our experimental results indicate that the vast majority of sequence expressions used in subscripts are invariant, linear, or indeterminate; the remaining more exotic sequence classes amount to at most a few percent. However, parallelizing compilers, such as Nascent, can benefit greatly from this: in discussing their experiences in hand-parallelizing four of the Perfect benchmarks, Eigenmann et al note the importance of having compilers detect *generalized induction variables* [9].[1] A factor of 8 speedup was obtained in one case by replacing a geometric induction variable with its closed form in terms of the loop index.

[1]Generalized induction variables are referred to in this work as polynomial and geometric IVs.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[3] F. E. Allen, John Cocke, and Ken Kennedy. Reduction in operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, 79–101. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[4] Zahira Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the '90 ACM SIGPLAN Conference on Programming Language Design and Implementation,* White Plains, New York, 283–295. ACM Press, New York, June 1990.

[5] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the '90 ACM SIGPLAN Conference on Programming Language Design and Implementation,* White Plains, New York, 257–271. ACM Press, New York, June 1990.

[6] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the Perfect Benchmarks. In *Proceedings of the International Conference on Supercomputing*, 254–266. ACM Press, New York, March 1990.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing Static Single Assignment form. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages,* New York, 25–35. ACM Press, New York, January 1989.

[8] 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.

[9] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fourth International Workshop on Languages and Compilers for Parallel Computing*, 65–83. Springer-Verlag, New York, 1991. Lecture Notes in Computer Science, volume 589.

[10] Jeanne Ferrante and Karl J. Ottenstein. A program form based on data dependency in predicate regions. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages,* New York, 217–236. ACM Press, New York, January 1983.

[11] R. Gupta and M. Spezialetti. Loop monotonic computations: An approach for the efficient run-time detection of races. In *Proceedings of the SIGSOFT Symposium on Testing, Analysis, and Verification*, 98–111. ACM Press, New York, October 1991.

[12] Mohammad Reza Haghighat. Symbolic dependence analysis for high performance parallelizing compilers. Technical Report 995, University of Illinois, Center for Supercomputing Research & Development, Urbana, Illinois, May 1990. M.S. thesis.

[13] Mohammed R. Haghighat and Constantine D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth International Workshop on Languages and Compilers for Parallel Computing*, 538–562. Springer-Verlag, New York, 1992. Lecture Notes in Computer Science, volume 757.

[14] Paul Havlak. Construction of thinned gated single-assignment form. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Sixth International Workshop on Languages and Compilers for Parallel Computing*, 477–499. Springer-Verlag, New York, 1993. Lecture Notes in Computer Science, volume 768.

[15] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, 5–54. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[16] David B. Loveman and Ross A. Faneuf. Program optimization – Theory and practice. In *Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines*, 97–102. ACM Press, New York, March 1975.

[17] David A. Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[18] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[19] Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *Proceedings of the 27th Annual Hawaii International Conferences on System Sciences*, 43–52. IEEE Press, Los Alamitos, California, January 1994.

[20] Eric Stoltz, Michael Wolfe, and Michael P. Gerlek. Constant propagation: A fresh, demand-driven look. In *Proceedings of the Symposium on Applied Computing*, 400–404. ACM Press, New York, March 1994.

[21] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[22] Peng Tu and David Padua. Automatic array privatization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Sixth International Workshop on Languages and Compilers for Parallel Computing*, 500–521. Springer-Verlag, New York, 1993. Lecture Notes in Computer Science, volume 768.

[23] Michael Wolfe. Beyond induction variables. In *Proceedings of the '92 ACM SIGPLAN Conference on Programming Language Design and Implementation,* San Francisco, California, 162–174. ACM Press, New York, June 1992.

[24] Michael Wolfe and Utpal Banerjee. Data dependence and its applications to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, April 1987.

Biographical Note

Michael P. Gerlek was born and raised in Massachusetts, in the center of Boston's "hi-tech corridor." He attended Merrimack College in Andover, Massachusetts and graduated in 1991 with the degree of Bachelor of Science in Computer Science. His interest in pursuing a career in computer science research, and specifically in the area of high performance computer systems, was formed during his undergraduate years when he was awarded a U.S. Department of Energy Fellowship for Undergraduate Research and as a result spent nearly two years serving interships at the Los Alamos National Laboratory in New Mexico.

In 1991, Michael entered the Computer Science and Engineering Department of the Oregon Graduate Institute. Under the guidance of Dr. Michael Wolfe, he was a member of the Sparse Group performing research in the field of compilers for high performance computing. He studied variously applications of Static Single Assignment forms, compiler back-ends, and multithreaded systems, and contributed to the development of the Nascent compiler. While at OGI he authored or coauthored several papers and technical reports, and was one of two selected by the faculty for the award of Outstanding Student in 1994. He completed his M.S. degree in 1995.

In 1994, he spent six months working at the Supercomputer Systems Division (SSD) of Intel Corporation where he analyzed the performance of advanced architectures and compilers. Michael plans to pursue a career in industrial R&D, working in the realm of compilers and performance analysis for high performance systems.