RICE UNIVERSITY

Optimizing Programs over the Constructive Reals

by

Vernon A. Lee Jr.

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

Doctor of Philosophy

Approved, Thesis Committee:

Hans-J. Boehm, Director
Associate Professor of Computer Science

Robert Cartwright
Professor of Computer Science

John Dennis
Noah Harding Professor of Mathematical Science

Keith Cooper
Associate Professor of Computer Science

Houston, Texas

March, 1991

Optimizing Programs over the Constructive Reals

Vernon A. Lee Jr.

Abstract

A real number x is *constructive* if an algorithm can be given that computes arbitrarily accurate approximations to x. *Constructive real arithmetic* provides the numerical programmer with a useful mechanism for prototyping numerical programs and experimenting with numerical algorithms. A constructive real arithmetic package could be used to evaluate expressions in a symbolic algebra system, thus avoiding potential instability problems. Constructive real arithmetic could also be used by an automatic assertion checking system in a programming environment, since assertions in numerical programs are naturally stated in terms of real, not floating point, arithmetic. Numerical programmers could use it to test programs for correctness independent of round-off error; constructive real arithmetic could also be used for computations performed in highly unstable sections of programs.

The basic differences between implementations of constructive real arithmetic is the method chosen to represent the algorithms that compute the approximations to constructive real values. Previous implementations have used different specific representations, but all fall into one of two categories. Most commonly, a complete execution history is recorded as the program executes, consisting of all information necessary to recompute any constructive real value. When an approximation to a real value is required, these methods trace back through the execution history, and compute an approximation using sufficient resources to produce the required accuracy in the result. Because a complete execution history is stored, there is a large amount of overhead associated with these methods. Examples of this type of implementation include lazy infinite continued fractions, lazy infinite digit sequences, and functional approaches based on bignum arithmetic or on variable-precision intervals. A proof is presented that shows that one can use variable-precision intervals in such a way that the cost of all the reexecutions necessary to compute an approximation of a given accuracy is only a constant factor more than the final computation.

An alternate approach simply executes the program using variable-precision interval arithmetic, and requires that that programmer explicitly examine the results after execution and decide whether to reexecute the program using higher precision, and, if so, which precision to use. This method has several drawbacks as an implementation of constructive real arithmetic. The primary difficulty is that it does not provide what we consider to be the proper abstraction for constructive real arithmetic. We view this approach more a *tool* that can be used to construct a constructive real arithmetic package.

In this dissertation we present a new approach that uses data-flow analysis of the program text to discover, at compile-time, much of the information previously gathered at run-time in the methods that store execution histories. We can determine prior to execution which instructions may require approximations from their arguments; these instructions are called *output instructions*. Using *static dependences*, the program instructions are partitioned into sets, based on the output instructions that may depend on the value that each instruction computes. A new program is produced that consists of a copy of the original program that uses *interval arithmetic* at a fixed precision, and a set of subroutines formed from the partition sets. The program copy is then executed, and when an output operation occurs, the subroutines that compute arguments of the output operation are executed using higher precision, which is increased until those arguments are sufficiently accurate. The reexecution-bound proven for the variable-precision interval execution history method is retained in this implementation, so that the sum of the costs of the reexecutions is only a constant factor more than the cost of the final, highest-precision, reexecution.

Acknowledgments

First and foremost I wish to thank my advisor, Hans Boehm, for his guidance, patience, and confidence in me. I would also like to thank the other members of my committee, Corky Cartwright, John Dennis, and Keith Cooper, for their support and advice during the years of my work. I also owe much to Preston Briggs, for cheerfully receiving all my bug reports on his code generator.

Thanks are also due to Xerox Corporation, for providing to me the opportunity to work closely with my advisor during the final months of my dissertation work. In addition, my friends and fellow students at Rice, and elsewhere, have provided invaluable assistance, advice, and support. And finally, I wish to thank my parents, for always telling me I could accomplish anything.

Contents

Chapter 1

Background

In this chapter we provide an introduction to and history of constructive real arithmetic, and define some key terms to be used throughout this dissertation. Previous constructive real implementations are reviewed, and their strengths and weaknesses discussed.

1.1 Definitions

We say a real number x is *constructive* if one can give an algorithm g_x to calculate arbitrarily accurate approximations to x [5] [24]. Such numbers are alternatedly called *representable* or *recursive*. For example, 1, π, and $\sqrt{2}$ are all constructive real values. An example of a real number that is not constructive would be the number X defined as the decimal expansion $0.d_0d_1d_2\ldots$ where d_i is defined to be 1 if the i^{th} program in a listing of all programs halts on all inputs, and otherwise 0. Clearly, this X is a real number for which no approximating algorithm can be given.

Constructive real values are represented by algorithms that produce approximations of arbitrary accuracy on demand. An implementation that uses approximations from a set T is called a *T-based implementation*. Thus an implementation that uses floating point values as its approximation type is a floating point-based implementation, and one that uses rational values would be a rational-based implementation.

Real values are created by either converting finite values into reals, or by applying real operations to other real values. Finite values include all non-real numerical values, including integers, rationals, and floating point values. When these values are converted to constructive real form, they can be viewed as *exact* approximations to themselves. Examples of real operations include addition, multiplication, and square root.

When a constructive real function is applied to a constructive real value x, we create a new approximating algorithm for the resulting real value. The algorithm produced by a T-based implementation to perform a function $\circ$ on constructive real values has the following basic structure:

- An argument is given to the algorithm that tells it how accurate an approximation should be produced.
- The algorithm in turn invokes the approximation algorithms for its arguments to get type-T approximations to each of them.
- Finally, it executes the T-function corresponding to $\circ$ on the argument approximations.

For example, to add two constructive real values in a floating point-based implementation, a function is created that computes floating point approximations to the two arguments, and then applies the floating point addition function to those approximations.

Thus the algorithm for each real value produced by a real operation has references to the algorithms for each of its argument values. This gives rise to a graph structure, such as the one shown in Figure 1.1. Graphs created for real values are Directed Acyclic Graphs, or DAG*s*. DAGs are basically tree structures, except that nodes may have more than one parent, and there may be many nodes with no parents (instead of a single root as with trees). This DAG-building aspect of exact real implementations will prove to be important later on.

A program written to use constructive real values will differ in some important ways from a program that uses floating point or rational arithmetic. Turing and Rice [30], [26] give the following theorem:

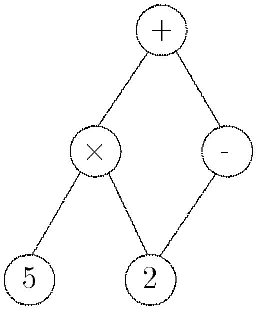

Figure 1.1 DAG for the expression $(5 \times 2) + (-2)$

Theorem 1.1 *Let $r \in \mathbf{R}$ represent an arbitrary computable real number. No algorithm can compute, in finite time:*

1. *if $r = 0$*
2. *if $r > r'$ for any fixed computable real r'*
3. *if r is rational*
4. *the first digit of the decimal expansion of r*
5. *the first term $\lfloor r \rfloor$ of the regular continued fraction expansion of r*
6. *the value $f(r)$ of any function f that is not continuous at r*

We call operations such as those described by cases 1 through 6 *output operations*. Unfortunately such operations are often important parts of programs — particularly such operations as comparison and printing out the first several digits of a decimal (or other base) expansion of a real number. Since we cannot determine the result of such output operations exactly, we determine their results to within a given tolerance. For example, instead of computing whether a constructive real x is equal to 0, we compute whether x is within a tolerance ϵ of 0, using a finite approximation to the real value x.

If $\bar{x}$ is an approximation to x that is accurate to $\pm\epsilon$, i.e., $|x - \bar{x}| \leq \epsilon$, then we can compare the *approximation* $\bar{x}$ to 0. If $|\bar{x}|$ is greater than ϵ, there is no way that x itself can be 0, since $|x - \bar{x}| \leq \epsilon$. In this case, if $\bar{x}$ is greater than 0, we can then conclude $x > 0$; similarly, if $\bar{x} < 0$, we know $x < 0$. If, however, $|\bar{x}|$ is less than ϵ, we do not know on which side of 0 that x lies; in this case, we say that $x = 0$ to a tolerance of ϵ.

Each output operation in a program specifies such a tolerance, or *accuracy requirement*, for each of its arguments. Before executing I, we must compute approximations to each of I's arguments that satisfy the corresponding accuracy requirement. In the example above, ϵ is the accuracy requirement for the comparison to 0.

In the remainder of this chapter, we will review previous implementation strategies for the constructive reals, and analyze their strengths and weaknesses.

1.2 Lazy List Implementations

1.2.1 Continued Fractions

The general form for continued fractions is shown in Figure 1.2. Each a_i and b_i is an integer value.

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{\ddots \; \overline{\qquad a_{n-1} + \cfrac{b_n}{a_n}}}}}$$

Figure 1.2 A continued fraction.

We can require that each $b_i = 1$ without restricting the values that may be represented [19]; with this restriction we can give continued fractions by simply listing the a_i terms. A common abbreviation takes the form $/a_0, a_1, \ldots/$ for the continued fraction shown in Figure 1.2, provided all b_i terms are required to be 1. Note that even with this restriction that all $b_i = 1$, there are still many ways to compute the terms of a continued fraction. For example, a *classical* **Z** *continued fraction* representation $/a_0, a_1, \ldots, a_i, \ldots/$ for a computable real value r is produced by the following equations:

$$\begin{aligned} r_0 &= r \\ a_t &= \lfloor r_t \rceil \\ r_t &= a_t + \frac{1}{r_{t-1}} \end{aligned}$$

where $\lfloor x \rceil$ produces the integer nearest to the constructive real value x. Using, for example, the floor function $\lfloor \cdot \rfloor$, we could generate a different series of continued fraction terms that converges to the same real number r.

Continued fractions can be used to approximate constructive real values, by representing the continued fractions using *lazy lists*. A lazy list consists of a list of terms, and a procedure for computing more terms. When a more accurate approximation is required, the procedure is invoked as necessary until enough terms are computed.

Note that the procedure for the result of, for example, an addition will contain references to the lazy lists representations of the arguments to that addition operation; thus the continued-fraction approach does construct DAGs.

The algorithm for computing the first term of a continued fraction is similar to that for computing the first digit of a decimal expansion. It follows easily from Theorem 1.1 that $\lfloor r_t \rfloor$ is not computable for any value of t, nor is any discontinuous function on the constructive reals. We can get around this problem by using a *redundant representation* [33][25]. Using a system that has more than one representation for each real value x allows us to compute terms for a continued fraction expansion.

Vuillemin [31] defines the operation $\div$ such that for any real r, $z = r \div 1$ is an integer such that $-1 < r - z < 1$. Then, rather than defining $a_t = \lfloor r_t \rfloor$ as above, he defines $a_t = r_t \div 1$. Since $\div$ is computable on all constructive real values, he can then produce continued fraction approximations[1]. Note, however, that the ambiguity in the definition of $\div$ means that in this system there is no unique representation for any given real number.

Continued fractions were first suggested for use in an implementation of the constructive reals by Gosper. Gosper [13] describes a scheme for performing lazy evaluation on continued fractions; as the author acknowledges, his algorithms do not work for all inputs. He suggests a system that would be a hybrid of continued fraction arithmetic and some kind of approximate arithmetic, where the latter would be used when the continued fractions did not suffice. Vuillemin [31] presents an elegant, detailed study of continued fraction arithmetic.

1.2.2 Lazy Digit Lists

Perhaps the most straightforward way to implement the constructive reals is to use *lazy lists* of the digits of real values in some base b. Lazy digit list implementations have been studied by Boehm, Cartwright, and Riggle, in [7], and by Boehm in [6]. Schwartz further refined this approach in [28]. Lazy digit list representations are similar to the representation described above for lazy infinite continued fractions. They consist of a list of leading terms, followed by a procedure that can calculate the remaining digits. Figure 1.3 shows an example of a lazy list value.

[1] If r is undefined, r is infinite, or any term r_t of the expansion is infinite, the expansion calculation does not terminate. Vuillemin provides a slightly modified definition of $\div$ to handle these special cases.

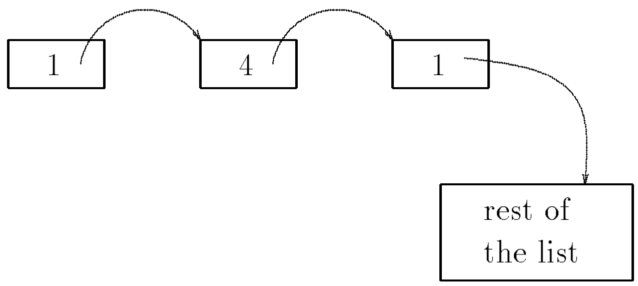

Figure 1.3 A lazy list structure representing the value $0.141\ldots$

The structure shown in Figure 1.3 can be abbreviated

$$(1, 4, 1, \bullet)$$

where the $\bullet$ represents the rest of the list, that is, the function that can be invoked to calculate the remaining digits. If a computation needs the fourth digit, the closure $\bullet$ will be invoked to calculate it, and, supposing, say, that 5 is the next digit in the expansion, the list will become

$$(1, 4, 1, 5, \bullet)$$

From Theorem 1.1 (see also [24]) however, we know that for general constructive real values we cannot compute even the first digit of the decimal expansion (or of any other normal base). Again this problem can be solved through the use of a redundant notation. The authors of [7] do this by using digits in the range

$$[-b+1, -b+2, \ldots, 0, \ldots, b-2, b-1]$$

This scheme works by containing carry values to a finite number of steps away from the point a carry originates. That is, carry propagation is limited to a finite number of digits. Thus, after a certain number of digits to the right of say the k^{th} digit have been computed, one can declare that the k^{th} digit is fixed, and will not change. The carry value is "swallowed up" by the extra storage capacity at each digit location. In a non-redundant representation, carries can propagate arbitrarily far; this is in part what gives rise to the properties of Theorem 1.1.

1.2.3 Problems

The apparent advantage of a lazy list approach is that computation proceeds incrementally, and only on demand. It seems that this method would calculate the required information and nothing else.

However, a serious problem arises, which we can call a problem of *granularity*. One may not evaluate "half a digit" — to examine any properties of a digit of a real value, one must fully evaluate that digit. This can cause calculation to perform much *more* computation than necessary to calculate a value.

Any operation on finite approximations to real values, such as arbitrary-sized integers or variable precision floating point values, engenders a loss of accuracy; that is, the error increases with each operation. For example, the sum of two numbers with errors of $\pm\epsilon$ has error $\pm 2\epsilon$ plus any error added due to round-off. Thus, to produce an approximation to the result of a real operation that is accurate to within some error α, we must ensure that the arguments used to compute the result of the operation are sufficiently more accurate than α.

For example, given the lazy list representations $(1, 4, 1, 5, \bullet)$ and $(3, 3, 3, 3, \circ)$, we know that each digit of each list is correct. But if we add these two numbers, we get $(4, 7, 4, ?, \diamond)$; we do not know if the last digit should be 8 or 9, since we don't know the fifth digits of the two summands. If they are 5 and 6, for example, the fourth digit of the sum will be 9, but if they sum to less than 10, the fourth digit of the sum will be 8. Therefore, to produce an approximation of $x + y$ accurate to four digits, we need approximations to x and y that are accurate to five digits. If x and y are to be multiplied, we need much more than one extra digit in each (the exact amount required would depend on the values of x and y).

Unfortunately, in the lazy list implementation, the only unit in which we can generate this extra accuracy is the digit used for the representation. Naturally, to take advantage of the machine's arithmetic capability, using machine words as the size of the digits in the lists is most efficient. However, this means that *each* operation will add *at least* an extra digit of accuracy requirement to its arguments — that is the smallest amount of additional accuracy that can be requested.

Any value from which a given amount of accuracy is required is called an *output value*. Output values include arguments to statements that print approximations, comparisons, and such operations as rounding to the nearest integer.

As another example, if we want n digits of accuracy from the expression shown in Figure 1.4, the outermost addition will require $n + 1$ digits of accuracy from its arguments, the next outermost $n+2$, and so on, until the innermost use $n+5$ digits of accuracy. If we use base 2^{31} as in [7], this translates into 155 extra bits of accuracy in each of the deepest arguments. It is easy to show that only $\lceil ln_2 6 \rceil = 3$ extra bits are required for a summation of six values; that is, 152 bits more accuracy than necessary are computed.

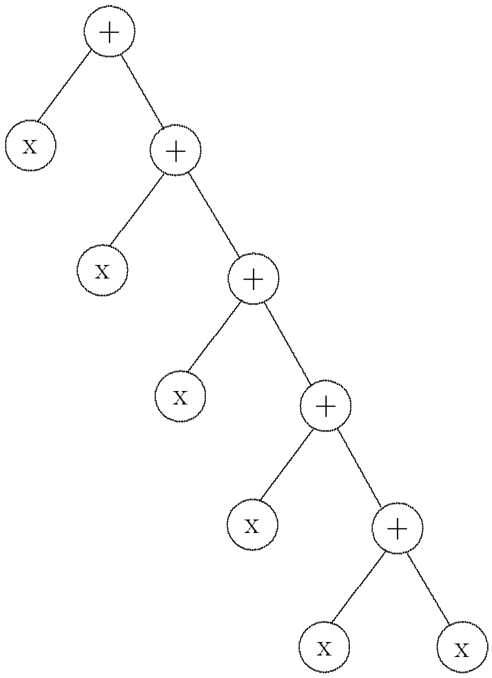

Figure 1.4 Addition of six values.

The problem is actually much worse for operations more complex than addition. In the case of division, the number of extra digits calculated at each level of an expression can grow *exponentially* as the depth of the expression increases [7]. Computing approximations to division trees of any depth becomes intractable. The actual relative precision required for a single division is only a constant amount more than the accuracy required of the result — thus this exponential growth is unnecessary, a side-effect of the choice of representation.

Granularity problems are not the only difficulties one encounters in the lazy list implementations of the reals. Some others include

1. Multiplication and division routines are difficult to write in such a way that precision demands are minimized.

2. The lists use a lot of memory — incrementality comes at a high cost in space.

3. The accuracy requests in this method are passed top-down through the expression DAGs. It will be shown in Chapter (next chapter) that a bottom-up approach is preferable.

Another problem with lazy list objects is that proving termination and resource consumption theorems about programs involving them seems to be quite difficult. In the next section we discuss a different approach to implementing the constructive reals, that avoids the granularity problem.

1.3 Accuracy Functions

1.3.1 Description

Bishop [5] defines a real number as a *regular sequence* of pairs of rational numbers. A sequence $\{a_0, a_2, \ldots\}$ is regular if for each a_m, a_n in the sequence

$$|a_m - a_n| \leq m^{-1} + n^{-1}$$

This definition leads to the following formulation for a real number x

$$|f_x(n) - x| \leq \frac{1}{n} \tag{1.1}$$

where f_x maps integers to rational numbers. This says that for any constructive real number x, there is a function f_x that can compute an arbitrarily accurate approximation to x. As n becomes larger, $1/n$, the maximum error in the result, goes to 0. The approach discussed in this section, *functional* representation of the constructive reals, implements real values directly as functions such as f_x in the above.

There are some difficulties with implementing the reals using the formula in equation 1.1. First, the range of f_x is over the rationals, and rational arithmetic can be expensive. Second, extremely large values of n are necessary to specify very accurate results — for example, to compute 50 decimal digits of accuracy we would have to use $n = -10^{50}$. This makes computations involving n, the argument to f_x, expensive. Finally, since n is an integer value, we are forced to calculate *all* values to within an error of ± 1 ($n \geq 1 \Rightarrow |\frac{1}{n}| \leq 1$). If x is very large, then we must then evaluate its entire integer part to examine any of its properties; consider comparing x to 0 if $x = 10^{100}$.

In [6], the author uses the following formulation instead:

$$|b^n f_x(n) - x| < b^n \tag{1.2}$$

In this scheme, f_x maps integers to integers. The integers returned by f_x are arbitrary sized integers, or *bignums*, implicitly scaled by b^n, so that larger integers represent more accurate approximations. Since n is now in an exponent, its value will not get so large that computing with it will become expensive. And finally, results of any accuracy may be calculated, since for positive n the tolerance is greater than 1. We call this representation the *accuracy function* implementation, since the arguments passed to the functions are requested accuracies.

Mathematical operations are performed on these functional representations simply by combining closures. We shall use Church's lambda notation, where $f = \lambda x_0, x_1, \ldots, x_k.M$ denotes a function such that $(\lambda x_0, x_1, \ldots, x_k.M)N_0, N_1, \ldots, N_k$ results in the expression M with every instance of the variable x_i being replaced with the expression N_i for each i. Using this notation, we can give the addition operation as

$$+_{acc} = \lambda f_x, f_y.\lambda n.(f_x(n-1) + f_y(n-1))/b \tag{1.3}$$

The addition function combines the functions for x and y to produce a new functional representation. The resulting function, when invoked with an accuracy request of n, (i) invokes f_x and f_y with accuracy requests of $n+1$ (to compensate for accuracy that will be lost in addition), (ii) adds the resulting integers returned by f_x and f_y, and (iii) scales the results by the base b (so that it is scaled to n and not $n-1$).

For the algorithm in 1.3 to be correct, every f_z it produces from the addition of any two f_x and f_y must satisfy:

$$|b^n f_z(n) - (x+y)| < b^n \tag{1.4}$$

The proof that 1.3 is correct is given in [6].

Other mathematical operations are performed with algorithms similar to 1.3. Other functions must evaluate the functions for their operands to accuracies more complicated that simply increasing n to $n-1$, however. The accuracy required an operand for multiplication, for example, depends on the magnitude of the other operand. Division can be implemented using Newton iteration on bignums, as discussed in [19].

Transcendental functions such as sin, cos, etc., are implemented by explicitly calculating the Taylor series expansion using bignum arithmetic. The scaling used depends on the accuracy requested. Terms are evaluated until they vanish with respect to the requested accuracy.

Computation with this style of functional reals proceeds as follows. The program is executed, and operations are performed by combining the procedures for real values, as in 1.3. When an operation requiring accuracy is encountered, such as an operation to output the first n digits of a real value, the accuracy required is passed to the function representing that real. For example, if we have computed an expression for a real value x, as shown in Figure 1.5, and we wish to output its value to ten decimal digits to the right of the radix point, we pass -10 to the procedure for x[2].

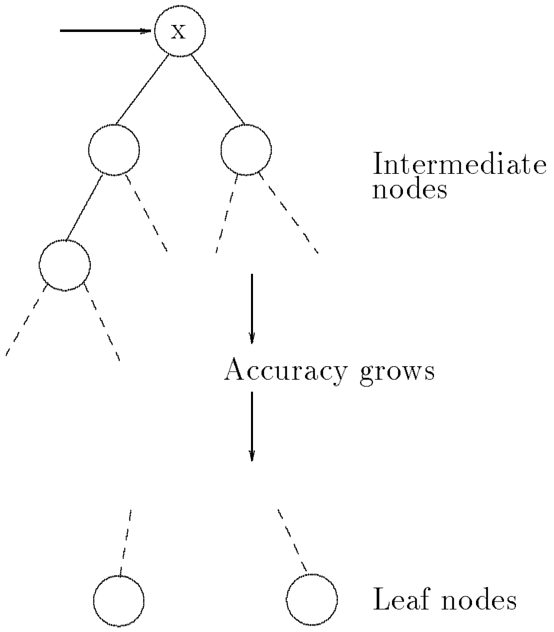

Figure 1.5 An expression for some real x calculated using functional reals.

[2]This discussion assumes the base in use to be 10 for clarity; if a different base were in use, the accuracy request would be converted from decimal to that base before being passed to the function.

Now the function for x computes the accuracy required from each of its arguments, those values that correspond to the children of x's node as shown in Figure 1.5. The function for x then passes these higher accuracies to each of its arguments, which themselves will calculate the accuracy required of their own arguments and pass that on, and so on. This continues until the function for *leaf* value is reached, that is, a value with no arguments (a constant value such as 3 or π). A leaf value needs only produce the accuracy requested and return the answer as an appropriately scaled bignum.

Note that when accuracy is increased, the minimum increase now is only one digit of base b. In [6] the author chooses $b = 4$ - this means that accuracy will increase much more slowly in this approach than in the lazy lists implementation. Looking back at the example in Figure 1.4, with a base of 4, only 10 bits of additional accuracy are required from the most deeply nested values, as opposed to the 155 extra bits that were required in the lazy digit algorithm. The differences are more marked in more complicated examples and operations.

When the arguments to a function return the requested approximations, the function computes its result, shifts away the extra digits that were requested to account for round-off error, and returns it. Computation proceeds back up the expression DAG, until the function that originally requested an approximation is reached. In our example, that corresponds to returning an approximation to the output procedure, which then outputs the value to the terminal. At this point computation proceeds again, until another operation requiring an approximation is reached.

In summary, computation flows down from the top node to the leaves, then back up to the top. The requested accuracy is given to the top node, and this request becomes larger and larger as it is passed farther and farther down into the expression DAG. When the leaves are reached, they produced appropriately scaled values and the calculation returns back up the structure. At each stage on the way up a calculation is performed, followed by a scaling operation that puts the intermediate result into the correct form for use by its parent. Finally the approximation requested is produced and processed. This general process is shown in Figure 1.6.

1.3.2 Problems

The accuracy function representation avoids the granularity problem discussed in Section 1.2.3; however, there is another problem. In general, functional representa-

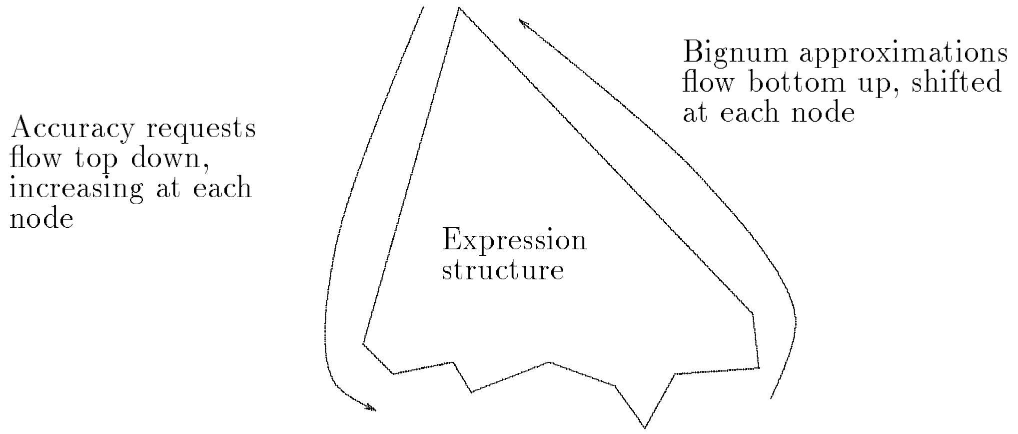

Figure 1.6 Computation flow with accuracy functions.

tions cannot make use of previously calculated information when asked to produce more accurate approximations. Some operations, such as logarithm, inverse, and square root, can use Newton iteration to add accuracy to previous results, but many operations, including addition and multiplication, have to start from scratch if they receive an accuracy higher than any previously requested. All expressions, however, can store the *most accurate approximation* yet calculated at the DAG node corresponding to that expression, and use its value when accuracy requests less than its accuracy are received.

The reason that most reevaluations at higher accuracy occur is *node sharing*. Typical program produce DAGs containing nodes that each have several parents. For example, for the simple summation loop as in Program 1.7, the expression DAG for $y(n)$ is Figure 1.8 is produced.

```
y(1) := 0
for i := 2 to n
     y(i) := y(i - 1) + C
end
```

Figure 1.7 A Simple Summation Loop.

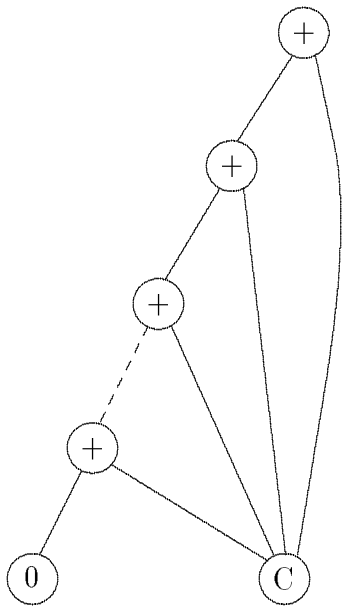

Figure 1.8 Expression DAG Produced by Summation.

Now consider how a program that output each `y(i)` value would proceed. First, `y(1)` would be output, say, to k digits of accuracy. Then `y(2)` would be output — and because `y(2) = y(1) + C`, slightly more than k digits of accuracy would be required of `y(1)`. Therefore `y(1)` would be recalculated for a slightly higher accuracy.

Now when `y(i)` is output for higher `i` values, each `y(j)` for $\texttt{1} \leq \texttt{j} < \texttt{i}$ would have to be recalculated. Since we are using in general algorithms that must be recomputed from scratch, this obviously involves much unnecessary reevaluation[3].

Various optimistic strategies have been attempted, that is, algorithms that try to "over-evaluate" under the assumption that calculated values will be referenced again with higher accuracy requirements. The problems encountered in this case come from having output values embedded within and sharing parts of the structures for other output values. The program fragment shown in Figure 1.9 demonstrates this problem.

When the program executes, first $\texttt{x = 0 +}\ \pi$ will be evaluated to `m` digits of accuracy. Then $\texttt{x = (0 +}\ \pi\texttt{) +}\ \pi$ will be evaluated to `m` digits of accuracy; this will

[3]Note that even with this drawback, in our experience the functional representation is still more efficient than lazy list representations.

```
...
x := 0.0
for i := 1 to n
     x := x + π
     output x to m digits
end
```

Figure 1.9 A Simple Program Fragment.

require that the subexpression `0 + π` be evaluated to slightly more than `m` digits of accuracy. Every iteration of the loop requires that all the previous expressions for `x` be evaluated to slightly higher precision than before. Since in general functional implementations have to start evaluating from scratch (i.e., they cannot simply add a digit like lazy lists), this can be quite expensive[4]. Note that at any iteration k, an analysis of the DAG for `x` would reveal only what has happened in previous iterations, and not that the DAG will be evaluated again in the future.

Sharing is very common in typical numerical programs and can be *much* more complicated than that presented in the simple examples above. Whether or not an efficient strategy for evaluating accuracy DAGs that avoids this type of incremental reevaluation exists is an open problem. We conjecture, however, that such an strategy does not exist.

Sharing could be avoided only by copying the DAGs of each argument when *each* real operation is performed. This obviously would result in expensive copying and heap usage.

Another problem with accuracy functions will be discussed at the end of the next section.

1.4 Precision Functions

1.4.1 Intervals

An *interval* is a pair $[a, b]$ of real numbers with $a \leq b$. Intervals can be represented using either the *endpoints* a and b, or a single *midpoint* m and an *error* e, where $m = (a + b)/2$ and $e = (b - a)/2$. Examples showing these two representations

[4]Note that even with this reevaluation problem, functional implementations still seem to be significantly faster than lazy list implementations.

are given in Figure 1.10. We call the lower-valued endpoint of an interval x its *left* endpoint, and denote it $L(x)$; the higher-valued endpoint is called its *right* endpoint and is denoted $R(x)$. An interval with endpoints a and b is written $[a, b]$. The *precision* of an interval indicates the number of significant digits used to represent the midpoint (or the endpoints in that representation). The *width*, or *accuracy*, of an interval is defined by $R(x) - L(x)$.

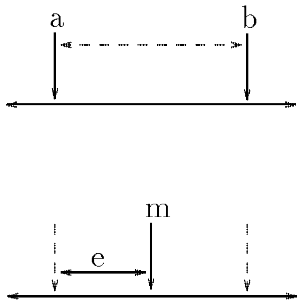

Figure 1.10 Interval Representations.

Intervals can be represented using different number lines — for example, a floating point interval is an interval whose endpoints (or midpoint and error) are given by floating point values.

Definition 1.1 *Given the set of intervals with endpoints of type T, if $f : T \times T \times \cdots \times T \mapsto T$ is a continuous function on n-vectors of T, then we define f on n-vectors of T intervals as:*

$$f(x_1, x_2, \ldots, x_n) = \{f(v_1, v_2, \ldots, v_n) \mid \forall i, 1 \leq i \leq n\,[\,v_i \in x_i\,]\,\}$$

We will make use of Definition 1.1 in Chapter 2.

Definition 1.2 *An interval x* approximates *an interval y if and only if $L(x) \leq L(y)$ and $R(x) \geq R(y)$. If x approximates y we write $x \supseteq y$.*

Definition 1.3 *A non-interval value x* approximates *another non-interval value y iff $x = y$.*

Definition 1.2 simply states that any interval x that includes the entire range of another interval y approximates y. Note that there may be no approximation relationship between overlapping intervals. Definition 1.3 is presented for the convenience of the work in the next chapter.

Definition 1.4 *An interval x* approximates *a constructive real value ω if and only if $\omega \geq L(x)$ and $\omega \leq R(x)$.*

Definition 1.4 states that an interval approximates all real values that it contains.

The width of an interval indicates how *accurate* an approximation it is to the real values it contains.

The midpoint representation is preferable for efficiency reasons; we need only store one arbitrary precision mantissa, and a fixed-size error value. We can use a fixed amount of memory for all error fields by simply shifting erroneous digits of the mantissa away. If we store the exponent in terms of a base with digits the size of machine words, we can reasonably assume that the exponent value is of bounded size. Thus, our interval representation will look something like the structure shown in Figure 1.11. If an interval computation is performed with intervals having at most precision p, that computation is called a "precision-p evaluation."

exponent value	error value	mantissa			
		word 0	word 1	...	word n

Figure 1.11 An Example Interval Implementation of Precision n.

Aberth [1] [2] [3] presents a modified version of the BASIC programming language that allows a user to run a program, examine its results, and ask that the program be reexecuted using more significant digits (i.e., more precision) in order to produce more accurate results. This is a useful framework for some computations, but we do not consider it, or other packages that provide only variable-precision intervals [21] [23] [34], to be true constructive real arithmetic. For example, this approach may produce *radically* different results for each reexecution of the program, because as the precision is increased, the results of comparisons may change. We will discuss in Chapter 4 how our method executes programs *monotonically*, that is, any output from the program is guaranteed to be a prefix of the final output — nothing output is ever "taken back," and give an example of how a simple reexecution strategy can produce non-monotonic output. The most important drawback of this approach is that it does not truly provide an abstraction of constructive real arithmetic; rather, it is more a tool that can be used to construct such an abstraction.

Richman [27] proposed a method that, given some function f, a vector of arguments, and a desired accuracy in the result, would use arbitrary-precision interval values to compute the magnitudes of the values involved in the computation of the function. Then, using these magnitude estimates, the amount of precision that is required to produce the desired accuracy is computed from the partial derivatives of f with respect to each of its arguments. This method, while interesting, requires that the partial derivatives of f be derived symbolically, and thus it is not applicable to general programs.

An functional representation based on intervals can be described using the following equations:

$$p_0 > p_1 \Rightarrow \Delta(g_x(p_0)) \leq \Delta(g_x(p_1)) \tag{1.5}$$

$$\lim_{p \to \infty} \Delta(g_x(p)) = 0 \tag{1.6}$$

$$\forall p[x \in g_x(p)] \tag{1.7}$$

The functions g_x map *precisions* to interval approximations. The precision argument value is a positive integer that specifies the maximum number of significant digits to be used to represent any value in the calculation. The function $\Delta(i)$ gives the width of the interval i; larger intervals correspond to less accurate approximations.

Thus, Equation 1.5 guarantees that the accuracy of the approximations produced by g_x will monotonically increase as the precision increases. Equation 1.6 ensures that the intervals produced converge to zero width, and Equation 1.7 guarantees that each interval produced by g_x contains x. Together, 1.6 and 1.7 imply that as the precision goes to infinity, the intervals produced converge to x. We call this functional representation the *precision function* method.

Computation with precision functions is simpler than with accuracy functions. When a function is invoked with some precision, say P, the function simply passes P to each of its arguments. It need do no accuracy calculations, as accuracy functions must, to decide what values need to be passed to its arguments. When P is passed to a leaf value, that is, a constant with no arguments, an approximation of P significant digits is produced and returned.

After receiving approximations from its arguments, each function computes its result using a arbitrary precision *interval arithmetic* package. This package is external to the constructive real type itself. The interval operation associated with the function produces a new interval approximation to the function's result, which it then passes to its parent. Computation proceeds back up the expression DAG until the node from which the reevaluation request is reached. This process is shown in Figure 1.12.

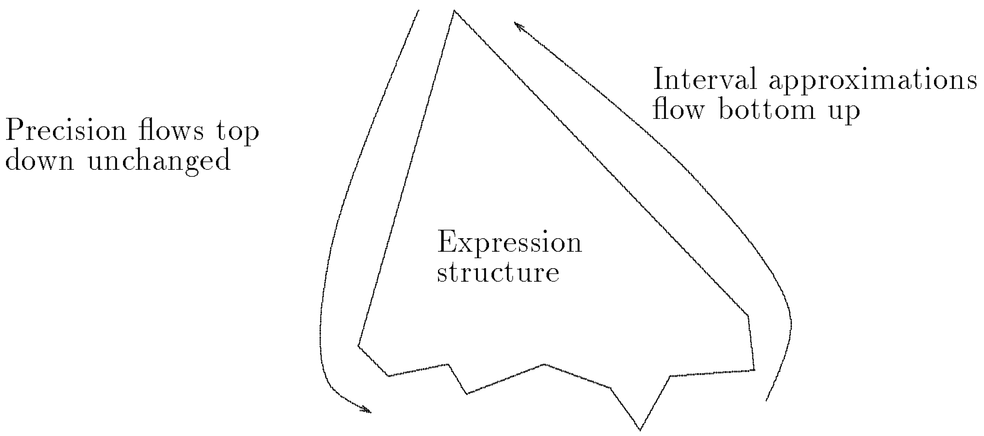

Figure 1.12 Computation flow with precision functions.

Note that we no longer do accuracy calculations at each node to account for round-off error. This computation is part of the interval arithmetic package — the space between the endpoints of each interval explicitly gives the range in which the actual value lies. As computation proceeds, the intervals grow larger. Separating the error analysis and the constructive real computation makes coding simpler and easier to optimize.

When the value requested is finally returned, the constructive reals code examines the size of the interval. If it is larger than the accuracy required by the operation that requested it, say, it is only accurate to 8 decimal digits and 10 are required, the precision is increased and the calculation is performed again. Note that in the accuracy-function representation, we know the accuracy that is produced, because it is guaranteed to be as accurate as the request.

We can characterize computation with precision functions as a *bottom-up* calculation, and accuracy functions as *top-down*. The accuracy produced by accuracy function computations is determined solely by the argument passed to the top of the expression DAG. Similarly, lazy list computations are top-down — the number of digits required at the top of an expression is exactly the accuracy that is produced. The accuracy produced by a precision function computation is determined by the accuracy produced at the leaf values of the expression and by the accuracy lost at each interval operation due to round-off error.

Bottom-up computation using precision can avoid the problems of incrementally increasing accuracy requests that were encountered with accuracy functions. If no approximation exists yet, one is computed using a default precision value. If an approximation for x already exists that is accurate enough, it is used and the computation proceeds. If the existing approximation is not accurate enough, then a new approximation is calculated using higher and higher precision until a sufficiently accurate approximation is produced. We shall show in the next section that there is a scheme that makes the cost of these reevaluations only a constant factor higher than a single evaluation.

1.4.2 Bound on Reevaluation Cost for Precision Functions

The primary problem with accuracy functions is that the function for each value must recalculate from scratch to raise the accuracy of an approximation, and the number of recalculations for a given real value can be large. In contrast, with precision functions, while complete recalculation is required in general, we can argue that number and costs of recalculation is bounded, in the sense described below.

The following is simplified by ignoring additive constants in statements about running times and calculation precisions. This does not affect the final conclusion.

We define the number of significant digits determined by an interval $[a, b]$ to be the logarithm of $(a + b)/(2(b - a))$. Note that as the interval grows smaller, the number of significant digits increases, that is

$$\lim_{(b-a)\to 0} \left(\frac{a+b}{2(b-a)} \right) = \infty$$

The following discussion is restricted to calculations involving, and resulting in, only intervals determining a positive number of significant digits. With this definition and restrictions, most operations on intervals, then, including standard arithmetic algorithms, satisfy the following statements:

1. Doubling the precision of a calculation at least doubles the number of significant digits determined by the result.

2. Doubling the precision of a calculation *at least* doubles the time required to perform the operation.

The first observation should be obvious, since the number of significant digits lost due to round off error in any given calculation is independent of the precision used to perform that calculation. Arithmetic operations on intervals are obviously at least linear in the intervals' precision, and the second statement follows easily.

Consider a sequence of n evaluations at precisions $1, 2, 4, \ldots, p/2, p$. Suppose that the cost of the final evaluation (the evaluation at precision p) is t. Then by observation 2 above, the cost of the evaluation at precision $p/2$ is bounded by $t/2$, and in general the cost of the evaluation at precision 2^i is less than or equal to $t/(2^i)$. Thus the total cost of all of these evaluations is

$$\sum_{i=1}^{n} \frac{t}{2^{n-i}}$$

which is less than $2t$.

The total time required for evaluating an expression DAG is thus bounded by a constant (2 in this simplified discussion) times the cost of the last evaluation of that graph.

1.4.3 Problems

A serious problem that both accuracy functions and precision functions share is that they both explicitly build the expression DAGs in memory as the program executes. For example, the DAG shown in Figure 1.13 shows the memory structure built for a very simple expression.

In essence, all the methods described in this chapter, construct a complete history of the program's execution. The lazy list implementations as well as the functional implementations consist of combining functions to create more complex functions, and each result is represented explicitly in a heap structure by all of the computations that created it.

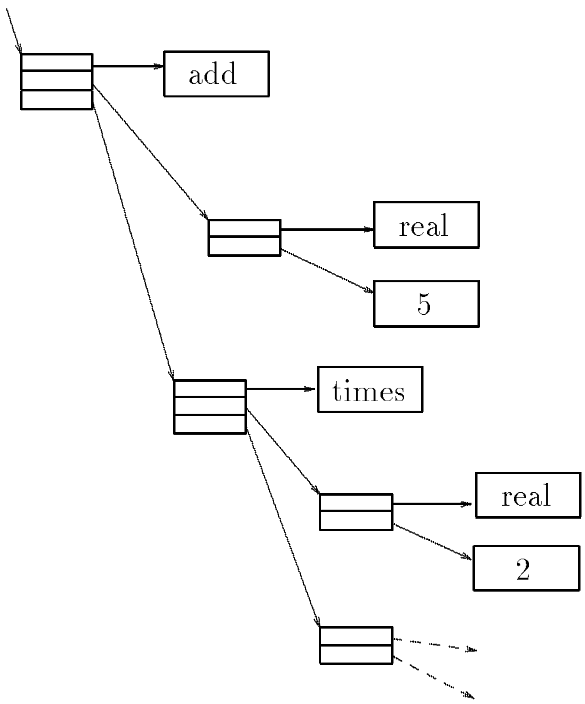

Figure 1.13 Memory Representation of $(2 \times \ldots) + 5$

The DAG in Figure 1.8, for example, encodes the loop structure that created it. DAGs from even simple programs, like Gaussian elimination on an eight by eight matrix, can generate DAGs with thousands of nodes, hundreds of levels deep. All of this memory usage is expensive, both in terms of memory use and in time spent by the garbage collector scanning it all. In the next chapter, we will discuss a strategy that avoids this problem.

Chapter 2

Representing Real Expressions as Program Code

2.1 Introduction

All of the implementation strategies in Chapter 1 — lazy digit streams, continued fractions, accuracy functions, and precision functions — perform constructive real operations by combining closures of arguments to produce new closures for results. The closures for the arguments become part of the closures for the results, and results in turn become part of larger closures — the resulting structures take the general form of DAGs. These closures can be represented in a variety of ways, depending on the particular strategy used, but they are all actual function objects produced and managed dynamically. When an approximation to a real value x is required for an output operation, the node in the DAG associated with x is invoked. It invokes the nodes associated with its arguments to produce approximations to their values, then combines those approximations to produce an approximation to x. When the approximation is produced, the output operation is performed, and the program continues, constructing more DAGs. Thus, producing an approximation to a constructive real value amounts to alternately creating and recursively executing function closures on the program heap.

DAG structures may take up a considerable amount of space at runtime; even simple programs can produce DAGs containing thousands of nodes [15]. Also, the space and time overhead associated with allocating and traversing DAG structures can be quite high. The space used, the resulting swapping costs, the garbage collection costs, and the cost of recursively traversing the structures to compute approximations all add up to a considerable amount of overhead.

Consider exactly what information is contained in the DAG structures. The nodes and leaves in DAGs contain constant values and functions that were applied to constants and to results of other functions. The particular *structure* of the DAGs reflects exactly what instructions were executed; this includes the results of conditional statements in the program and the number of times each loop was executed. Thus, a DAG is essentially a *complete history* of the program execution that produced it.

The new approach described in this dissertation computes much of the information *statically* that would normally be contained in the DAGs. We analyze the program text prior to execution, and therefore independent of any inputs, to discover as much as possible about the real values used in output operations. Then, when the program is executed, we use the information computed statically, along with information gathered during execution, to find and execute *dynamic program slices*. In previous work [32] [20] [4], a program slice is defined to be a subset of the program that behaves identically to the program with respect to a certain subset of its variables and instructions at a certain point in the execution. We define a *dynamic* program slice differently, as will be explained later in this chapter. We produce approximations to arguments of output operations by reexecuting the instructions in appropriate dynamic program slices using intervals of sufficiently high precision.

Previous work has been done in finding *static* program slices [32] [14] for use in debugging, for semantic comparison of similar programs, and for a kind of automatic parallelization of programs. Informally, a static program slice for an instruction consists of all the instructions in the program that *may* affect a given instruction in any execution of the program, as far as can be detected using only *static* information — that is, information independent of any execution.

Dynamic program slices have been studied by Korel and Laski [20] and Agrawal and Horgan [4] in the context of program debugging. Korel and Laski find static control dependences prior to execution, and use them to find dynamic data dependences as the program executes on a given input x; they then use an iterative algorithm over the dynamic dependences to compute dynamic program slices as required. They find a subset of the original program that can be executed to produce the same behavior on input x with respect to a chosen point in the execution and a chosen subset of the program's variables. Thus their strategy uses dynamic information, that is, data produced by a specific execution of the program, to find a set of instructions.

Agrawal and Horgan encode dynamic dependences into a *reduced dynamic dependence graph* (RDDG) that is constructed as the program executes on some input x. When the point of interest is reached in the current execution, they trace through the RDDG to find the instructions that affected the current instruction. The RDDG is similar to the DAGs discussed above, except that it only applies to the current point in the execution of the program — as the program executes, the RDDG is updated to give the instructions that affected the most recently executed instruction. The RDDG is designed to ensure that the size of the graph is bounded by the number of possible slices in the program; as the authors observe, without this type of structure, graphs of unbounded size could result if a new node was created every time an instruction was executed. Finding the dynamic slice amounts to tracing back along edges in the RDDG. Note that DAG-building methods *do* create a new node every time an instruction is executed that computes a new value, and that they do in fact grow to a size proportional to the number of instructions executed.

As did the method of Korel and Laski, Agrawal's and Horgan's work finds a subprogram of the original program that can be executed on the specific input x, and will behave the same as the original program with respect to a chosen point in the execution and a chosen subset of the variables of the program.

Our interest, however, is to find not only the specific instructions that may affect a certain value computed in a program execution, but also the *specific instances* of those instructions that affected the value of interest. There may be a very large number of instances of a given instruction in a program execution, only some of which may actually affect a given value. The approaches described by Korel and Laski, and Agrawal and Horgan, while well-suited to the task of debugging, do not provide this information on specific instances that recomputing exact real values efficiently requires.

The next section defines the programming language we analyze. The remainder of this chapter then presents a series of definitions building up to defining dynamic program slices. In the following chapter, we show dynamic program slices are actually computed and executed.

2.2 The Input and Intermediate Languages

This section describes the restrictions we place on the language used by the programmer, and the restrictions on the intermediate code representation on which our data-flow analysis is performed.

2.2.1 The Fortran Subset

The programs we accept as input to our analysis are written in a subset of the Fortran language. The following structures are included:

- Scalar and array assignments and expressions.

- Inductive do loops and while loops.
- if-then-else statements.
- Restricted use of subroutines.

We permit the use of scalar values, and of arrays, but require that there be no *aliasing* of variables; that is, only references to an unique variable name can affect the value or values associated with that name.

An *inductive do loop* is a loop of the form:

```
do i = M, N, S
     ...
enddo
```

The value of the variable `i` iterates from the integer value `M` to the integer value `N`; if `M` $\leq$ `N` then `S` must be greater than zero, and we say this is a *forward* loop; otherwise, `S` must be less than zero, and we call it a *backwards* loop. The body of the loop may contain no assignments to `i`. If `S` is omitted, it is assumed to be 1. Note that all iteration, that is, all instances where an instruction could be executed more than once, occurs due *only* to inductive do loops in our language subset.

While loops are allowed, and are transformed in the following way before any analysis is performed. A while loop is a loop of the form

```
while P do
     ...
enddo
```

The body of the while loop is executed repeatedly until the predicate `P` becomes **false**. For this form of loop we insert a loop counter `i` as follows:

```
do i := 0, ∞
     if not P then break
     ...
enddo
```

The name of `i` is chosen such that there are no references to `i` anywhere else in the program. Thus each iteration of the loop is identified with a unique value of the counter `i`. The `break` command causes execution of the loop body to terminate, passing control to the instruction immediately following the `enddo`.

An `if-then-else` statement is a statement of the form

$$\begin{array}{l} \texttt{if}\ P\ \texttt{then} \\ \qquad S_1, S_2, \ldots, S_n \\ \texttt{else} \\ \qquad R_1, R_2, \ldots, R_m \\ \texttt{endif} \end{array}$$

P is a predicate referencing only scalar values. If P evaluates to the Boolean value **true**, the instructions $S_1, S_2, \ldots, S_n$ are executed; otherwise, the instructions $R_1, R_2, \ldots, R_m$ are executed. The value of m may be 0.

The restrictions in this language, specifically the restrictions that the only iteration is due to do loops, lets us make the observation that if an instruction is reexecuted, it must be with a different set of values for the counters of the loops surrounding that instruction. This property of the language will be used below.

We allow calls to subroutines, including calls to Fortran primitive functions such as square root, exponential, and trigonometric functions; however, our analysis treats all subroutine calls as atomic operations, and performs no analysis on them.

2.2.2 The Intermediate Language

After the programmer inputs a program in the subset of Fortran described above, it is translated into an intermediate language. The intermediate language we use for our analysis and transformations is a modified form of the ILOC language called ILOC$_R$. ILOC is used in the Rice University $\mathbf{R}^n$ Project [9], and is designed to simplify data-flow analysis and code improvement transformations. ILOC$_R$ is a modified version of ILOC that includes some special instructions used only by our constructive real analysis.

Assignment statements in ILOC$_R$ take the following form:

$$\begin{array}{lcl} v_0 & \texttt{:=} & f(v_1) \\ v_0 & \texttt{:=} & g(v_1, v_2) \end{array}$$

The variables v_0, v_1, and v_2 may be of different types, as long as they match the definitions of f and g. For example, g could be addition of integers, or multiplication of an integer by an interval, or f could compare v_1 to 0 and return -1, 0, or 1 to indicate its result.

To use an array element, the element is loaded into a scalar variable using a separate load instruction:

$$v_j \ := \ A(s_1, s_2, \ldots, s_n)$$

A is an n-dimensioned array of type T, each of the s_i are integer variables, and v_j is a variable of type T. Once the value has been loaded, it can be used in an assignment statement. To store to an array, the value is first assigned to a scalar variable, and then a separate store instruction is used:

$$A(s_1, s_2, \ldots, s_n) \ := \ v_j$$

It should be clear that arbitrary Fortran assignment statements can be transformed into ILOC_R loads, stores, and scalar assignments. The restrictions described in this section are introduced only to make the data-flow analysis process simpler.

2.3 The Execution Scheme

Our strategy for computing approximations to constructive real values is based on variable-precision interval arithmetic. Interval arithmetic was discussed in detail in Section 1.4.1. Briefly, an interval is a pair $[a, b]$, where $a \leq b$, and an interval computation of precision p produces more accuracy, that is, a narrower interval, than a computation of precision $p' < p$.

Our scheme transforms the input program into a new program, of two parts. The first part, called the *main section*, contains every instruction from the original program, except that constructive real operations are transformed into interval arithmetic operations of a fixed precision p_d. The original program contains uses of constructive real functions specified by the programmer. In our scheme, the constructive real operations are transformed into interval operations (in the precision-function scheme, these constructive real operations would be converted into operations that build DAGs).

The second part of the new program, called the *slice section*, consists of several subroutines that are generated in such a way that they can use varying precision interval operations. For each output operation in the original program, there is a subroutine that computes approximations to the value of I's constructive real argument at an arbitrary precision, specified as an argument to the subroutine.

Recall that each output operation has an *accuracy requirement* that approximations to its constructive real argument must satisfy before the operation can be performed. Instructions are generated in the main program so that, when an output operation I is encountered, the accuracy of the approximation x to I's interval argument is checked against I's accuracy requirement. If x is accurate enough, I is executed and computation continues in the main section. If x is not accurate enough, the subroutine for I in the slice section is invoked. This subroutine executes a dynamic program slice that computes x using precision $2p_d$ instead of p_d. If the result of this calculation still does not satisfy I's accuracy requirement, the subroutine is invoked again, using precision $4p_d$. This doubling of the precision and reexecution of the subroutine continues until a result of sufficient accuracy, call it x_s, is produced. When that occurs, I is executed, using x_s instead of x. Then execution returns to the main section, again using precision p_d intervals until the next output operation is encountered.

The rest of this chapter consists of definitions that eventually lead up to a definition of *dynamic program slices*. The next chapter will go into how we actually produce the slice section, and how it is used to find and execute the appropriate dynamic program slices.

2.4 Definitions

In this section we give some basic definitions that we need to define program slices later in this chapter. Some of the definitions in this section are from Korel and Laski [20].

Definition 2.1 *If a program P is executed on some input x, a* trajectory *produced by P on x is a finite sequence T of instructions, where for each j, $1 \leq j \leq k$, the j^{th} element of T is the j^{th} instruction executed by P.*

If a trajectory T has k members, we write $|T| = k$.

Definition 2.2 *The elements of a trajectory are called* instruction instances. *The textual instructions themselves are called* instruction sites. *Instruction sites thus represent many possible instruction instances.*

A trajectory is just a "trace" of the program's execution on some specified input. Note that a trajectory does not necessarily correspond to the *complete* execution of the program — it may represent only a prefix of the program's execution.

In the remainder, we will use "instruction" or "site" alternately for "instruction site," and "instance" for "instruction instance," where there is no ambiguity, for the sake of brevity.

Definition 2.3 *Given a program P, a loop L is said to* contain *or* surround *another loop L' in P iff the entire text of L' is included within L in P. If L contains L' we write $L \supset L'$. Note that for any L and L', if $L \supset L'$, then L cannot contain L'.*

A sequence of loops $< L_1, L_2, \ldots, L_n >$ is a chain *iff $\forall i, 1 \leq i < n\,[\,L_i \supset L_{i+1}\,]$*[5].

Definition 2.4 *Given a chain of loops $V =< L_1, L_2, \ldots, L_n >$, a* value vector for V *is a vector $\vec{L} =< l_1, l_2, \ldots, l_n >$ where each l_i is an integer.*

If for a given trajectory T, there is a point in T at which loop L_i has the value l_i for $1 \leq i \leq n$, then $\vec{L}$ is valid for V in T.

Definition 2.5 *Given two vectors of loop counter values $\vec{L} =< l_1, l_2, \ldots, l_n >$ and $\vec{L'} =< l'_1, l'_2, \ldots, l'_n >$, for a chain of loops $L =< L_1, L_2, \ldots, L_n >$ in a program, we say $\vec{L}$* precedes *$\vec{L'}$ for L iff*

$$\exists j, 1 \leq j \leq n\,[\,\forall i, 1 \leq i < j\,[\,l_i = l'_i\,] \wedge l_i \lhd l'_i\,]$$

where $\lhd$ is $<$ for loops with positive step values, and $\lhd$ is $>$ for loops with negative step values.

An instance *of L in a trajectory T is given by a vector $\vec{L}$ if $\vec{L}$ gives only a value for each loop surrounding L. An* iteration *of L is an instance of L, plus a value that L takes on during that instance.*

Recall the restriction from the input language definition that the only iteration that may occur in a program is due to loop iteration. We can now state this property formally:

Lemma 2.1 *Given a program P that produces trajectory T on some input x, and any instruction site I in P, for each instance of I that occurs in T there is an unique*

[5]Note: the subscripts given the L_i in the sequence should not be taken to imply that the loops are necessarily consecutive in the original program.

vector of integers $\vec{L} =< l_1, l_2, \ldots, l_n >$ *such that the values of the chain of loops* $< L_0, L_1, \ldots, L_n >$ *surrounding* I *take on the values in* $\vec{L}$ *for that instance of* I*, and* only *that instance of* I *in* T*.*

Proof. Obvious from the restrictions placed on the input language. ∎

We denote the instance of an instruction site I that occurs when the loops surrounding I have values $\vec{L}$ as $I_{\vec{L}}$.

Note that our definition of paths ensures that they be *finite*. Clearly any trajectory must be finite as well. However, a trajectory may be a prefix of a non-terminating execution of P.

Korel and Laski denote a trajectory of a program by listing the index numbers[6] of the instructions in the order they are executed. For example, a trajectory for the program shown in Figure 2.1 is denoted

$$T =< 1, 2, 3, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, \ldots > \tag{2.1}$$

We shall denote the p^{th} member of a trajectory T by $T(p)$. If $T(p) = I$, we will refer to the instance of I at the p^{th} position in T as I^p. Note that if two instances I^p and J^r occur in a trajectory T that I^p occurs before J^r in T if and only if $p < r$.

It it important to note that an instruction instance is defined with respect to a *single* trajectory T; an instruction instance in another trajectory is a different entity.

We now give some definitions that presume the existence of a specific trajectory. In this sense these are *dynamic* definitions, since they are concerned with events that occur during program executions.

Definition 2.6 *Given some trajectory* T*, a* dynamic use *of a variable* v *in* T *is an instruction instance* I^p *in which the value of* v *is used. A* dynamic definition *of* v *in* T *is an instruction instance* I^p *that assigns a value to* v*.*

Note that this is not a *potential* use but an *actual* one; a dynamic use or definition of a variable is an instruction instance in a trajectory T, that is, a specific event in T, that uses or defines, respectively, a value of v.

These uses and definitions are different from those normally encountered in static analysis, with respect to arrays. Since a trajectory represents an actual execution,

[6]The index number of an instruction I is simply the number of instructions preceding I in the textual listing of P, plus one (the textually first instruction in P is thus number one, the second two, etc.).

```
1.    a := 0
2.    b := 1
3.    for i := 1 to 5
4.         c[i] := a + b
5.         print c[i]
6.         a := b
7.         b := c[i]
      end
8.    print a + b
```

Figure 2.1 A simple program.

the subscript values at each array reference are known. Thus a use or definition of an array element involves *only* that element at a particular instruction instance, whereas in static analysis an instruction is usually assumed to reference the entire array. Even with sophisticated static analysis, array references in general cannot be limited to a single element — statically they must be considered to reference at least some subset of the array's elements. But using dynamic slicing, array elements are effectively separate variables.

We now present some more dynamic definitions.

Definition 2.7 *We denote the set of non-loop variables used by an instruction* instance I^p *in some trajectory as* $U_D(I^P)$*, and the set of non-loop variables defined by* I^p *as* $D_D(I^p)$*. For an instruction site* I*, we denote the set of variables that may be used by* I *in any trajectory as* $U_S(I)$*, and the set of all variables* I *may define in any trajectory as* $D_S(I)$*.*

Note that $U_S(I)$ may only contain more than one variable when I is a store to an array, given the restrictions on the language. Furthermore, it should be obvious that for any instances I^p of an instruction I in a trajectory T, $U_D(I^p) \subseteq U_S(I)$, and $D_D(I^p) \subseteq D_S(I)$.

Loop counter variables are explicitly excluded from Definition 2.7, because we treat them as special values. The way we deal with loop counter variables will be discussed in Chapter 4.

For example, in the program in Figure 2.1, the sets for the dynamic case are:

$$\begin{aligned} U_D(1^1) &= \emptyset & \qquad D_D(1^1) &= \{a\} \\ U_D(4^4) &= \{a, b, i\} & \qquad D_D(4^4) &= \{c[1]\} \\ U_D(4^8) &= \{a, b, i\} & \qquad D_D(4^8) &= \{c[2]\} \end{aligned}$$

While the sets for the static case are:

$$\begin{aligned} U_S(1) &= \emptyset & \qquad D_S(1) &= \{a\} \\ U_S(4) &= \{a, b, i\} & \qquad D_S(4) &= \{c\} \end{aligned}$$

In this simple example, the only difference is for the array definition at instruction 4. In the dynamic case, only one array element is defined, while in the static case, the whole array is defined.

2.5 Dynamic Slices

Definition 2.8 *Let T be a trajectory of program P on input x. A* dynamic slicing criterion *of program P is a pair $C = (x, I^q)$, where I^q is an instruction instance in T.*

A dynamic slicing criterion specifies the point of interest in a program execution, and, by giving the input that causes P to follow trajectory T, the criterion implies a specific trajectory. We now define formally what it means for an instruction instance to depend on another instruction instance.

Definition 2.9 *Given a trajectory T, if $v \in D_D(I^p)$, $v \in U_D(I^p)$, if there is a subsequence of T that begins with I^p and ends with J^r, containing no intervening definitions of v, then there is a* dynamic data dependence *from I^p to J^r in T. We call I^p the* source *of the dependence, and J^r the* sink.

If J^r has a dynamic data dependence on I^p then we say that $I^p \in \mathrm{DD}(J^r)$, or that $I^p \,\mathrm{DD}\, J^r$.

Note that v may be either a scalar variable, or a single element of an array. It is important to understand that Definition 2.9 is concerned with instruction *instances*, not with instruction *sites*. *Static* dependences have been used in many areas, including compiler optimization [17] [22], automatic vectorization and parallelization of programs [12] [16], and debugging [32]. Static data dependences, which will be used in the next chapter, and concerned with what *may* occur; dynamic dependences are concerned only with what *has* occurred.

Definition 2.10 *Given an instruction instance J^r in a trajectory T of a program P, we define the set $\mathcal{D}_\mathcal{D}(J^r)$ to be the smallest set that satisfies the following:*

$$\mathcal{D}_\mathcal{D}(J^r) = \mathrm{DD}(J^r) \cup \{K^q \mid \exists I^p \in \mathcal{D}_\mathcal{D}(J^r)\ [\ K^q \in \mathrm{DD}(I^p)\,]\}$$

We call $\mathcal{D}_\mathcal{D}(J^r)$ the dynamic value group *for J^r in T.*

$\mathcal{D}_\mathcal{D}(J^r)$ is thus the transitive closure of the $\mathrm{DD}(\cdot)$ relation on J^r.

For example, consider again the program in Figure 2.1. Below are the $\mathrm{DD}(\cdot)$ sets for each assignment instruction instance in the trajectory T given by Figure 2.1.

$$\begin{aligned}
\mathrm{DD}(1^1) &= \{\} \\
\mathrm{DD}(2^2) &= \{\} \\
\mathrm{DD}(4^4) &= \{1^1, 2^2\} \\
\mathrm{DD}(6^6) &= \{2^2\} \\
\mathrm{DD}(7^7) &= \{4^4\} \\
\mathrm{DD}(4^8) &= \{6^6, 7^7\} \\
\mathrm{DD}(6^{10}) &= \{7^7\} \\
\mathrm{DD}(7^{11}) &= \{4^8\} \\
\mathrm{DD}(4^{12}) &= \{6^{10}, 7^{11}\} \\
\mathrm{DD}(6^{13}) &= \{7^{11}\} \\
\mathrm{DD}(7^{14}) &= \{4^{12}\} \\
&\ldots
\end{aligned}$$

We can then compute the dynamic value group for 4^8, $\mathcal{D}_\mathcal{D}(4^8)$, iteratively:

$$\mathcal{D}_\mathcal{D}(4^8) \;\rightarrow\; \mathrm{DD}(4^8)$$

$$
\begin{aligned}
&= \{6^6, 7^7\} \\
&\rightarrow \{6^6, 7^7\} \cup \mathrm{DD}(6^6) \cup \mathrm{DD}(7^7) \\
&= \{2^2, 4^4, 6^6, 7^7\} \\
&\rightarrow \{2^2, 4^4, 6^6, 7^7\} \cup \mathrm{DD}(2^2) \cup \mathrm{DD}(4^4) \\
&= \{1^1, 2^2, 4^4, 6^6, 7^7\}
\end{aligned}
$$

Definition 2.11 *An* environment *E is a set of pairs $(v, \bar{v})$ where each v is a variable and each $\bar{v}$ is a value, and for any variable v, there is at most one pair in E that has v has its left member. If $(v, \bar{v}) \in E$, then we say E gives v the value $\bar{v}$, or that $E(v) = \bar{v}$.*

If E and E' are environments, we say E refines *E' iff for all variables v*

$$(v, \bar{v}) \in E\,[\,\exists(v, \hat{v}) \in E' \;\wedge\; \bar{v} \sqsupseteq \hat{v}\,]$$

If E refines E' we write $E \preceq E'$.

Recall from Definitions 1.2 and 1.3 that $x \sqsupseteq y$ for interval x and y iff $x \subseteq y$, and for non-interval x and y, $x \sqsupseteq y$ iff $x = y$. If $E \preceq E'$, it means that every interval variable v given a value $\bar{v}$ by E is given a value by E' that is a superset of $\bar{v}$. This reflects our desire to compute more accurate interval approximations to real values (see Section 1.4.1). Also, if $E \preceq E'$, we know that all non-interval variables have the same values in E that they have in E'. Note E' may define other variables that are not defined in E, but that any variable defined by E must be defined by E'. Note also that the empty environment $\{\}$ refines all other environments.

Note that a variable in an environment can be either a scalar variable or a *single element* of an array.

We now define the effects of assignment instructions on environments.

Definition 2.12 *Let **E** be the set of all environments. Now suppose I is an instruction site in a program P of the form*

$$v_0 := f(v_1, v_2, \ldots, v_n)$$

where each v_i is a variable (the various v_i need not be distinct). Let E be an environment such that

$$\forall i, 1 \leq i \leq n\,[\,(v_i, \bar{v}_i) \in E\,]$$

Then we define the function $trans_I : \mathbf{E} \mapsto \mathbf{E}$ *as follows:*

$$trans_I(E) = \left\{ (w, \bar{w}) \;\middle|\; \bar{w} = \left\{ \begin{array}{ll} \hat{w} & w \neq v_0 \wedge (w, \hat{w}) \in E \\ f(\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_n) & w = v_0 \end{array} \right. \right\}$$

We call $trans_I(E)$ *the* environment produced by executing I on E. *If* $I_1, I_2, \ldots, I_n$ *are all instructions, then we call* $(trans_{I_1} \circ trans_{I_2} \circ \cdots \circ trans_{I_n})(E)$ *the environment produced by executing* $I_1, I_2, \ldots, I_n$ *on* E.

Definition 2.12 simply defines what it means to execute an assignment statement I by associating I with a function trans_I, and says to get the result of executing a sequence of instructions $I_1, I_2, \ldots, I_n$ on an environment E, we simply compose the trans_{I_j} functions and apply the result of the composition to E.

Lemma 2.2 *Suppose* f *is a continuous function of* n *interval arguments, Then given values* $u_1, u_2, \ldots, u_n$ *and* $v_1, v_2, \ldots, v_n$ *such that*

$$\forall i, 1 \leq i \leq n \, [\, u_i \subseteq v_i \,]$$

then

$$f(u_1, u_2, \ldots, u_n) \subseteq f(v_1, v_2, \ldots, v_n)$$

Proof. Recall Definition 1.1 from the previous chapter. From it we know that

$$f(v_1, v_2, \ldots, v_n) = \{ f(x_1, x_2, \ldots, x_n) \mid \forall i, 1 \leq i \leq n, x_i \in v_i \}$$

and similarly that

$$f(u_1, u_2, \ldots, u_n) = \{ f(y_1, y_2, \ldots, y_n) \mid \forall i, 1 \leq i \leq n, y_i \in u_i \}$$

If $y_i \in u_i$, then $y_i \in v_i$, by assumption. Thus every $y_1, y_2, \ldots, y_n$ such that

$$\forall i, 1 \leq i \leq n \, [\, y_i \in u_i \,]$$

must also satisfy

$$\forall i, 1 \leq i \leq n \, [\, y_i \in v_i \,]$$

Therefore every member of $f(u_1, u_2, \ldots, u_n)$ is also a member of $f(v_1, v_2, \ldots, v_n)$. ∎

With these definitions, we are now ready to define dynamic program slices. In essence, as we will discuss below, this definition defines a dynamic program slice as follows. Given a program P that produces trajectory T on input x, a dynamic program slice from P is a trajectory T' made up of some of the instances from T. Actually, it is slightly more involved — the instances in T' are identical to the corresponding instances in T, except that they may use more accurate values for interval arguments. We now give the formal definition:

Definition 2.13 *Let $C = (x, I^q)$ be a dynamic slicing criterion on program P, with T a trajectory of P on input x. Then a* dynamic program slice *of P on C is a* trajectory T' *such that for each instance $K^p \in \mathcal{D}_\mathcal{D}(I^q)$ in T, there is an instance K^q of K in T', and the value of each variable v used by K^q in T' is approximated by the value of v used by K^p in T.*

Previous definitions of dynamic program slices [20] [4] have defined them as collections of instruction *sites*, without specifying which instances of these sites are involved; they are dynamic in the sense that dynamic information is used to narrow the set of instruction sites involved in computing the values of interest. In contrast, the scheme described here defines dynamic program slices as sets of instruction *instances*; we use dynamic information to find sets of dynamic entities. Dynamic slices that are sub-programs make sense for debugging, but our interest is in efficiently recomputing a specific value, not in locating erroneous instruction sites.

In effect, a dynamic program slice on a particular instruction instance is a trajectory containing each instruction instance in the dynamic value group $\mathcal{D}_\mathcal{D}(I^q)$, where the instances in T' may use values of arguments that are more accurate than the values used by the corresponding instances in T.

There is no explicit restriction on the *results* of any of the instruction instances in T'; however, from Lemma 2.2 we know that, since the interval arguments of each instance K^p in T' are subsets of the arguments of the corresponding instance K^r in T, the result of K^p must be a subset of the result of K^r.

Note that our definition doesn't care how control-flow effects are reproduced, just that the assignment statements in the original program are reexecuted the same way in the dynamic slice that they were in T. In particular, it does not require that they be executed in exactly the same order in T' as in T.

Lemma 2.2 does not apply to discontinuous functions, including output operations. For example, consider the discontinuous function C_r defined as

$$C_r(x) = \begin{cases} 1 & L(x) > 2^r \\ -1 & R(x) < -2^r \\ 0 & \text{otherwise} \end{cases}$$

Now suppose x is an interval $[0, 3]$, that is, $L(x) = 0$ and $R(x) = 3$, and y is another interval $[0, 1]$, so that $L(y) = 0$ and $R(y) = 1$. Then $y \subset x$ but $C_1(y) \neq C_1(x)$. If y and x turn out to be values of some variable used in T' and T, respectively, then a conditional based on the C_1 function would branch to different locations in T' and T. Thus, we cannot simply reexecute output operations in the dynamic program slice to recreate their results there — this problem will be addressed in Chapter 4.

The next chapter discusses how the slice section of the program is constructed, and shows that this construction can be used as a dynamic program slice for the output instances in the main section. The following chapter will show how the slice section is executed to produce trajectories T' as in Definition 2.13.

Chapter 3

The Program Transformation

Recall from Section 2.3 that we transform the input program into a new program with two sections: the "main section" and the "slice section." In this chapter, we describe how most of the information used to construct these sections is computed.

The front end of the Fortran compiler we use produces a version of the original program in the intermediate language ILOC$_R$; we shall call this version of the program P_0. The ILOC$_R$ language consists of a set of pseudo-operations, representing stores to and loads from memory, arithmetic operations, comparisons of simple data types, and control-flow operations. Constructive real operations are denoted in P_0 using the set of pseudo-operations shown in Figure 3.1.

`RMUL x y z`	x := y $\times$ z
`RDIV x y z`	`x := y / z`
`RADD x y z`	`x := y + z`
`RSUB x y z`	`x := y - z`
`RSQR x y`	`x := sqrt(y)`
`RCON x k`	`convert rational value k into real x`
`ROUT x r`	`output x to r decimal digits` accurate to ± 1 in the last digit
`RCMP k x r`	k := 1 if x $> 2^r$ k := -1 if x $< -2^r$ `k := -1, 0, or 1 otherwise`
`RFLR x y r`	x := $\lfloor$y$\rfloor$ if $\lvert$y $- k\rvert > 2^r$ x := either $\lfloor$y$\rfloor$ or $\lfloor$y$\rfloor + 1$

Figure 3.1 Exact Real Pseudo Instructions.

The back end of the compiler knows how to transform most of ILOC$_R$'s pseudo-operations into assembly language operations for the machine's assembler. However, it has no knowledge of the constructive real pseudo-operations — we must transform

P_0 into a new ILOC$_R$ program P_1 that uses only the recognized ILOC$_R$ operations, that duplicates the functions of the constructive real pseudo-operations in P_0. The new program, P_1, will be made up of the main section and slice section.

We can easily transform the arithmetic operations `RMUL, RDIV, RADD, RSUB, RSQR,` and `RCON` into simple subroutine calls. The hard part of the transformation is translating the output operations: `ROUT, RCMP,` and `RFLR`. Note in the definition of these operations that they are not functions; that is, for a certain set of arguments, they may produce an unpredictable value. This is a result of the nature of constructive real values — as discussed in Chapter 1, simple comparisons such as comparing a constructive real value x to 0 are undecideable in general. Thus we must use modified forms of the operations we are accustomed to using on simpler data types, that use accuracy requirements to specify a tolerance within which to perform the operation. In these operations the accuracy requirements are specified as the `r` argument. Section 1.1 discusses this problem in detail.

P_1 consists of two parts. The first part is the "main section," which contains all of the instructions from P_0, except that the constructive real pseudo-operations are replaced with instructions that perform interval operations at a fixed precision p_d. For example, we replace each `RDIV` with code that takes two interval values and divides them to produce a new interval, and each `RCON k x` operation is replaced with instructions that create an interval representation of `k` of precision p_d and store that interval in `x`.

The second part of P_1 consists of a set of subroutines, collectively called the "slice section." For each output operation I in P_0 with constructive real argument x and accuracy requirement r, there is a subroutine S that computes an interval approximation to x using a specified precision. We put code in the main section before I that checks the precision p_d approximation of x against r; if the approximation is accurate enough, we perform the output operation on that interval[7]. If it is not accurate enough, we repeatedly invoke subroutine S with precision $2p_d$, then $4p_d$, then $8p_d$, and so on, until S produces an approximation to x whose accuracy satisfies the requirement indicated by r.

The next several sections discuss the form of the subroutines in the slice section of P_1, including how they are computed.

[7]We will demonstrate how we perform the different output operations on interval approximations later in this chapter.

3.1 Introduction

Our goal for the slice section is to generate a subroutine S_I for each output operation I, such that, when invoked, S_I reexecutes a *dynamic program slice*[8] for I using some specified precision p for its interval operations. By our definition of a dynamic program slice, this will result in recomputing an interval approximation to I's argument at precision p. We will then return that approximation to the main section, where it will check it against I's accuracy requirement, and either then execute I on the approximation, or invoke S_I again with a higher precision.

Our algorithm finds dynamic program slices using two sources of information: first, we compute static data dependences, to be used to approximate the dynamic data dependences, and use this static information to generate the slice section. This step is the focus of this chapter.

Second, we use the static dependences along with subscript values at array load and store instructions, current loop values, and some other information gathered while the program executes, to find the particular instruction instances that a given instruction instance has dynamic dependences on. In the next chapter, we will describe this computation.

Finding static data dependences is relatively straightforward. Many studies have been made of the computation, use, and meaning of static dependences [10] [12] [14] [16] [32]. We compute the static dependences using use-def chains [17] along with some information gathered about references to array elements to find data dependences.

The next section describes how we compute static data dependences. The following sections will discuss how they are used to generate the slice section. In the next chapter, we describe how we gather information during program execution, and how we use it to find and execute dynamic program slices.

3.2 Computing Static Data Dependences

We first define what a static data dependence is, using the $U_S(\cdot)$ and $D_S(\cdot)$ functions from Definition 2.7.

---

[8]See Section 2.5.

Definition 3.1 *A* program flow graph *G is a four-tuple (N, E, n_I, n_F), where N is a set of* nodes*, E is a binary relation on N, $n_I \in N$ is the* initial node *and $n_F \in N$ is the* final node*.*

We call the pairs $(a, b) \in E$ edges*, and say that a is a* predecessor of b. *and b a* successor of a.

A sequence of nodes $\mathcal{P} =< n_1, n_2, \ldots, n_k >$, where $(n_i, n_{i+1}) \in E$ for $1 \le i < k$ is called a path *in G. If $n_1 = n_I$ and $n_k = n_F$ then $\mathcal{P}$ is called an* program path *in G.*

To derive the program flow graph for a program P, we put an node in N for each instruction in P, and put an edge in E with each pair of instructions (I, J) where flow of control can pass directly from I to J. We assume without loss of generality that all programs terminate at the instruction associated with node n_F. G thus characterizes all possible executions of the program P.

Definition 3.2 *Given a program P, and a scalar variable v from that program, if $v \in U_S(J)$, and $v \in D_S(I)$, then J has a* static data dependence *on I iff there is a path in the program flow graph G from I to J that does not contain an intervening instruction site that assigns to v.*

Definition 3.3 *An instruction site J that may use one or more elements of an array A has a* static data dependence *on an instruction site I iff I may define one or more elements of A, and there is a path in G from I to J.*

Definition 3.4 *If there is a static data dependence from an instruction site I to another instruction site J, we call I the* source *of the dependence and J the* sink*, and say that $I \in \mathrm{DS}(J)$, or that $I \,\mathrm{DS}\, J$. The transitive closure of the $\mathrm{DS}(\cdot)$ function on an instruction site J is called the* static value group for J*; we denote it $\mathcal{D}_\mathcal{S}(J)$.*

Note that the definitions of static dependences do not depend on a specific execution of the program, as does the corresponding definition of dynamic data dependences. Also, unlike dynamic data dependences, we have to define static data dependences differently for scalar variable references and array references. This is because a given instruction *instance* can only use a single array element the corresponding instruction site references, while an instruction *site* represents the potential use or definition of many elements in an array.

Computing the *minimal* static value group for an instruction site I is in general undecideable; this follows easily from the result shown by Weiser [32] that the problem of computing the minimal (static) program slice is undecideable.

An important observation about static value groups is that, although $\mathcal{D}_{\mathcal{S}}(I)$ contains all of the instructions that may have instances in $\mathcal{D}_{\mathcal{D}}(I)$, there may be *many* instances of some instruction site $I \in \mathcal{D}_{\mathcal{S}}(I)$ in the dynamic value group $\mathcal{D}_{\mathcal{D}}(I)$. For example, in the program shown in Figure 3.2, we find that

$$\mathcal{D}_{\mathcal{S}}(3) = \{1, 2\}$$

but that the dynamic value group for the only instance of 3 in the trajectory shown in Figure 3.3 is given by

$$\mathcal{D}_{\mathcal{D}}(3^5) = \{1^1, 2^2, 2^3, 2^4\}$$

```
1    k := 0
     for i := 1 to 3 do
2        k := k + 1
     enddo
3    print the first n digits of k
```

Figure 3.2 A Simple Program Containing a Loop.

1^1 `k := 0`
2^2 `k :=` k_0 `+ 1`
2^3 `k :=` k_1 `+ 1`
2^4 `k :=` k_2 `+ 1`
3^5 `print the first n digits of` k_3

Figure 3.3 The Trajectory Produced by the Program in Figure 3.2.

We now prove an important lemma on the relationship between static data dependences and dynamic data dependences, and the corresponding theorem on the relationship between static and dynamic value groups.

Lemma 3.1 *Given a program P that follows trajectory T on input x, and an instruction I in P with instance $I^p \in T$,*

$$\forall J^r \in T\,[\,J^r\,\mathrm{DD}\,I^p \Rightarrow J\,\mathrm{DS}\,I\,]$$

Proof. If $J^r\,\mathrm{DD}\,I^p$ for some instruction instance J^r, then J^r defines either a scalar variable or an array element whose value is used by I^p in T. If a scalar variable v, then J clearly defines v, and by the definition of a trajectory (i.e., that it must follow a path in the program flow graph G in P), there is at least one path $\mathcal{P}$ from J to I in G. Furthermore, one of the paths from J to I must contain no definition of v; if such a definition existed, the value given v by J could not have reached I in any execution. Therefore if v is scalar, then, by Definition 3.2, $J\,\mathrm{DS}\,I$.

If J^r defines an array element whose value is used by I^p in T, then obviously I references the same array J defines elements of, and there must be a path from J to I in G, by the definition of trajectories. Therefore by Definition 3.3, $J\,\mathrm{DS}\,I$. ∎

Theorem 3.1 *Given a program P that follows trajectory T on input x, an instruction I in P with instance $I^p \in T$, and the dynamic value group $\mathcal{D}_{\mathcal{D}}(I^p)$ for I^p, for each instruction instance $J^r \in \mathcal{D}_{\mathcal{D}}(I^p)$, the corresponding instruction site $J \in \mathcal{D}_{\mathcal{S}}(I)$.*

Proof. Since $\mathcal{D}_{\mathcal{D}}(\cdot)$ and $\mathcal{D}_{\mathcal{S}}(\cdot)$ are defined as the transitive closures of DD and DS, respectively, this result follows easily from Lemma 3.1. ∎

Lemma 3.1 and Theorem 3.1 are important, and in fact are the basis of our method for finding dynamic program slices. Lemma 3.1 shows that static dependences *approximate* dynamic dependences, in a sense that *there can be no dynamic data dependence if there is no corresponding static data dependence.*

3.2.1 Computing Array Dependences

Since the definitions of scalar and array static data dependences are separate, we compute them separately. Finding the array data dependences is simple; all we must do is find each pair of instruction sites (I, J) where I may define an element of an array A, J may use an element of A, and I can possibly be executed before J. This requires only two things: first, a simple scan of the program to find which instructions may define or use which arrays. Note from the form of ILOC$_R$, our intermediate language, that any given instruction may only reference a single array, or define a single array. Second, given the program flow graph G with a set of instruction site nodes N and a set of edges E, we find the solution of the following set of simultaneous equations:

$$\mathrm{Pred}(J) = \{I \mid (I,J) \in E\} \tag{3.1}$$

$$\mathrm{AR}(J) = \mathrm{Pred}(J) \cup \bigcup_{I \in \mathrm{Pred}(J)} \mathrm{AR}(I) \tag{3.2}$$

The function $\mathrm{AR}(J)$ gives the set of instructions I for which there is a path in G from I to J. Note that this does not necessarily mean that there is some execution of the program that follows this path, but that is the a typical weakness of all forms of static analysis.

Systems such as the one defined by Equations 3.1 and 3.2 have been extensively studied for use in compiler optimization [18], and there are many ways to solve them. We use a simple iterative algorithm.

Note how imprecise a static array data dependence is with respect to possible dynamic data dependences on array elements. Consider the example program shown in Figure 3.4 (The `**` operation in the program indicates exponentiation). By Definition 3.3, there is clearly a static array data dependence from instruction 1 to instruction 2, since 1 defines elements of `A` and 2 references elements of `A`.

```
for i := 1 to 4 do
     A[i] := 2 ** i              1
enddo
for j := 1 to 4 do
     print j, A[j]               2
enddo
```

Figure 3.4 A Program Containing Array References.

However, consider the trajectory that would be produced by the program in Figure 3.4, as shown in Figure 3.5. Clearly there are dynamic data dependences *only* between instruction instances 1^1 and 2^5, 1^2 and 2^6, 1^3 and 2^7, and 1^4 and 2^8; there is *not* a dynamic data dependence from each instance of 1 to each instance of 2. A large part of our algorithm involves trying to find only the source instances a

```
A[1] := 2 ** 1        1^1
A[2] := 2 ** 2        1^2
A[3] := 2 ** 3        1^3
A[4] := 2 ** 4        1^4
print 1, A[1]         2^5
print 2, A[2]         2^6
print 3, A[3]         2^7
print 4, A[4]         2^8
```

Figure 3.5 Trajectory Produced by the Program in Figure 3.4.

given sink instance depends on, or at least the smallest number of source instances we can find cheaply. This will be covered in detail in the next chapter.

3.2.2 Computing Scalar Dependences

Finding scalar data dependences is somewhat more complicated, because with a scalar variable v we know that any instruction that assigns to v *kills* all previous definitions of v — thus we can compute static data dependences for scalars that are more precise than those we compute for arrays.

To find the scalar data dependences, we will compute a function reaches$(\cdot)$ [18] that, when applied to an instruction site J, gives us the set of instructions I that define scalar variables, and for whom there exists a path in G from I to J.

Definition 3.5 *A* basic block *is a sequence of instructions, consecutive in the original program, that are always executed start to finish.*

Definition 3.6 *Define* $\mathrm{defout}(b)$ *to be the set of instructions I in basic block b with result Y, such that Y is not defined by any instruction J occurring after I in b. Define* $\mathrm{defthru}(b)$ *to be the set of instructions I in any block in the program whose results are not defined by any instruction in b.*

Now we can define reaches$(\cdot)$ over all instructions in the program with the set of equations:

$$\mathrm{reaches}(J) = \bigcup_{I \in \ \mathrm{Pred}(J)} \mathrm{defout}(I) \cup (\mathrm{reaches}(I) \cap \mathrm{defthru}(I)) \tag{3.3}$$

Again, this system may be solved using any of several different methods, and we use a simple iterative algorithm. We can then define static scalar data dependences in terms of reaches($\cdot$):

Definition 3.7 *There is a static scalar data dependence from instruction site I to instruction site J iff I defines a variable v, J uses v, and $I \in reaches(J)$.*

The function reaches($\cdot$) is a well-understood one, studied extensively in compiler optimization [18] [17]. We observe that Definition 3.2 and Definition 3.7 are identical given that reaches($\cdot$) computes what it claims, which we assume.

3.3 Structure of the Slice Section

In the remainder of this chapter, we describe how we use the static data dependences to divide the instructions in the program into disjoint sets of instruction sites called *slice chunks*. We show how the static data dependences give rise to dependences between the slice chunks, and that we can execute each slice chunk separately, by first executing each other slice chunk on which it has a direct dependence. This result is the basis for our slice section execution strategy.

3.3.1 Segments

There are several considerations we must take into account when we output the slice section. First of all, we must satisfy the basic requirement that there be a subroutine for each output site that recomputes the output argument at arbitrary precision. We also wish to maximize *sharing* of intermediate values and minimize *duplication* of instruction sites.

As discussed in Section 1.3.2, constructive real values quite often have more than one "parent," that is, they are used in more than one expression in the program. A simple example of this phenomenon is that of a value applied repeatedly to a product, such as the example shown in Figure 3.6. In this example, the value of `x` is used `N` times in the DAG that makes up the value of `z[N]`, `N-1` times in the DAG for `z[N-1]`, and so on for each member of the `z` array.

Now suppose we execute the program containing the fragment in Figure 3.6, and that at the first instance of output site 4, the interval value for `z[1]` is not accurate enough to print out `n` digits. Then the subroutine for output site 4 is invoked, and

```
        ...
1       x := ...
2       z[1] := 1
        for i := 2 to N
3            z[i] := z[i - 1] * x
        enddo
        for j :  = 1 to N
4            print n digits of z[j]
        enddo
        ...
```

Figure 3.6 A Program Fragment Exhibiting Sharing.

`z[1]` is recalculated at precision $2p_d$. This will include the recalculation of `x` at precision $2p_d$, since `z[1]` has a dynamic data dependence on `x`.

Now suppose that the interval value for `z[2]` is insufficiently accurate when the second instance of 4 occurs. Now `z[2]` would need to be recomputed at precision $2p_d$. Since `z[2]` has a dynamic data dependence on `x` and `z[1]`, we also have to recompute their values. Thus `x` and `z[1]` get computed twice at the same precision — once for the first instance of 4, and once for the second instance of 4. the fact that those computations were just performed, since neither value was saved.

In general, if every `z[j]` must be recalculated, we will end up recomputing `x` $(N(N+1))/2$ times, and each `z[j]` will be recomputed $N-j$ times. It should be clear that we should only have to compute each value once at a given precision — thus the simple approach described in this example is clearly performing far too much extra computation.

Consider a program P with two output sites I_j and I_k. We compute the static value group for each: that is, we find $\mathcal{D}_{\mathcal{S}}(I_j)$ and $\mathcal{D}_{\mathcal{S}}(I_k)$. There will be some instructions that are only in the static value group for I_j, some only in the static value group for I_k, and some that appear in both static value groups. This last group, the intersection $\mathcal{D}_{\mathcal{S}}(I_j) \cap \mathcal{D}_{\mathcal{S}}(I_k)$, which we shall abbreviate as $\mathcal{D}_{\mathcal{S}}(I_j, I_k)$, contains instructions that calculate values used for recomputing the arguments to both I_j and I_k. Thus the two output sites give rise to three disjoint sets of instruction sites

$$\mathcal{D}_{\mathcal{S}}(I_j, I_k) \tag{3.4}$$

$$\mathcal{D}_{\mathcal{S}}(I_j) - \mathcal{D}_{\mathcal{S}}(I_j, I_k) \tag{3.5}$$

$$\mathcal{D}_{\mathcal{S}}(I_k) - \mathcal{D}_{\mathcal{S}}(I_j, I_k) \tag{3.6}$$

Thus (3.5) contains those instructions in only the static value group for I_j, (3.6) contains the instructions that appear only in the static value group for I_k, and (3.4) contains the instructions that appear in both.

If we generate a separate subroutine for each of these sets, and make provisions to store the values computed by (3.4) whenever they are computed and to retrieve them when needed by either the subroutine for (3.5) or for (3.6), we would ensure that no value was ever computed twice at the same precision. Furthermore, each instruction site in the program would appear only once in the slice section — in one of these three subroutines.

We give a formal definition of these disjoint sets:

Definition 3.8 *If O is the set consisting of all output sites in a program P, and $\{I_1, I_2, \ldots, I_k\} \subseteq O$, then we define the* segment *for $I_1, I_2, \ldots, I_k$ as the set $\mathcal{S}_{I_1,I_2,\ldots,I_k}$ of instruction sites such that for $k > 1$*

$$\mathcal{S}_{I_1,I_2,\ldots,I_k} = \{J \mid \forall K \in O\, [\, J \in \mathcal{D}_{\mathcal{S}}(K) \;\; \mathit{iff} \;\; K \in \{I_1, I_2, \ldots, I_k\}\,]\,\}$$

When only one output site is involved, we include it in the segment, thus:

$$\mathcal{S}_{I_j} = \{J \mid \forall K \in O\, [\, J \in \mathcal{D}_{\mathcal{S}}(K) \;\; \mathit{iff} \;\; K \in \{I_1, I_2, \ldots, I_k\}\,]\,\} \cup \{I_j\}$$

The reason for this inclusion will be discussed below. We call the segment associated with the single output site I_j the output segment *for I_j. We denote the set of all non-empty segments* **S**.

Thus $\mathcal{S}_{I_1,I_2,\ldots,I_k}$ contains all instruction sites that are in the static value groups for each output site I_j, $1 \leq j \leq k$, and for *no other* output sites. Note that in the preceding example, the set (3.4) is $\mathcal{S}_{I_j,I_k}$,

Applying this strategy to a program with three output sites I_1, I_2, and I_3, we could have a subroutine for each of the segments shown in Figure 3.7. In general, for a program with n output sites, there could be as many as $2^n - 1$ segments, since there is one set for each subset of O, the set of output sites in the program. This exponential term may seem alarming at first glance, but it is important to note that subroutines are not necessary for empty segments — therefore, the number of subroutines generated is bounded by the number of instruction sites in the program.

$$\mathcal{D}_{\mathcal{S}}(I_1, I_2, I_3)$$
$$\mathcal{D}_{\mathcal{S}}(I_1, I_2) - \mathcal{D}_{\mathcal{S}}(I_1, I_2, I_3)$$
$$\mathcal{D}_{\mathcal{S}}(I_1, I_3) - \mathcal{D}_{\mathcal{S}}(I_1, I_2, I_3)$$
$$\mathcal{D}_{\mathcal{S}}(I_2, I_3) - \mathcal{D}_{\mathcal{S}}(I_1, I_2, I_3)$$
$$\mathcal{D}_{\mathcal{S}}(I_1) - (\mathcal{D}_{\mathcal{S}}(I_1, I_2) \cup \mathcal{D}_{\mathcal{S}}(I_1, I_3) \cup \mathcal{D}_{\mathcal{S}}(I_1, I_2, I_3))$$
$$\mathcal{D}_{\mathcal{S}}(I_2) - (\mathcal{D}_{\mathcal{S}}(I_1, I_2) \cup \mathcal{D}_{\mathcal{S}}(I_2, I_3) \cup \mathcal{D}_{\mathcal{S}}(I_1, I_2, I_3))$$
$$\mathcal{D}_{\mathcal{S}}(I_3) - (\mathcal{D}_{\mathcal{S}}(I_1, I_3) \cup \mathcal{D}_{\mathcal{S}}(I_2, I_3) \cup \mathcal{D}_{\mathcal{S}}(I_1, I_2, I_3))$$

Figure 3.7 Potential Segments for a Program with Three Output Sites.

Calculating the segments for a program is simple. The algorithm is based on the fact that each disjoint set can be associated with a subset of the set of output sites in the program. We simply tag each instruction I_j with the names of the output sites whose static value groups I_j appears in; this tag indicates which segment I_j belongs to.

We mentioned above that part of this strategy involves saving results of some instructions. All segments $\mathcal{S}_0$, except those associated with singleton sets of output sites, will compute some values that need to saved, since there will be at least two other segments that depend on the instruction sites in $\mathcal{S}_0$. We will address the problem of saving intermediate results in the next chapter.

Consider the example program shown in Figure 3.8. Instructions 7 and 8 are the two output sites. The static value group for 7 is the set $\{1,2,4,5\}$, and the static value group for 8 is $\{1,3,4,6\}$. When we calculate the segments, we get the three segments

$$\begin{aligned} \mathcal{S}_{7,8} &= \{1,4\} \\ \mathcal{S}_7 &= \{2,5\} \\ \mathcal{S}_8 &= \{3,6\} \end{aligned}$$

Note that instruction sites 4 and 5 are in the same static value group $\mathcal{D}_{\mathcal{S}}(7)$, they are in different segments $\mathcal{S}_{7,8}$ and $\mathcal{S}_7$, respectively, and are in the same set of loops (the `i` loop).

Now suppose we naively generate a separate subroutine for of the segments, $\mathcal{S}_7$, $\mathcal{S}_8$, and $\mathcal{S}_{7,8}$. This would produce something like the three subroutines in Figures 3.9-

```
1    x := 0
2    y := 0
3    z := 1
     for i := 1 to N do
4         x := x + 1
5         y := y + x
6         z := z * x
     enddo
7    print n digits of y
8    print n digits of z
```

Figure 3.8 An Example Program.

3.11. Consider the subroutine for the segment $\mathcal{S}_{7,8}$. When called, it would compute all the values of `x`, but return only its final value, `x` = `N` — the other values of x are overwritten by the computation. Thus the subroutine for $\mathcal{S}_7$ would add the final value of `x` to `y`'s initial value of `0` `N` times; clearly this is incorrect. A similar problem would occur in $\mathcal{S}_8$. What we need to recompute `y`, for example, is a way to execute instructions 1 and 2, and then to alternate between instruction 4 and instruction 5 `N` times. In other words we need to be able to execute *parts* of these subroutines; we need to further divide the segments into something smaller.

3.3.2 Slice Chunks

Definition 3.9 *A* loop boundary *is a statement in a program that marks either the beginning of a loop or the end of a loop. A* program boundary *is either the beginning or the end of a program, an output instruction, or a loop boundary. We denote the* i^{th} *program boundary in a program as* B_i*.*

Consider a list of the instruction sites in a program in textual order; each sequence of instructions between two consecutive program boundaries is called a program chunk*. We denote the program chunk that begins after program boundary* B_i *with* C_i*. The program chunk between the beginning of the program and the first loop boundary is denoted* C_0*. We denote the set of non-empty program chunks as* $\mathbf{C}_P$*.*

```
        subroutine both
1          x := 0
           for i := 1 to N do
4                x := x + 1
           enddo
           return(x)
        end
```

Figure 3.9 Naive Subroutine for Segment $\mathcal{S}_{7,8}$.

```
        subroutine yonly
           x := call both()
2          y := 0
           for i := 1 to N do
5                y := y + x
           enddo
           return(y)
        end
```

Figure 3.10 Naive Subroutine for Segment $\mathcal{S}_7$.

```
        subroutine zonly
           x := call both()
3          z := 1
           for i := 1 to N do
6                z := z * x
           enddo
           return(z)
        end
```

Figure 3.11 Naive Subroutine for Segment $\mathcal{S}_8$.

An instruction site I is a member of a program chunk C iff either I occurs between the two program boundaries that delineate C, or I is an output site and I is the program boundary that ends C.

Denote the chain of loops surrounding a program chunk C as $L(C)$, that is, $L(C) =< L_0, L_1, \ldots, L_k >$ where $L_i \supset L_{i+1}$, for $1 \leq i < k$, and all the instructions in C are within the body of L_k.

The set $\mathbf{C}_P$ of program chunks is thus a partition of the set of non-loop statements in the program, that is, each instruction site in the program appears in exactly one program chunk. An example of a program partitioned into program chunks is shown in Figure 3.12.

```
B0            —
C0            total := 0.0
              —
B1            do i := 1, N
B2                 do j := 1, N
                         —
C2                       r[i, j] := 0.0
                         —
B3                       do k := 1, N
                              —
C3                            prd := a[i, k] * b[k, j]
                              r[i, j] := r[i, j] + prd
                              —
B4                       enddo
                         —
C4                       total := total + r[i, j]
                         —
B5                 enddo
B6            enddo
B7
```

Figure 3.12 The Program Boundaries in an Example Program.

Now, we combine the concepts of segments (Section 3.3.1) and program chunks to produce *simple slice chunks*:

Definition 3.10 *We define the set* $\mathbf{C}_S$ *as*

$$C \in \mathbf{C}_S \;\; \textit{iff} \;\; \exists C \in \mathbf{C}_P, \mathcal{S} \in \mathbf{S}\, [\, C = \{I \mid I \in C \cap \mathcal{S}\} \wedge C \neq \{\}\,]$$

That is, $\mathbf{C}$ *is the set of non-empty intersections of the segments and program chunks. We call the members of* $\mathbf{C}$ simple slice chunks *(abbreviated SSC). We denote the SSC that is the intersection of segment* $\mathcal{S}_{I_1,I_2,\ldots,I_k}$ *and program chunk* C_j *with* $c^j_{I_1,I_2,\ldots,I_k}$*.*

Note that the chain of loops around an SSC c *is the same as the chain of loops around the program chunk that* c *was created from.*

SSCs do not have all the qualities that we need to find and execute dynamic program slices. We must perform two transformations on the set of simple slice chunks to produce the set of slice chunks we will use. Before we make this transformation, we need to prove a lemma:

Lemma 3.2 *Suppose* A *is an array, that* I *and* J *are stores to* A*, and that* I *and* J *are each in the same program chunk* C*. Then either* I *and* J *are in the same SSC* c*, or* C *is contained in no loops.*

Proof. Suppose I and J are stores to an array A in a program chunk C, and that C is contained in one or more loops. Now suppose that I and J are in different SSCs. The presence of a loop around C means that there is a path in the program flow graph from I to J, and likewise a path from J to I. By the definition of simple slice chunks, if I and J are in different slice chunks, they must be in different segments, since they are by assumption in the same program chunk.

Now, by the definition of segments, if I and J are in different segments, there must be some load K that has a static data dependence on I and not on J, or a load L that has a static data dependence on J and not on I. Assume without loss of generality that the case is the former.

If a load K has a static data dependence on I, then there must be a path from I to K in the program flow graph. But since there is a path from J to I, there is obviously also a path from J to K, and thus K must have a static data dependence on J as well, a contradiction. Thus if I and J are in the same program chunk C, and they both store to an array A, and C is contained in one or more loops, I and J must be in the same SSC.

Note: if C is not contained in any loops, I and J may be in the different slice chunks. However, if C is contained in no loops, we know that each instruction in C may only be executed at most *once* in any trajectory produced by the program. ∎

Definition 3.11 *A definition I of a variable v has a* output dependence *on a definition J of the same variable iff there is a path from J to I in which v may not be defined.*

Output dependences [12] are used in parallelizing compilers to ensure that transformations that change the order of execution of definitions of the same variable preserve the semantics of the original program. If there are two definitions I and J of some variable v, and a use K of v that has a dependence on J, and the compiler moves the execution of I between J and K, then K will incorrectly use the value of v defined by I.

Our division of program chunks into simple slice chunks can cause the instructions in a program chunk to be executed in an order not consistent with the output dependences in the program. We perform two modifications to the set of simple slice chunks to avoid this possibility.

The first modification is simple: for each program chunk C where there exists stores I and J to an array A in C, such that I and J are in different slice chunks c and c', respectively, we merge c' into c. Recall from Lemma 3.2 that that each such program chunk occurs in a part of the program where there are no loops.

The second modification is more complex. It is performed by inspecting each program chunk C. C is examined for pairs of slice chunks c and c', $c \neq c'$, from C, where there is a path $\mathcal{P}$ of the form $< I, \ldots, K, \ldots, J >$ in C such that there is a variable v and

- I may define v.
- J may use v.
- K may define v, and there is a static data dependence from K to J.

Each such pair of slice chunks is merged into a single, new slice chunk. Note that v can either be a scalar variable or an array element (if v is an array element, then I and K are stores to the array, and J is a load).

We call the set of sets of instructions produced by performing these modifications on the set of simple slice chunks the set of *slice chunks*.

Definition 3.12 *We say there is an* inter-chunk dependence *from an SSC c to another SSC c' iff there exist instruction sites I and J, $I \in c$, $J \in c'$, where there is* *a static data dependence from I to J, and either $c \neq c'$, or the dependence (I, J) is loop-carried.*

For each such dependence (I, J), if v is the variable defined by I and used by J that causes the dependence, we say that v is exported *from c' and* imported *by c.*

If there is an inter-chunk dependence from c to c', we alternately say that c' has an inter-chunk dependence on c, or simply that c' has a dependence on c.

Slice chunks are the units of computation that will we use for computing interval approximations for output operations. Values within a program chunk that are used by only one other instruction are thus computed just as in the main program — the definition of the value is executed, and then the instruction that uses that value is executed. Because program chunks may contain conditional structures, values that are computed conditionally are included in the set of values that may be computed locally within a program chunk. The goal is to make as much of the computation as possible execute cheaply, with no more overhead than it would have if it were executed as a separate program, using intervals of the current precision.

The values corresponding to inter-chunk dependences are those values that either are used by either more than one other instruction, or that are simply located in different program chunks in the original program. Some of these variables must be saved when they are computed, so that when their values are needed by another slice chunk, they will not have to be recomputed. We call the variables whose values must be saved when they are computed *open values.*

Variables whose values are simply computed in one slice chunk c for use by another slice chunk c' are not necessarily open. If c is executed on demand from c' to compute a variable v's value, and no other slice chunk depends on c, v's value can simply be saved temporarily when it is computed, to be retrieved when the execution of c ends and the execution of c' begins. We perform a conservative analysis of inter-chunk dependences and find exported variables that need not be saved, that is, variables that are not *open.* The next chapter discusses how open values are saved when they are computed, and retrieved when required by another slice chunk.

In the remainder of this chapter, we shall describe our scheme for using slice chunks to compute higher-accuracy approximations to output arguments computed in the main section.

Definition 3.13 *Given a program P that produces trajectory T on some input x, a program chunk $C = I_1, I_2, \ldots, I_n$ in P, suppose there exist positions q and p in T such that $T(q-1) \notin C$, $T(p+1) \notin C$, and that*

$$\forall s, q \leq s \leq p, T(s) \in C$$

Then the range of instruction instances $T(q), T(q+1), \ldots, T(p)$ is called an instance of C in T, *and is denoted $C_T^{q,p}$.*

If $C_T^{q,p}$ is a program chunk instance in T, and $\vec{L}$ is the vector of loop values that the loops $L(C)$ around C have when $C_T^{q,p}$ occurs, then we may alternately denote $C_T^{q,p}$ as $C_{\vec{L}}$.

If c is a slice chunk from the program chunk C, then the set consisting of the instances in $C_T^{q,p}$ that are instances of instruction sites in c is called the instance of c at $\vec{L}$, *and is denoted $c_{\vec{L}}$.*

Note that, by the definition of program chunks, each instance of some program chunk C in a given trajectory corresponds to a different set of loop values for the loops surrounding C. That is, if C is contained in the chain of loops $L(C) =< L_1, L_2, \ldots, L_n >$, then each instance of C in a given trajectory occurs with a different vector of values $l_1, l_2, \ldots, l_n$ for the loops in $L(C)$ — the values of all loops are *constant* over a given program chunk instance.

Also, it is important to note that not all instruction sites in C will have a corresponding instruction instance in any given instance of C, because C may contain conditional instructions.

Definition 3.14 *Given a program chunk C from a program P that produces trajectory T on some input x, X_C to be a function that takes a pair consisting of a vector $\vec{L}$ of values for the loops $L(C)$ around C, and an instruction site I in C, where for each instance $C_{\vec{L}}$ of C in T*

$$X_C(\vec{L}, I) = \begin{cases} \text{true} & \textit{if there is an instance of } I \textit{ in } C_T^{q,p} \\ \text{false} & \textit{otherwise} \end{cases}$$

Note that X_C is computable, because by definition a trajectory is a path through the program that *has been executed*; thus for X_C to be defined on a program chunk instance, that instance must have already occurred, and X_C could be computed by saving a trace of each instance of C. In the next chapter, a relatively efficient algorithm for computing the X_C functions will be given.

Definition 3.15 *Given a slice chunk c from a program chunk C, that is, the* program for c *is the program consisting of the instruction sites in c in the order they occur in C. Note that the instructions in c need not be consecutive instructions in the program chunk.*

Let $\vec{L}$ be a vector of values of the loops surrounding C, that corresponds to an instance $C_T^{q,p}$ of C in T, and E be an environment. Then an execution of the program for c with loop value vector $\vec{L}$ with environment E *is a trajectory of instruction instances, produced by executing, in the order they occur in C, exactly those instructions I in c such that $X_C(\vec{L}, I) = \text{true}$, where for each instruction I for which X_C is* true*:*

- *For each loop counter variable referenced by I, the value given by $\vec{L}$ is used.*
- *For each non-loop counter variable referenced by I, the value given by current environment is used.*
- *After executing instructions $I_1, I_2, \ldots, I_j$, the current environment is given by*

$$(trans_{I_1} \circ trans_{I_2} \circ \cdots \circ trans_{I_j})(E)$$

Definition 3.15 is a directive on how to execute a slice chunk as a separate entity. Let T_c be the program trajectory produced by executing the program for c at some $\vec{L}$ with environment E. For each instruction instance I^p in T_c, we assume that E defines each variable v used by I^p, that is not defined by an instance in T_c that occurred before I^p.

We now define the environment E that such an execution needs to execute.

Definition 3.16 *Let P be a program that follows trajectory T on some input x. Let c be a slice chunk from a program chunk C, such that there is an instance $C_T^{q,p}$ of C in T.*

Let v be a variable whose value is $\bar{v}$ when it is used by an instruction instance I^r in $C_T^{q,p}$, I is a member of c, and ρ is the dynamic data dependence for v that has I^r as its sink. Then

- $(v, \bar{v}) \in E_P^q(c)$ *iff* the dynamic source of ρ occurred before $C_T^{q,p}$ in T
- $(v, \bar{v}) \in E_O^q(c)$ *iff* the dynamic source of ρ occurred in $C_T^{q,p}$, but was not an instance of a member of c.

Finally, let $E^q(c) = E_P^q(c) \cup E_O^q(c)$.

Thus the set $E^q(c)$ contains all variable-value pairs corresponding to dynamic data dependences whose sinks are instances of a member of c that occur in $C_T^{q,p}$, and whose sources are *not* instances of members of c in $C_T^{q,p}$. The values given by the environment $E^q(c)$ represent all the *external* information we need to execute the program for c at $\vec{L}$.

Theorem 3.2 *Let P be a program that follows trajectory T on some input x. Given a slice chunk c, from a program chunk C, such that there is an instance $C_T^{q,p}$ of C in T, let $\vec{L}$ be the set of loop values corresponding to $C_T^{q,p}$.*

Then if the program for c is executed at $\vec{L}$ starting with an environment E such that $E \preceq (E_P^q(c) \cup E_O^q(c))$, and this execution produces trajectory T_c, then each instruction instance J^r in T_c defining a variable v assigns to v a value $\hat{v}$ that is approximated by the value that the instance of J in $C_T^{q,p}$ in T assigned to v.

Proof. We give the proof by induction on the position of J^r in T_c.

Basis. If J^r is the first instruction instance in T_c, then all variables used by J^r must be given by E. By definition of E, the values it assigns must be approximated by the values used by the instance of J in $C_T^{q,p}$. Thus by Lemma 2.2, the value computed by J^r must be approximated by the value that was computed by the instance of J in $C_T^{q,p}$.

Induction Hypothesis. Assume for each instruction instance K^s that occurs at or before position n in T_c, that K^s computes a value that is approximated by the value the instance of K computed in $C_T^{q,p}$.

Induction Step. Suppose J^r occurs at the $(n+1)^{\text{st}}$ position in T_c. For each variable v used by J^r, either J^r uses a value of v computed by an instance that occurred prior to J^r in T_c, or J^r uses the value for v given by E. If J^r uses the value given by E, then by the same argument as that in the Basis, then J^r uses the value of v given by $E^q(c)$.

If J^r uses a value of v computed by an instance L^t in T_c, then by the Induction Hypothesis, the value L^t computes is approximated by the value computed by the instance of L in $C_T^{q,p}$. Thus if we show that the instance of J in $C_T^{q,p}$ used the value computed by the instance of L in $C_T^{q,p}$, we will have shown that J^r uses a value of v that is approximated by the value that the instance of J in $C_T^{q,p}$ in T used for v.

The only way the instance of J in $C_T^{q,p}$ could have used a value *other* than that computed by the instance of L in $C_T^{q,p}$, is if there is some instance K^s in $C_T^{q,p}$ that defines v, and K^s occurs after the instance of L in $C_T^{q,p}$, and before the instance of J in $C_T^{q,p}$.

If K^s occurs in that position in $C_T^{q,p}$, but J^r uses the value computed by L^t in T_c, then K must not be a member of c, since, by Definition 3.15, if K is in c, and is executed in $C_T^{q,p}$, then the execution of the program for c with the correct vector of loop values will include an instance of K in the same relative position.

Now, since K^s occurs after the instance of L in $C_T^{q,p}$, there must be a path from L to K in the program flow graph. Also, since the instance of J in $C_T^{q,p}$ occurs after K^s, there must be a path from K to J in the program flow graph. Thus there must be a path $\mathcal{P}$ in the program flow graph of the form $< L, \ldots, K, \ldots, J >$, such that L and K define v, L and J are in the same slice chunk c, and K is in some slice chunk $c' \neq c$.

But the second modification performed to produce slice chunks from simple slice chunks merged all pairs of slice chunks like c' and c. Thus there can exist no such K, and therefore the instance of J in $C_T^{q,p}$ must use the value of v computed by the instance of L in $C_T^{q,p}$. ∎

Recall in our definition of segments that we defined the output segment of an output site I_j to be the segment $\mathcal{S}_{I_j}$. We now make a similar definition for slice chunks:

Definition 3.17 *For each output site I_j, suppose I_j occurs in program chunk C_i.*

The slice chunk created from the output segment for an output instruction I_j and program chunk C_i is called the output chunk *for I.*

Now suppose we execute a program P on some input x, and it produces trajectory T up to the point of an instance of some output site in some slice chunk c. Let the instance of the program chunk containing this output site in T be $C_T^{q,p}$, and let $\vec{L}$ be the vector of loop values corresponding to $C_T^{q,p}$. If we could find an environment E that refined $E_P^q(c) \cup E_O^q(c)$, then by Theorem 3.2, we could execute the program for c at $\vec{L}$ with E, and compute the argument to the output operation with higher precision.

If the values in E are computed with higher and higher precision, by the definition of intervals in Section 1.4.1, the accuracy of the argument to the output operation will eventually satisfy its accuracy requirement. Thus if we can compute such an environment E, we will have given an algorithm that solves the central problem of constructive real arithmetic.

In the next chapter, we show how such environments are computed.

Chapter 4

Computing Dynamic Data Dependences

In this chapter we describe how we use the computations of Chapter 3 to find and execute dynamic program slices during an execution of the program.

As discussed in the previous chapter, in Section 3.3.2, before we may execute the program for a slice chunk c for a vector $\vec{L}$ of loop values with some precision p, we need to somehow compute an environment E such that, if $C_T^{q,p}$ is the program chunk instance corresponding to $\vec{L}$, E refines $E_P^q(c) \cup E_O^q(c)$, and the value for each interval variable in E has been computed with precision at least p[9].

The values in E can only come from the slice chunks upon which c has inter-chunk dependences. In this chapter, we show how a sufficient E is computed, by finding which instances of the slice chunks that c depends on computed values that c needs at $\vec{L}$; that is, we compute the sources of the dynamic dependences whose sinks are are c, but whose sources are not.

To find these slice chunk instances, we consider each inter-chunk dependence $\rho = (c', c)$, and the static dependences on instruction sites that caused ρ. If (I, J) is a dependence that caused ρ, then we wish to find if there is a dynamic data dependence from an instance of I to $J_{\vec{L}}$ in the current trajectory T, where $J_{\vec{L}}$ is the instance of J that occurred in the instance of c for loop values $\vec{L}$, and, if there is such a dynamic dependence, which instance of I is its sink.

There are four types of static data dependences, and we deal with each type separately. These four types are loop-carried array dependences, loop-independent array dependences. loop-carried scalar dependences, and loop-independent scalar dependences.

The first section explains, given a static dependence and an instance of the sink instruction, how to find the instance of the source instruction that the instance of the sink depends on. Knowing the instance of the source that must be executed is the

[9]That is, each operation used to compute the interval value had arguments that were computed with precision at least p, and constant values were had at least p significant digits.

same as knowing what instance of the slice chunk that contains that source must be executed. In the next section, we will show, given an instance of a slice chunk c, how we find the E environments for and execute the instances of the slice chunks that c depends on to get the values they define, so that we may execute the instance for c. The remainder of the chapter will then give some details about the bookkeeping operations required to perform these computations.

4.1 Finding the Dependences

4.1.1 Array Dependences

Some of the techniques described in this section are inspired by work in vectorizing and parallelizing compilers, such as [16]. These compilers analyze programs statically to find recurrence relations between subscript expressions of array loads and stores. The concern of vectorizing and parallelizing compilers is, put simply, to find which loop iterations another loop iteration depends on. This is the same problem we are trying to solve — to find which instance of a source a given instance of a sink depends on.

Each loop-independent array dependence indicates that there may be dynamic data dependences from instances of the source I to instances of the sink J in one or more trajectories produced by the program. If I is a potential store to an array A, and J is a potential load from A, then given an instance J^r of J in some trajectory T, we wish to find the closest instance I^p of I to J^r that occurred before J^r in T, such that I^p defined the *same element* of A that was used by J^r.

If I^p and J^r reference the same element of A, then each of I^p's subscripts much be equal to the corresponding subscript of J^r. Thus we wish to find the closest instance I^p before J^r in T such that the subscripts of I^p equal the subscripts of J^r.

For many store instructions, the loops around the store uniquely determine its subscript values. That is, given any vector $\vec{v}$ of subscript values of a store I, we can determine exactly the values that the loops around I must have had when I's subscripts had the values specified by $\vec{v}$.

For example, in the program in Figure 4.1, each value of the subscript `s` of I corresponds to a distinct value of the loop counter `k`. Thus if our sink instance is the instance of J that occurred when `l` $= l_0$, we can set `s` $= l_0$, and using the equation `s = i - 1`, solve for `k` $= l_0 + 1$.

```
            ...
            do k = 1, N
                 s := k - 1
I                a[s] := ...
            end
            do l = 1, N
J                ... := a[l]
            end
            ...
```

Figure 4.1 The Subscript of I Uniquely Determines a Value of `k`.

When a vector of subscript values $\vec{v}$ occurs for more than one vector of loop values around a store I, we must calculate the most recent occurrence of I with subscripts $\vec{v}$ before the instance J^r of the sink J in a different way. With some simple rules, this is easy to compute:

1. Determine all loop values possible by using the subscript values at the sink along with equations given by the program. Any constraints determined by subscripts are noted.

2. Any loops that the sink and source have in common retain the values they had around the sink instance, as long as this does not contradict any values given by Rule 1.

3. Consider the set of loops around the source that are not around the sink. From the outermost to the innermost, assign to each loop not yet assigned a value its *most advanced value*, given the values of the loops surrounding it, as long as this does not contradict any values given by Rules 1 or 2.

If a loop L_j is contained in loops $L_1, L_2, \ldots, L_{j-1}$, then the *most advanced value* for L_j given values $l_1, l_2, \ldots, l_{j-1}$ for the loops around L_j is the largest[10] value ever reached by L_j's counter variable in the trajectory when $L_i = l_i$ for $1 \leq i < j$.

Rule 1 matches sink subscript values to loop varaible values to generate equations. In many instances, including the simple example in Figure 4.1, these equations are

[10]If L_j has a negative step size, it is the *smallest* such value.

all the information that is required to find values for the loops around the source. However, in some cases, we will only be able to derive constraints. For example, in the program shown in Figure 4.2, for the sink with $\mathtt{r} = r_0$, using Rule 1 we can derive only the constraints

$$\mathtt{k} + \mathtt{l} \leq 2\mathtt{N} \tag{4.1}$$

$$\mathtt{k} + \mathtt{l} = r_0 \tag{4.2}$$

$$\mathtt{k} \geq 1 \tag{4.3}$$

$$\mathtt{l} \geq 1 \tag{4.4}$$

```
        ...
        do k = 1, N
            do l = 1, N
I               a[k + l] := ...
            end
        end
        do r = 1, 2N
J           ... := a[r]
        end
        ...
```

Figure 4.2 More Complicated Subscripts.

This situation is an example of where Rule 3 is applied. From Definition 2.5, we know that given two instances of a loop body, the instance that has a greater value of its outermost loop counter occurs later in a trajectory. Since the `k` and `l` loops have finished iterating before an instance of J in Figure 4.2 occurs, we know then that the closest instance of I to an instance of J with $\mathtt{r} = r_0$ will have the greatest value for `k` among all the instances of I whose subscript equals r_0. In other words, we set `k` as follows:

$$k = \begin{cases} r_0 - 1 & s_0 \leq N \\ N & s_0 > N \end{cases}$$

and set `l` to:

$$l = \begin{cases} 1 & s_0 \leq N \\ s_0 - N & s_0 > N \end{cases}$$

Lemma 4.1 *Given a static, loop-independent array data dependence* (I, J)*, and an instance* J^r *of* J*, the algorithm given by Rules 1 through 3 finds the instance* I^p *of of* I *closest to* J *in* T *such that* I^p *has subscript values equal to the subscript values of* J^r*.*

Proof. Suppose there is an instance I^s of I with the same subscript values as I^p and J^r, and that $p < s < r$, that is, I^s occurs after I^p and before J^r in T.

Then the values of the loops around I at I^s are a more advanced vector than the loops at I^p. By Definition 2.5, there must be a loop L_i around I such that the value of L_i's counter at I^s is greater than[11] the value of L_i's counter at I^p, and that the loops containing L_i at I^p have the same values as the loops containing L_i at I^s. But Rule 3 guarantees that we selected the most advanced value of L_i's counter given the values of the loops containing L_i when we derived I^p. Thus I^s cannot occur after I^p in T, a contradiction. ∎

It is not feasible to try to solve for the loop counters for all subscript expressions. Our implementation only attempts algebraic solutions for *linear* subscript expressions, that is, expressions of the form

$$C_1 v_1 + C_2 v_2 + \cdots + C_n v_n$$

where each C_i is an integer constant, and each v_i is a loop counter variable[12].

Definition 4.1 *Given a static array data dependence* (I, J)*, any loops whose values can be determined using Rules 1 through 3 for instances of* J *are called* bindable for (I, J)*. Any loop that is not bound for* (I, J) *is called* unbindable for (I, J).

For more complex expressions, we do not try to solve the equations explicitly. Instead, we record enough information as the main program executes to enable us to *loop up* the loop values corresponding to the most recent instance of a source with a given set of subscripts.

[11] If the step size of L_i is negative, then the value of L_i's counter at I^s must be les than the value of L_i's counter at I^p.

[12] We perform a simple transformation on the program prior to this analysis that finds all induction variables [18] for each loop and unites them into a single loop counter variable.

If (I, J) is an dependence on array A with one or more unbindable loops, then we create an array β_I with one dimension for each subscript of I, that is, an array of the same dimension as A. The elements of this array are vectors containing the values of the loops around I. Given a vector $\vec{s} = s_1, s_2, \ldots, s_n$ of subscript values specifying an element of A, $\beta_I(\vec{s})$ contains the vector of loop values specifying the last instance of I that wrote to $A(\vec{s})$, or a special value $\perp$ if no such instance of I has yet occurred.

Note that the elements of β_I change as the main program section executes. Every time an instance of the instruction I is executed with subscripts $s_1, s_2, \ldots, s_n$, the information stored at $\beta_I(s_1, s_2, \ldots, s_n)$ is changed. Note that only one β array is created for any given source.

Now consider a sink J that has no loops in common with the source. Then to find the instance of I that an instance J^r of J has a dynamic data dependence on, we can simply look up the element of β_I corresponding to the subscript values used for A at J^r. Note that J^r may not be the most recent instance of J in the main program — since J and I have no loops in common, however, no instances of I can ever occur after the first instance of J, and thus β_I never changes after the first instance of J.

When I and J have one or more loops in common, however, the situation becomes slightly more complicated. Suppose I and J have one or more surrounding loops in common, and that at some point in the execution of the program, the main program invokes a slice chunk, which in turn invokes an slice chunk containing an instance J^r of J. Now, since I and J have loops in common, it is possible that some instances of I occurred after J^r, and thus that $\beta_I(\vec{s})$ now contains a vector corresponding to one of these instances of I. In other words, if J^r has subscript values $\vec{s}$, $\beta_I(\vec{s})$ no longer gives the closest instance of I that occurred before J^r with the same subscripts as J^r.

We solve this problem by doing a little more bookkeeping. Every time an instance J^r of J occurs in the main program with subscript values $\vec{s}$, we look up $\beta_I(\vec{s})$ and store it, thus saving a kind of "pointer" to the instance of I that J^r must depend on. Then whenever J^r occurs later in an slice chunk, we simply look up the stored instance of I to compute the dynamic data dependence.

This strategy requires an array the size of A in which to store loop value vectors, and, if I and J have any loops in common, the ability to store a vector of values for the loops around I for *each instance* of J. This may seem like an excessive amount of storage, until one considers the storage required by the DAG-building methods. DAGs use a graph node for each instance of *every* instruction instance, and an arc between each pair of graph nodes for *every* dynamic data dependence. Our strategy for complex subscript expressions can be viewed as constructing a subset of the DAG-structures that would be created using one of the strategies in Chapter 1.

The most common situation that gives rise to unbindable loops occurs when a subscript's value depends on the result of a comparison. For example, in the program shown in Figure 4.3, the value of the subscript `sub` at I is not a linear function of the loops surrounding it (i.e., not a linear fuction of the `k` loop). We create an array β_I of one dimension and length `N`.

Initially, every element of β_I is set to $\perp$, since no instances of I have yet occurred. Each time I is executed with `sub` $= s$, $\beta_I(s)$ is set to the current value of `k`. Furthermore, since J depends on I, and J and I have a loop in common (the `k` loop), every time J is executed with `l` $= l_0$, we save the current value of $\beta_I(l_0)$.

```
      ...
      do k = 1, N
           max := a[k]
           sub := 1
           do l = k + 1, N
                if a[l] > max then
J                    max := a[l]
                     sub := l
                endif     end
           t := a[sub]
I          a[sub] := a[k]
           a[k] := t
      end
```

Figure 4.3 Sort Array Elements Into Descending Order.

Loop-carried array dependences can be dealt with exactly the same way as loop-independent array dependences, except that we modify Rule 2:

2'. If sink and source have loop L in common, then if L is the carrying loop for the dependence, its value at the source is one step less than its value at the sink instances. If L is not the carrying loop, its value is retained, as long as this does not contradict Rule 1.

Other than this change, the algorithm and proofs for loop-carried array dependences are identical to those for the loop-independent case.

4.1.2 Scalar Dependences

Scalar dependences are simply array dependences where the arrays have zero dimensions. The rules for scalar dependences are simplified, as follows:

1. Any loops the sink and source have in common have the same values at the source instance that they have at the sink instance.

2. Consider the set of loops around the source that are not around the sink. From the outermost to the innermost, assign to each loop not yet assigned a value its *most advanced value*, given the values of the loops surrounding it, as long as this does not contradict any values given by Rule 1.

The rule for subscripts is deleted, since clearly scalar variables have no subscripts to worry about.

Note that no special information is required for scalar dependences, as it was for unbindable loops in the array dependence case. There are no unbindable loops in the scalar case.

The loop-carried scalar dependence case follows similarly — the carrying loop is assigned the value of the last iteration prior to the iteration containing the sink instance, and otherwise the Rules are applied as above.

4.2 Slice Chunk Instances

In this section, we describe our strategy for executing slice chunk instances. The problem is, given a vector $\vec{L}$ of values given values for the loops around an slice chunk c, corresponding to a particular program chunk instance $C_T^{q,p}$ in the current trajectory T, to execute slice chunk instances that the instance of c specified by $\vec{L}$ depends on to compute the values that c needs. In the last section, we explained how to find the loop vectors for each of the slice chunks that c has a static dependence on; in this section, we explain how the values used by the slice chunks c depends on are saved and used to build the environment that we require to execute the program for c at $\vec{L}$.

4.2.1 Execution Order

A given slice chunk c will, in general, have inter-chunk dependences on a set of slice chunks. Given a set of loop vector values specifying the values of the loops around around c, we wish to build an environment E such that E refines $(E_P^q(c) \cup E_O^q(c))$.

We do this by executing the instances of the c_i that the instance of c depends on, and accumulating their results. When we have executed all the instances of the c_i slice chunks that the instance of c depends on, we will have computed all the values of variables in such an environment E.

The problem we must address in this section arises when instances of two different c_i define the same variable v. Recall from Section 3.3.2 our definition of *output dependences*, Definition 3.11. In that section, we modified the set of simple slice chunks to produce the set of slice chunks, in order to ensure that output dependences *within* slice chunks are always respected. In this section we address a similar problem for instructions in different slice chunks.

In Section 3.3.2, we used output dependences as the basis for our discussion. In this section, we use a relaxation of output dependences, call a *def-order dependence*[10].

Definition 4.2 *An instruction I that may define a variable v has a* def-order dependence *on another instruction J that may define v, iff there is a path from J to I in which v may not be defined, and there is an instruction K that may use v such that K has a static data dependence on each of I and J.*

If there is a def-order dependence from a definition I of v to a definition J of v, we write $J \vdash^v I$.

Def-order dependences are less strict than output dependences, since they require the presence of a sink that both definitions are sources for. We are restricted to the use of output dependences for definitions in the same program chunk due to the way we reexecute slice chunks. But when we execute slice chunks from other program chunks, we can use def-order dependences.

Note the requirement in Definition 4.2 that there be a path from J to I in which v may not be killed rules out pairs of definitions that a sink K may depend on, that can never both be executed (e.g., if I and J were in alternate arms of an if-then-else construct, and neither I's nor J's definition can reach the if-then-else, there would be no def-order dependence). If v is a scalar variable, there must be a path that contains no instruction site that assigns to v; if v is an element of some array A, there must be a path that either contains no stores to A, or in which it cannot be determined that v is definitely killed.

We now extend the definition of def-order dependences to slice chunks.

Definition 4.3 *A slice chunk c that contains the source I of a dependence on some variable v has a* def-order dependence *on another slice chunk c' that contains the source J of a dependence on the same variable, iff there is a slice chunk $c'' \neq c'$ such that c'' depends on c for v, and c'' depends on c' for v, and there is a def-order dependence from I to J.*

If there is a def-order dependence from a slice chunk c' to a slice chunk c'' arising from definitions of variable v, we write $c'' \vdash^v c'$.

Definition 4.3 is similar to Definition 3.12 from the last chapter; that definition extended data dependences to slice chunks, and Definition 4.3 extends def-order dependences to slice chunks. Note that since inter-chunk dependences can involve many variables, the definition specifies that the I and J instructions define the same variable.

Definition 4.4 *For each slice chunk c, and each variable v imported by c, we define $\Psi(c, v)$ to be the set of slice chunks that export v for c.*

Lemma 4.2 *For each slice chunk c, and each variable v imported by c, the inter-chunk def-order dependence relation $\vdash^v$ between the slice chunks containing the instructions in $\Psi(c, v)$ is a* quasi order *on $\Psi(c, v)$.*

Proof. To show $\vdash^v$ is a quasi order, we must show it is irreflexive and transitive [29].

Irreflexivity. A relation R is irreflexive iff for all x, $x \not R\, x$. By our Definition 4.3, two slice chunks must be distinct to have a def-order dependence between them.

Transitivity. A relation R is transitive iff for all x, y, and z, $xRy \wedge yRz \Rightarrow xRz$. Consider three slice chunks c_0, c_1, and c_2, such that $c \vdash^v c_1$ and $c_1 \vdash^v c_2$. Then there must be instructions I, J, and K in c_0, c_1, and c_2, respectively, such that $I \vdash^v J$ and $J \vdash^v K$.

Since $I \vdash^v J$, we know that there is a path $\mathcal{P}_\infty$ from I to J in v may not be defined. Similarly, since $J \vdash^v K$, there is a path $\mathcal{P}_\in$ from J to K in which v may not be defined. The concatenation of these two paths $\mathcal{P}_\infty; \mathcal{P}_\in$ is a path from I to K in which v may not be defined. Thus $I \vdash^v K$, and therefore $c_0 \vdash^v c_2$. ∎

Because $\vdash^v$ is a quasi order, we can *sort* the slice chunks in $\Psi(c, v)$ into an order consistent with $\vdash^v$ for each variable v imported into c. In the next section, we use this property

4.2.2 Executing Slice Chunk Instances

One at a time, for each inter-chunk dependence that has c has its sink, we compute the vector of loop values for each source slice chunk c', that is, we compute the instance of c' that the current instance of c depends on. The algorithm for executing a particular slice chunk instance is shown in Figure 4.4 where c is a slice chunk from program chunk C, and we wish to compute the environment with which to reexecute the program for c corresponding to an instance $C_T^{q,p}$ of C in the current trajectory T. We refer to the instance of c as $\bar{c}$. The algorithm takes $\bar{c}$ as its argument, computes an environment E such that $E \preceq (E_P^q(c) \cup E_O^q(c))$, and executes the program for c with the appropriate loop values using the environment E.

In the subroutine `BuildEnvironment`, note that the values of a variable v are added to that export a particular variable v are executed in an order consistent with the def-order dependences between them. This ensures that the proper definition of v is the last one added to E. The reason we must go through the list of source slice chunks is that they may not all have occurred; for any given $i, 1 \leq i < n$, we do not know whether any of the remaining c_j for $j > i$ contained an instance of the definition of v. This information could be computed in advance, by finding all the instances of the slice chunks that c depends on for v, but this computation could be as expensive as simply executing the slice chunk instances as we find them. In future work, it is worth investigating the relative merits of each approach.

Some important details of the execution of slice chunks are not shown in the algorithm in Figure 4.4. Recall from Section 3.3.2 the definition of *open values*; as each slice chunk instance is executed, all of open values must be saved. Then in `GetValue`, if a slice chunk instance $\bar{c}$ has already been executed with precision p or higher, the stored value can be retrieved without recomputing it. The remainder of this chapter discusses how these values are stored, and also how conditional instructions are dealt with.

4.3 Finding Most Advanced Values of Loops

Many of our rules listed above require that we find the most advanced value of a loop, given values for each of its surrounding loops. In this section we show how these computations are performed.

Before we analyze loops, we first perform a simple symbolic analysis on the variables in the program. This analysis consists of two passes; the first pass finds the

```
ExecuteSliceChunk(c̄, v)
   E := BuildEnvironment(c̄, p)
   let L⃗ be the vector of loop values for c̄
   execute the program for c at L⃗ with environment E
end

BuildEnvironment(c̄, p)
   E = {}
   for each variable v imported by c
      let c_1, c_2, ..., c_n be the sources for v, s.t. c_i ⊢^v c_{i+1}
      for i := 1 to n
         find the instance c̄_i of c_i that c̄ depends on
         if c̄_i has been executed in T then
            Delete any definition of v from E
            E := E ∪ {(v, GetValue(v, c̄_i, p))}
         fi
      rof
   rof
   return(E)
end

GetValue(v, c̄, p)
   if c̄ has not been executed with at least precision p then
      ExecuteSliceChunk(c̄, p)
   fi
   v̄ := value of v stored by c̄
   return(v̄)
end
```

Figure 4.4 Constructing Environments.

values that would be assigned to each variable if exactly one instance of each loop were executed. It also takes into account all values that could be assigned by different arms of **if** statements. The second pass analyzes the effects loop iteration could have on variable values.

Variable values are treated as combinations of loop counter variables and constants; any values that incorporate other types of values are marked as COMPLEX. COMPLEX variables include all non-integer variables, variables that may be assigned different values according to the results of comparisons, variables that may be assigned integer expressions that are not linear combinations of loop counter variables, and any variables that may be assigned expressions referencing other COMPLEX variables. Our analysis is thus very conservative.

After we do this very simple symbolic analysis, we look at the variables that give the bounds for each loop[13]. In many cases, these variables always contain the same constant value; for example, back in Figure 4.3, the bounds for the `k` loop are the constants `1` and `N`. Another common occurrence is exemplified by the `l` loop in that figure; the lower bound for the `l` loop is a simple function of the value of a surrounding loop.

When the variable giving the upper[14] bound of a loop L is not COMPLEX, we can determine its most advanced value given the values of the loops around L. At each point in the slice section where a most advanced value needs to be calculated, we insert code to perform the appropriate function.

When the upper bound variable is COMPLEX, we must record the most advanced value (MAV) for each different vector of values of the loops around L; that is, we must store the MAV of L for each instance of L. Thus, each time we start a new instance of L, that is, each time we reach the "do" statement in a trajectory, we record the value of its upper bound. Then when we need the MAV for L at some point in the slice section, we look up the saved value for L's MAV given values for the loops surrounding L.

[13]Although our examples are written in a pseudo-Fortran for clarity, recall that our analysis is all done on the intermediate language ILOC_R, and that all operations in ILOC_R are performed on virtual registers (i.e., variables). Thus even constants are copied into registers before comparisons (such as the comparison prior to each iteration of a loop) are performed.

[14]We use "upper bound" to mean the bound on the most advanced value of a loop, whether or not the loop's step size is positive.

This is similar to the storage we use to record source instances for unbindable loops, when the sink and source have loops in common. Note however that no β array is necessary for this case.

4.4 Decision Operations

As we explained in Chapter 1, comparison of exact real values is in general undecideable. We avoid this problem by computing comparisons using a tolerance. We denote the right endpoint, that is, the larger midpoint, of an interval z by $R(z)$, and the left endpoint by $L(z)$. Using this notation, we give the interval comparison operator *compare* as:

$$\text{compare}(x, y) = \begin{cases} -1 & \text{if } R(x) < L(y) \\ 1 & \text{if } L(x) > R(y) \\ 0 & \text{otherwise} \end{cases}$$

Figure 4.5 shows examples of the three cases enumerated in the definition of compare.

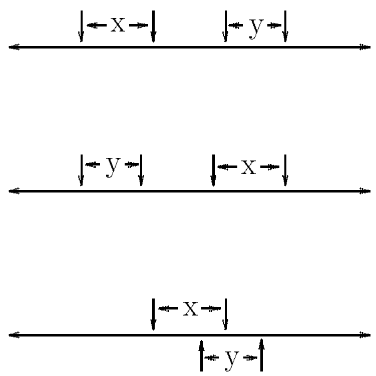

Figure 4.5 Comparison Cases.

Clearly, the result of a given invocation of the comparison function depends on the width of the interval arguments as well as their values. Even a slight difference in the width of the arguments could produce a different result. Furthermore, it is impossible in general to produce exactly a given width in an interval result. The vagaries of round-off error make it impossible to predict the amount of accuracy that will be lost by a given operation, in general.

Each comparison will have an accuracy requirement on its argument[15]. We can compute the argument to smaller and smaller width, that is, higher and higher accuracy, but we know only that the width will decrease (the accuracy will increase) as the precision used for the computation grows. At some precision, the argument will meet or exceed the accuracy requirement, and the operation will be performed.

The program in Figure 4.6 shows an example of a program containing an `if` statement. Suppose the first approximation to `x` that satisfies the accuracy requirement[16] for instruction (1) overlaps the approximation to `C`; then the body of the `then` clause is executed, since by definition of comparison on intervals, overlap is considered to mean equality. Thus the value of `K` is added to `x`, and this is reflected in the digits output at instruction (2) (assuming `K` is not so small that it does not affect the first n decimal digits of `x`).

```
(1)   if x = C then
            x := x + K
            end
(2)   print x to n digits
      y := exp(x)
(3)   print y to m digits
```

Figure 4.6 An example program.

Now suppose that the value for `y` at the following execution of instruction (3) is not accurate enough to satisfy instruction (3)'s accuracy requirement. Then the value of `y` at (3), and thus the value of `x` used to compute `y`, must be recomputed using higher precision.

Now when `x` is recomputed, and we again reach instruction (1), suppose that the value of `x` at (1) no longer overlaps the interval for `C`, but is clearly less than `C`; if we simply reexecuted the comparison operation, we would now *not* add `K` to `x`. When we reach instruction (2), and `x` is output again, its value will not simply be a more

---

[15] Note that we can reduce any comparison to an operation of one argument by subtracting the comparands and comparing the difference to zero.

[16] The accuracy requirement is not shown in the program.

accurate approximation to the earlier output of instruction (2) — it will be less the value of `K`.

Figure 4.7 shows an example of the intervals for `x` and `c` resulting from a low precision computation and a high precision computation. The top portion of the figure shows `x` and `c` evaulated using a lower precision — they overlap. The bottom portion of the figure shows more accurate approximations to `x` and `c` — in this case they no longer overlap.

Even if we suppress the second output of `x`, the value output by instruction (3) will reflect the fact that `K` was not added to the value of `x` used to compute `y` — that is, the `exp` of the approximation to `x` output by instruction (2) will not equal or possibly even be close to the value approximation to `y` output by instruction (3). Neither can we save all output instructions until the program terminates, and then output the results of all the output instructions at once. First of all, the program may not terminate; second, operations that print out a value and require the programmer to give input based on that value must be performed before the program execution may proceed.

This type of behavior is *non-monotonic*; a program that behaves monotonically produces only prefixes of its final output; the program in Figure 4.6 produces output that produces two approximations for `x` that are not *consistent*. It should be obvious that the semantics of non-monotonic behavior are quite undesirable.

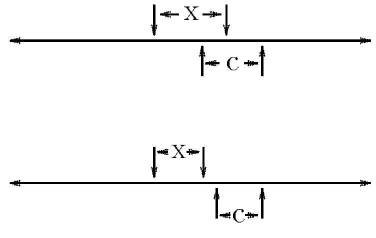

Figure 4.7 Intervals computed using different precisions.

In general, the results of decisions can change if the accuracy of the arguments increase. Note that we cannot require that the approximations be evaluated until they no longer overlap — if the values are actually equal, this process would never terminate. The intervals would get smaller and smaller, converging to the same value.

We make the following definition:

Definition 4.5 *The* first result of an output operation *o is the result produced by o using the first computed approximation of sufficient accuracy of the argument to o.*

When an output operation o is part of a program slice, that is, when it appears in one of the program sections, or it is a control flow instruction that one of the program sections depends on, we require that every reexecution of o produce o's first result. When an output instruction is encountered, no futher execution of the program can be done until an argument satisfying that instruction's accuracy requirement is computed; when this is done, the *first result* of the output instruction is computed and recorded. Then whenever the instruction is reexecuted, instead of recomputing the result, the first result is looked up and used instead.

Recall our discussion from Section 1.4.1 of the reexecution strategy used by Aberth. One of the strategy that was discussed was that it simply reexecutes all operations, including all output operations. As discussed above, the results of output operations can change as the programmer raises the precision used for the computation — thus as the the program is reexecuted repeatedly, the programmer could see markedly different results each time.

Note that first results need not be retained for some types of output operations — operations that have no effect on the program execution, specifically, operations that output approximations to real values, are in only one program slice; no instruction but the output instruction itself depends on it. These operations can be executed once and forgotten. Once the approximation is output, it is never needed again. We call those output instructions that *do* affect the program's execution, that is, those instructions whose results must be saved, *decision operations*.

When one uses only an interval mathematics package to execute a program containing real instructions, there is no control over the monotonicity of the program's behavior. Output operations will simply be reexecuted each time the precision is increased, and their results will be computed on the most recently computed value. Output operations that produce approximations for the user are reexecuted each time the program is reexecuted, and the results of decision operations may change as their arguments become more accurate.

First results are stored in program DAGs as part of the graph structure. For the program in Figure 4.6, if the first result determines that `x = C`, the dag shown at the top of Figure 4.8 would be produced. If `x` and `C` are clearly unequal, the dag at the bottom of Figure 4.8 would be produced. Note that DAGs automatically, implicitly save the first results that we save explicitly. Actually, DAG-building methods save the results of *all* comparisons implicitly as part of their DAGs.

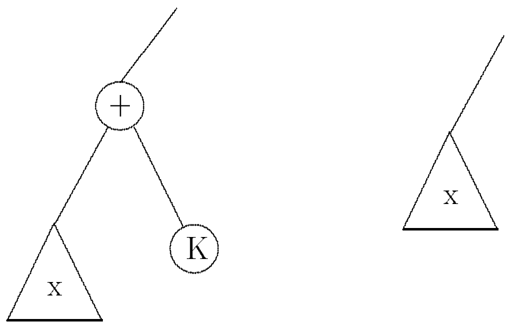

Figure 4.8 DAGs produced for different results of a comparison.

4.5 Open Values

Figure 4.9 shows a program fragment in which there are two output instructions, numbered (4) and (6). Instruction (4) depends on instructions (1) - (3), and instruction (6) depends on instructions (1) - (3) and (5).

Now suppose that when the program is executed, the first approximation computed for `x` at (1) is not accurate enough to print out `m` digits accurately. Then `x` is recomputed, perhaps several times, to produce a sufficiently accurate value.

Next suppose that, after the approximation to `x` is output, the approximation to `y` is not accurate enough to satisfy the requirement at (2). Then `y` must be recomputed, and thus we need a more accurate approximation for `x`. If we have saved the most accurate value for `x` that was computed for (1), we can simply use that value rather than recomputing an approximation to `x` unnecessarily.

In Section 3.3.2, we defined an *open value* to be a value defined in one slice chunk z that may be used in another section $y \neq z$. Since only (6) depends on instruction

(5), (5) will appear in a different section than instructions (1) - (3). Since instruction (5) uses the value of `x` computed at instruction (3), instruction (3) computes an *open value* (see Section 3.3.2).

```
      ...
(1)   x := C
(2)   for i := 1 to N do
(3)        x := log(x)
      end
(4)   output first m digits of x
(5)   y := sqrt(x)
(6)   output first m digits of y
```

Figure 4.9 Example Program.

We must store open values if we wish to avoid reevaluation costs being excessive due to repeatedly recalculating the same values. Note that DAG-building methods store open values at *each* node in the graph, since DAG-building methods maintain a most accurate approximation for each node (Section 1.3.2) to avoid unnecessary recalculation in that method.

4.6 Storing Computed Information

As discussed in Section 4.4, DAG-building methods store the first results of comparisons as part of the DAG structures they compute. Section 4.5 noted that DAG-building methods maintain open values at each node in the DAGs they produce. In our method, however, we must store both of these types of information explicitly.

The results of decision operations can be reduced to a single bit of information. Given a decision instruction d contained in loops $l_1, l_2, \ldots, l_n$, we must save a bit for the first result of d for each distinct loop value assignment $\vec{v}$ to $l_1, l_2, \ldots, l_n$. In this section we describe our method for storing these bits.

For each loop l in the program that contains one or more loops within its body, we number the loops inside l in their textual order; we call these numbers the *loop indices inside l*. These indices will be used to look up the bits associated with decisions that are contained within the body of l.

For each loop l_m with induction variable v_m, that is contained in loop sequence $l_1, l_2, \ldots, l_{m-1}$, where loop l_j has induction variable v_j for $1 \leq j < m$, we denote the initial value of v_m given the values of the v_i for $1 \leq i < m$ as

$$\mathrm{init}(v_1, v_2, \ldots, v_{m-1})$$

Likewise, we denote the final value of v_m as

$$\mathrm{final}(v_1, v_2, \ldots, v_{m-1})$$

We say that loop l_m is in the *body* of l_{m-1}. Each new execution of a particular textual loop l during an execution of the program is called an *instance* of loop l. Each time the *body* of loop l is begun is called an *iteration* of l. Note that there may be, and usually will be, several iterations per instance of a loop.

If l_m is a *forward* loop, that is, if its step size is positive, then the *low value* of l_m is simply its initial value. If l_m is a *backwards* loop, then low value of l_m is its final value. We define the *normalized value of* v_m to be v_m less its low value.

We store the bits in a structure indexed by loop indices and by normalized loop induction variable values. This structure consists of arrays alternately indexed by a loop index, then an induction variable value. At array entries that correspond to the location of decision statements, bits are stored in vectors.

The two basic forms that make up this structure are shown in Figures 4.10 and 4.11. Figure 4.10 shows the structure we call a *bodystruct*. We associate a bodystruct with each *instance* of each loop L that contains either any any loops, any decision instructions, or any definitions of open values in its body.

The first field of a bodystruct is a pointer to an array whose size may be increased dynamically. Each entry in this array is a pointer to a *loopstruct* (see below) corresponding to one iteration of one instance of L. Since in general the number of iterations in an instance of a loop cannot be calculated at compile time, the array's size can be increased as necessary. We increase its size by doubling each time more entries are required, thus reducing the cost of the size increases to a constant factor overall.

The second set of fields in a bodystruct is an array of pointers to *bit vectors*, one for each decision in the body of L. A bit vector is simply an array of true/false results for a decision instruction d. Each bit corresponds to a first result of an instance of d. Bit i in the vector corresponds to the first result of d when the instance of the loop L is in its i $^{\text{th}}$ iteration. As instances of d are evaluated, their results are recorded in its bit vector. Note that the array of pointers to bit vectors is of fixed size, and that that size is known at compile time.

The last set of fields in a bodystruct consists of an array of open value storage areas. Each open value that is computed in the body of an instance of L is stored in one of these entries. Note that the number of open values is also known at compile time.

Figure 4.11 shows the structure that is used for each iteration of each instance of loop L, called a *loopstruct*. A loopstruct consists of an array of pointers; each of these pointers corresponds to a different loop l in the body of L, and each points to a bodystruct for that l. There is a pointer in L's loopstruct for each iteration of the current instance of L.

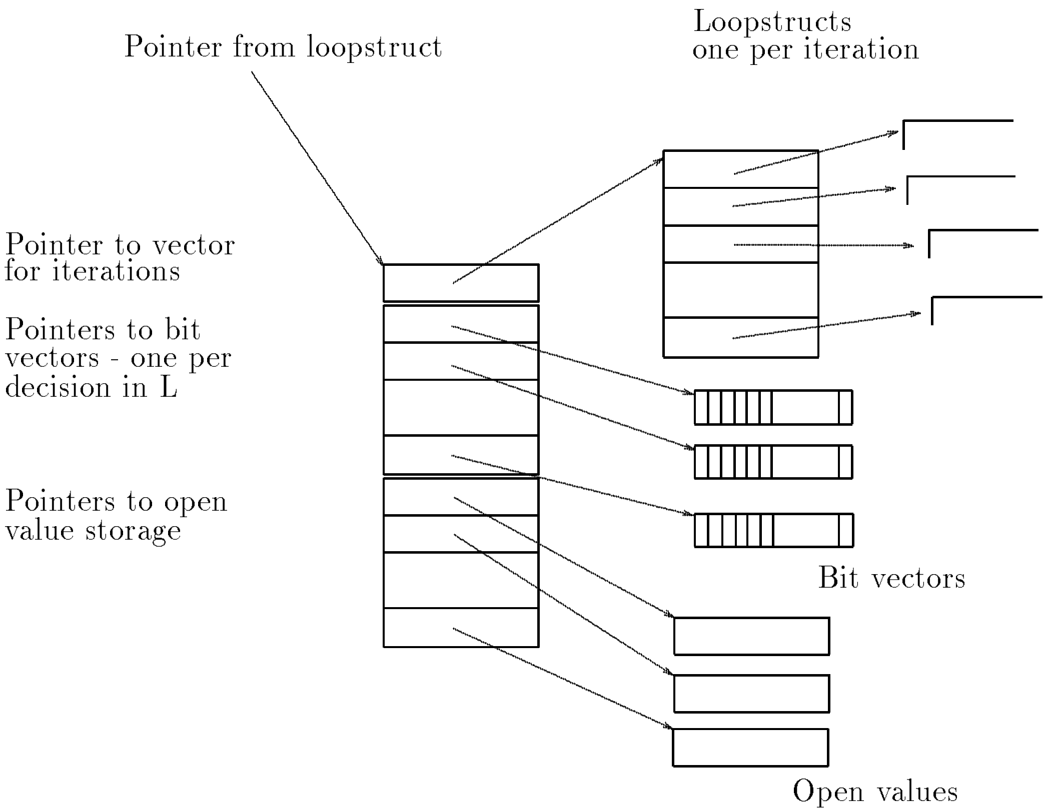

Figure 4.10 An Example Bodystruct.

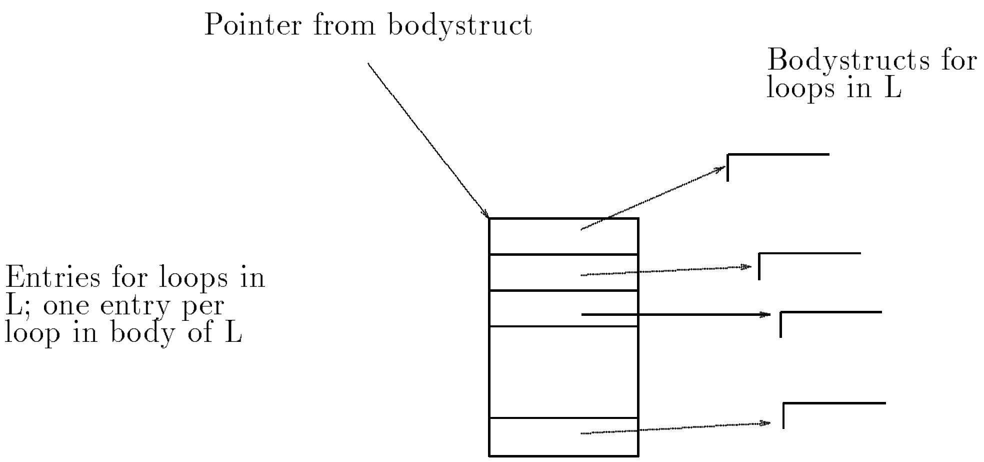

Figure 4.11 An Example Loopstruct.

An example program fragment is shown in Figure4.12, and a corresponding bit storage structure for that program in shown in Figure 4.13.

The structure for the `l` loop contains an entry for each iteration of that loop, and a bit vector for instruction (3), whose innermost containing loop is the `l` loop. Each iteration entry of the `l` loop contains a pointer to a structure for the `i` loop and a pointer to a structure for the `j` loop.

The structure created for an iteration of the `i` has only one field - a pointer to the bit vector for decision (1). Similarly, the structure for an iteration of the `j` loop contains only a pointer to the bit vector for instruction (2).

4.7 Optimizations

The subroutines shown in Figure 4.4 indicate the general procedure used to execute slice chunk instances. However, there are several shortcuts we can take that result in faster execution.

One optimization involves the representation of environments for slice chunks. Environments are not explicitly constructed and passed to slice chunks; thus, slice chunk do not have to reference variables in a special environment structure. Consider the procedure `BuildEnvironment` in Figure 4.4. Let V_c be the set of variables imported by a slice chunk c. Then, rather than building an environment containing

```
         for l := 1 to N do
             max := a[1, l]
             for i := 2 to N do
(1)              if a[i, l] > max then
                     max := a[i, l]
                 end
             end
             for j := 1 to N do
(2)              if a[j, l] ≠ max then
                     print "ratio is "
                     print first k digits of a[j, l] / max
                 end
             end
(3)          y := ⌊max⌋
         end
```

Figure 4.12 Example Program.

a variable-value pair for each variable in V_c, we simply save the current value of each member of V_c and set its value to the value returned by the appropriate call to `GetValue`. Then no special modifications have to be made to the code in the slice chunk c with respect to variable references — the existing referencing code executes simply. After c has finished executing, the saved values of its imported variables are restored from the stack, and execution continues.

Another important optimization involves slice chunks that have dependences on themselves. Suppose c is a slice chunk contained in a loop L, such that the instance of c in the i^{th} iteration of L requires a value computed by the $(i-1)^{\text{st}}$ iteration. Suppose further that c contains no loops itself, and that there are no other dependences whose sinks are within the L loop. Then consider what the algorithm of Figure 4.4 does when an instance of c needs to be executed.

When the i^{th} iteration of c is called, if i is not the first iteration of the loop containing c, then, since there is a dependence on c's previous iteration, the $(i-1)^{\text{st}}$ iteration must first be executed. Similarly, before the $(i-1)^{\text{st}}$ iteration may be executed, the $(i-2)^{\text{nd}}$ has to be executed, unless $(i-1)$ is the first iteration of the

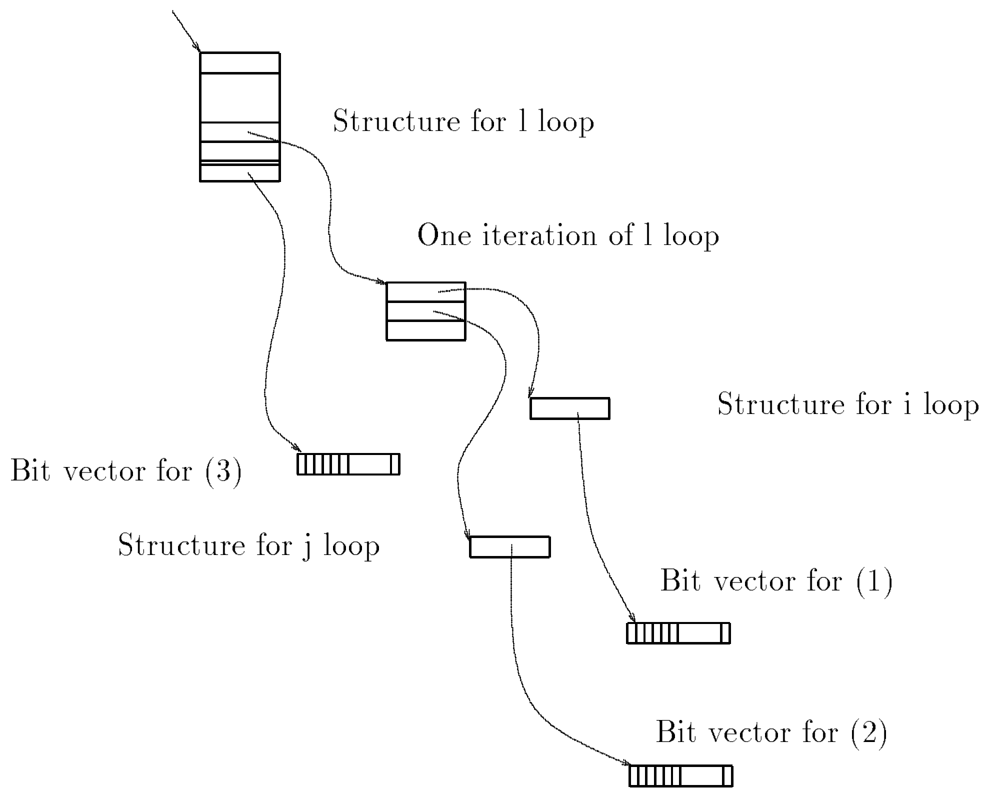

Figure 4.13 Example Bit Structure.

loop. When all iterations previous to the i^{th} have been executed, it can then be executed.

Instead of actually calling `ExecuteSliceChunk` once for each iteration of the loop L, we can instead just generate the code for the slice chunk c as an actual `DO` loop. This optimization removes the overhead of calling `ExecuteSliceChunk` and of saving and setting the values of the variables imported into c. This optimization, which we call "mashing loops," resulted in improvements of five to fifteen percent.

Chapter 5

Conclusions

This chapter discusses various details of our implementation and gives results for the example programs we have studied, and presents some conclusions and directions for future work.

5.1 Empirical Results

We perform our analysis on an intermediate code representation of the Fortran programs supplied by the user. The $\mathbf{R^n}$ project at Rice University uses an intermediate code called ILOC, designed for easy optimization and analysis. We perform our dependence analysis on a representation of the program in a variant of ILOC that we call ILOC$_R$. After analysis, we produce a new version of the program in ILOC, which is then given to the $\mathbf{R^n}$ code generator, which compiles ILOC to an executable form. All of our tests were performed on an IBM RT/PC.

All of our dependence analysis code as well as the programs that produce the new ILOC version are written in the C programming language. The "external dependence analysis tool" used by the research was computed in part by a simple program, and gathered in part by hand by the researchers. A file containing the array dependence information and the loop bound information is produced, and this file is read by the analysis phase, and used to produce the new version of the program. The information put in this file was carefully restricted to information that a standard dependence analysis tool could locate. The test programs are listed in the appendices.

The interval arithmetic package was written in Russell [8] [11]. Russell is an imperative-style language that supports first-class functional objects, polymorphic types, and first-class continuations. This package consists of subroutines that can be called from C that perform interval math operations such as addition, multiplication, square root, etc. It also contains a bignum arithmetic package, a garbage collector, and other utility functions that are part of the interval arithmetic code. These support functions are simply part of the Russell run time environment, and have been carefully tuned to the IBM RT/PC.

We now describe the test programs we analyzed, and then will present the experimental results.

GAUSS is a highly unstable eight-by-eight Gaussian elimination problem. This program contains some *unbindable* loops, due to the step in the algorithm that finds the maximum member of the current column, and swaps its associated row with the current row. The Fortran source of the GAUSS program is listed in the Appendix, in Section A.1. GAUSS 8 is an eight-by-eight matrix with a condition number of about 10^{31}; GAUSS 20 is the same problem with a twenty-by-twenty matrix, with condition number of about 10^{582}.

MATM is a twenty-by-twenty matrix multiply that prints out all the elements of the result matrix. The source for MATM is given in Section A.2 of the Appendix.

The RAY test is a program that performs Rayleigh iteration to find an eigenvector of a four-by-four matrix. Our test program performs six iterations. A matrix with a known eigenvector is used, and after each iteration, the computed eigenvector is compared to the known vector. Then for each iteration, the L2 norm of the difference of the computed eigenvector and the correct eigenvector is computed — this is the *error* in the current iteration. Then this error is divided by the *cube* of the error in the previous iteration, and an approximation of this ratio is displayed to the output. The RAY source is given in Section A.3 of the Appendix.

Since Rayleigh iteration converges *cubically*, the sizes of the elements of the difference of the computed and correct eigenvectors become smaller very rapidly. A very large amount of precision is required to compute the ratios described above — this is a very intensive use of high precision constructive real arithmetic.

Two types of measurements were performed to gauge the performance of our implementation. First, the execution time of each test program was measured using the UNIX *time* command. Each of the test programs was timed three times at each of a range of default precision values[17]. Table 5.1 shows the average time results for each test program. The *prec* column gives the default precision. The *prec-fn* column gives the average times for the interval-based DAG-building method, also called the precision-function method. The *precise* column contains the averate times for the

[17]Recall that interval-based methods use a *default precision* to determine how many significant digits are used in the first computation of each value.

data-flow analysis method described in this thesis. Finally, the *speedup* column gives the ratio of the precision function time to the data-flow analysis time.

Second, the maximum heap size of each test program was measured using a debugging feature in the Russell garbage collector. Table 5.2 displays the results of these measurements, taken for each of the test programs at a range of default precisions. Sizes are given in kilobytes, and are the maximum number of bytes on the heap at the time of any garbage collection. This is clearly only an approximate measurement of the memory usage we are interested in, that is, the size of the DAG structures in the case of the functional representations, and the size of the skeleton structure in the case of the sliced versions, but it is still informative. The table has the same structure as Table 5.1, except sizes are given rather than times.

In all examples, an accuracy requirement of 60 digits to the right of the decimal point was used.

It is important to note that precisions of 1 and 2 correspond to single precision and double precision floating point, respectively. Using our constructive reals implementation, the difference between the costs of running a program using default precision 1 and using default precision 4 is negligible, and thus it probably makes sense to use a default precision of at least 4 when computing over the reals.

Tables 5.1 and 5.2 show the results of the precision-functions and the slicing approach new to this research. Timing tests have also been performed for the accuracy-functions implementation, for the Gauss and Matm tests. The accuracy-function implementation takes on average 73.27 seconds and 92.2 kilobytes of heap space to execute the Gauss program, and 138.6 seconds and 1861.2 kilobytes of heap space for Matm[18]. On the Gauss 20 program it takes 9852.6 seconds and 1472.8 kilobytes of heap space. The Rayleigh test ran for several hours without terminating using the accuracy-function method.

5.2 Future Work

Many numerical problems involve the use of *iterative* algorithms. These algorithms involve making an initial guess, and then refining that guess by the repeated application of the iteration equations. Using simple floating-point arithmetic, iterative

[18]Note that the accuracy-functions have no "starting precision" concept, so only one measurement is done for each test.

test	prec	prec-fn	precise	speedup
Gauss 8	1	15.40	17.60	.8750
	2	15.10	14.43	1.0464
	4	12.70	11.37	1.1170
	8	10.77	9.33	1.1543
	16	7.00	5.53	1.2658
	32	12.27	12.03	1.0200
Gauss 20	1	237.23	258.70	0.9170
	2	234.20	223.03	1.0501
	4	210.60	186.53	1.1290
	8	190.90	169.00	1.1296
	16	157.40	137.30	1.1464
	32	245.93	191.57	1.2838
Matm	1	162.67	101.63	1.6006
	2	158.07	97.37	1.6234
	4	152.63	45.97	3.3202
	8	75.27	59.50	1.2650
	16	116.67	—	—
Ray	4	338.33	299.13	1.1310
	8	320.27	282.83	1.1324
	16	318.70	271.53	1.1737
	32	302.47	256.20	1.1806
	64	295.90	253.10	1.1691
	128	306.53	260.47	1.1769

Table 5.1 Results of Timings on Some Example Programs

test	prec	prec-fn	precise	ratio
Gauss 8	1	100.1	60.8	1.65
	2	101.1	61.2	1.65
	4	101.2	62.3	1.62
	8	101.5	70.1	1.45
	16	101.6	15.9	6.39
	32	149.0	21.2	7.03
Gauss 20	1	1,740.4	789.2	2.21
	2	1,735.1	826.8	2.10
	4	1,733.1	766.4	2.26
	8	1,733.1	849.9	2.04
	16	1,733.0	863.0	2.01
	32	1,744.0	90.8	19.21
Matm	1	1,752.5	716.0	2.45
	2	1,748.7	861.9	2.03
	4	1,730.4	67.9	25.48
	8	1,760.4	87.4	20.14
	16	2,299.7	125.9	18.27
Ray	4	771.3	413.6	1.86
	8	767.0	411.1	1.87
	16	770.8	408.3	1.89
	32	767.1	408.6	1.88
	64	770.8	414.6	1.86
	128	774.4	73.6	10.52

Table 5.2 Results of Space Measurements in Kilobytes

algorithms can produce quick, accurate results, although their accuracy is limited by the precision of the floating-point representation used.

Iterative algorithms can be incorporated into the implementation described here. An iterative algorithm effectively computes more and more accurate approximations to one or more real values. The programmer specifies a (potentially) infinite sequence of intervals for each value computed by the iterative algorithm. The intersection of all members in any such sequence must be a single point on the real number line — that is, the real number that the iterative algorithm is converging to. The arbitary-precision interval arithmetic package used to implement the reals in the slicing approach described here would make this sequence easy to specify.

When the execution of the program requires an approximation to a value computed iteratively, the sequence of intervals is "walked", that is, more and more elements in the sequence are evaluated, until an interval of sufficient accuracy is found. Probably the criterion used would be a function of the current precision — a requirement of a width no greater than twice the limit of the current precision might be reasonable.

Note that floating point implementations of these iterative algorithms effectively compute such a sequence of intervals. Rather than storing the error explicitly, as do intervals, however, floating point implementations have the error in the result stored implicitly in the termination test — for example, in a Rayleigh iteration the results of the n^{th} iteration are known to have three times as many significant digits as the $(n-1)^{\mathrm{st}}$ iteration. Thus supplying such a sequence would be a simple matter for a programmer.

5.3 Conclusions

We have presented a new approach to implementing constructive real arithmetic that uses data-flow analysis to prepare for possible recomputations of values. We have given the algorithms used to analzye the input program, and shown their validity. We have implemented this approach, and applied it to some representative test programs to obtain some empirical data on this method's performance. The performance improvements and the reduction in heap size are very positive.

Future work would include testing a larger range of programs, as well as testing the system on different architectures. Some more optimization could be done in the analysis and code generation phase. Further, the problem of subroutines needs to be addressed, so that complex programs can be studied.

Appendix A

Test Programs

In this appendix we give the Fortran listings of the test programs referenced in Chapter 5, and the contents of the array dependence files produced to simulate the results of an array dependence analysis tool. Within the program listings, superscripts are used to label array references and certain do loops. In the dependence listings, dependences are either of the form

$$N \longleftarrow M_0, M_1, \ldots, M_k$$

which indicates loop-independent dependences with sources M_i and sink N, or of the form

$$N \longleftarrow_L M$$

which indicates a loop-carried dependence with source M and sink N, and carrying loop L.

A.1 Gauss: Gaussian Elimination

A.1.1 Program Listing

```
program gauss
       double precision a(8, 9), x(8)
       double precision year, max, t, t1
       integer i, j, k, maxrw

       do i = 1, 8
            t = 1
            year = 1890.0 + ( i * 10.0 )
            do j = 1, 8
                 a(i, j)^1 = t
                 t = t * year
            enddo
```

```
enddo

a(1, 9)^5 = 75994575
a(2, 9)^6 = 91972266
a(3, 9)^7 = 105710620
a(4, 9)^8 = 123203000
a(5, 9)^9 = 131669275
a(6, 9)^10 = 150697361
a(7, 9)^11 = 179323175
a(8, 9)^12 = 203211926

do^2 k = 1, 7
     maxrw = k
     max = dabs( a(k, k)^13)
     do j = k + 1, 8
          t = dabs( a(j, k)^14)
          if (t .gt.  max) then
               max = t
               maxrw = j
               endif
          enddo
     do^4 j = k, 9
          t =  a(k, j)^15
          a(k, j)^4 =  a(maxrw, j)^16
          a(maxrw, j)^2 = t
          enddo
          do i = k + 1, 8
               t =  a(i, k)^17 /  a(k, k)^18
               do j = k + 1, 9
                    a(i, j)^3 =  a(i, j)^19 -  a(k, j)^20 * t
               enddo
          enddo
     enddo

     do^7 i = 8, 1, -1
          t = 0
          do j = i + 1, 8
```

```
            t = t +  a(i, j)21 *  x(i)25
            enddo
            x(i)24 = (  a(i, 9)22 - t ) /  a(i, i)23
        enddo

        do i = 1, 8
            call gprint( x(i)27)
        enddo

        do i = 1, 8
            year = 1890 + ( i * 10.0 )
            call gprint(year)
            t = 1
            t1 = 0
            do j = 1, 8
                t1 = t1 +  x(j)26 * t
                t = t * year
            enddo
            call gprint(t1)
        enddo
    end
```

A.1.2 Array Dependence File for Gauss

$$13 \longleftarrow 1$$
$$13 \longleftarrow_2 3$$

$$14 \longleftarrow 1$$
$$14 \longleftarrow_2 3$$

$$15 \longleftarrow 1, 5, 6, 7, 8, 9, 10, 11, 12$$
$$15 \longleftarrow_4 4$$
$$15 \longleftarrow_4 2$$
$$15 \longleftarrow_2 3$$

$$
\begin{array}{lll}
16 & \longleftarrow & 1,5,6,7,8,9,10,11,12 \\
16 & \longleftarrow_4 & 4 \\
16 & \longleftarrow_4 & 2 \\
16 & \longleftarrow_2 & 3 \\
\\
17 & \longleftarrow & 1,4,2 \\
17 & \longleftarrow_2 & 3 \\
\\
18 & \longleftarrow & 1,4,2 \\
18 & \longleftarrow_2 & 3 \\
\\
19 & \longleftarrow & 1,5,6,7,8,9,10,11,12,4,2,3 \\
19 & \longleftarrow_2 & 3 \\
\\
20 & \longleftarrow & 1,5,6,7,8,9,10,11,12,4,2,3 \\
20 & \longleftarrow_2 & 3 \\
\\
21 & \longleftarrow & 1,4,2,3 \\
22 & \longleftarrow & 1,4,2,3 \\
23 & \longleftarrow & 1,4,2,3 \\
25 & \longleftarrow_7 & 24 \\
26 & \longleftarrow & 24 \\
27 & \longleftarrow & 24
\end{array}
$$

A.2 Matm: Simple Matrix Multiply

A.2.1 Program Listing

```
program matm
       integer n, i, j
       double precision a(20, 20), b(20, 20), r(20, 20)
```

```
      n = 20

      do i = 1, n
           do j = 1, n
                a(i, j)^1 = i * 100 + j
           enddo
      enddo
      do i = 1, n
           do j = 1, n
                b(i, j)^2 = j * 100 + i
           enddo
      enddo

      do i = 1, n
           do j = 1, n
                r(i, j)^3 = 0.0
                do^6 k = 1, n
                     r(i, j)^4 =  r(i, j)^5 +  a(i, k)^6 *  b(k, j)^7
                enddo
           enddo
      enddo

      do i = 1, n
           do j = 1, n
                call^8 gprint( r(i, j) )
           enddo
      enddo
end
```

A.2.2 Array Dependence File for Matm

$$
\begin{array}{ccc}
6 & \longleftarrow & 1 \\
7 & \longleftarrow & 2 \\
5 & \longleftarrow & 3 \\
5 & \longleftarrow_6 & 4 \\
8 & \longleftarrow & 3,4
\end{array}
$$

A.3 Ray: Rayleigh Iteration

A.3.1 Program Listing

```
program ray
       double precision w(4), d(4, 4), a(4, 4), id(4, 4), temp(4, 4)
       double precision finx(4), t2(4, 4), x(4), t3(4)
       double precision ga(4, 5), y(4)
       double precision e, r, tws, max, t, le, ratio
       integer i, j, n, l, callk
       integer extcol, k, cont, maxrw

       n = 4
       extcol = 3     /* Column to use for finding eigenvector */
       cont = 6       /* Number of iterations */

       w(1)[1] = 1.5
       w(2)[2] = 2.3
       w(3)[3] = 0.8
       w(4)[4] = 9.8

       do i = 1, n
            do j = 1, n
                 d(i, j)[9] = 0.0
            enddo
       enddo
       d(1, 1)[10] = 800.5
       d(2, 2)[11] = 5.1
       d(3, 3)[12] = 0.00001
       d(4, 4)[13] = 50.01

       do i = 1, n
            do j = 1, n
                 id(i, j)[16] = 0.0
            enddo
            id(i, i)[17] = 1.0
       enddo
```

```
tws = 0.0
do i = 1, n
     tws = tws + (  w(i)^5 *  w(i)^6 )
enddo

do i = 1, n
     do j = 1, n
          temp(i, j)^20 =  id(i, j)^18 - ( 2.0 / tws ) * (  w(i)^7 *  w(j)^8 )
     enddo
enddo
do i = 1, n
     finx(i)^23 =  temp(i, extcol)^21
enddo

do i = 1, n
     do j = 1, n
          t2(i, j)^26 = 0.0
          do^10 l = 1, n
               t2(i, j)^28 =  t2(i, j)^27 +  temp(i, l)^14 *  d(l, j)^15
          enddo
     enddo
enddo

do i = 1, n
     do j = 1, n
          a(i, j)^30 = 0.0
          do^13 l = 1, n
               a(i, j)^32 =  a(i, j)^31 +  t2(i, l)^29 *  temp(l, j)^22
          enddo
     enddo
enddo

x(1)^35 = 0.0
x(2)^36 = 0.0
x(3)^37 = 0.0
x(4)^38 = 1.0
```

```
e = 0.0
do i = 1, n
     t =  x(i)[39] -  finx(i)[24]
     e = e + (t * t)
enddo

e = dsqrt(e)
do[15] k = 1, cont
     do i = 1, n
          t3(i)[48] = 0.0
          do[17] j = 1, n
               t3(i)[50] =  t3(i)[49] +  x(j)[40] *  a(j, i)[33]
          enddo
          enddo

          r = 0.0
          do i = 1, n
               r = r +  t3(i)[51] *  x(i)[41]
          enddo

          do i = 1, n
               do j = 1, n
                    ga(i, j)[52] =  a(i, j)[34] - (r *  id(i, j)[19])
               enddo
               ga(i, n + 1)[53] =  x(i)[42]
          enddo

          do[21] l = 1, n - 1
               maxrw = l
               max = dabs( ga(l, l)[54])
               do j = l + 1, n
                    t = dabs( ga(j, l)[55])
                    if (t .gt.  max) then
                         max = t
                         maxrw = j
                    endif
               enddo
```

```
        do^23 j = l, n + 1
            t =  ga(l, j)^56
            ga(l, j)^58 =  ga(maxrw, j)^57
            ga(maxrw, j)^59 = t
        enddo

        do i = l + 1, n
            t =  ga(i, l)^60 /  ga(l, l)^61
            do j = l + 1, n + 1
                ga(i, j)^64 =  ga(i, j)^62 -  ga(l, j)^63 * t
            enddo
        enddo
    enddo

    do^26 i = n, 1, -1
        t = 0.0 do j = i + 1, n
            t = t +  ga(i, j)^65 *  y(j)^68
        enddo
        y(i)^69 = ( ga(i, n + 1)^66 - t) /  ga(i, i)^67
    enddo

    t = 0.0
    do i = 1, n
        t = t + ( y(i)^70 *  y(i)^71)
    enddo
    t = dsqrt(t)
    do i = 1, n
        x(i)^43 =  y(i)^72 / t
    enddo

    le = e * e * e
    e = 0.0
    do i = 1, n
        t =  x(i)^45 -  finx(i)^25
        e = e + t * t
    enddo
```

```
            e = dsqrt(e)
            ratio = e / le
            callk = k

            call gprint(r)
            call gprint(e)
            call gprint(ratio)

            do i = 1, n
                call gprint( y(i)[73])
            enddo
            do i = 1, n
                call gprint( x(i)[44])
            enddo
        enddo
    end
```

A.3.2 Array Dependence File for Ray

5	$\longleftarrow$	$1, 2, 3, 4$
6	$\longleftarrow$	$1, 2, 3, 4$
7	$\longleftarrow$	$1, 2, 3, 4$
8	$\longleftarrow$	$1, 2, 3, 4$
15	$\longleftarrow$	$9, 10, 11, 12, 13$
18	$\longleftarrow$	$16, 17$
19	$\longleftarrow$	$16, 17$
14	$\longleftarrow$	2
21	$\longleftarrow$	20
22	$\longleftarrow$	20
24	$\longleftarrow$	23

$$
\begin{array}{lcl}
25 & \longleftarrow & 23 \\
\\
27 & \longleftarrow & 26 \\
27 & \longleftarrow_{10} & 28 \\
29 & \longleftarrow & 28 \\
\\
31 & \longleftarrow & 30 \\
31 & \longleftarrow_{13} & 32 \\
33 & \longleftarrow & 32 \\
34 & \longleftarrow & 32 \\
\\
39 & \longleftarrow & 35, 36, 37, 38 \\
39 & \longleftarrow_{15} & 43 \\
40 & \longleftarrow & 35, 36, 37, 38 \\
40 & \longleftarrow_{15} & 43 \\
41 & \longleftarrow & 35, 36, 37, 38 \\
41 & \longleftarrow_{15} & 43 \\
42 & \longleftarrow & 35, 36, 37, 38 \\
42 & \longleftarrow_{15} & 43 \\
44 & \longleftarrow & 43 \\
45 & \longleftarrow & 35, 36, 37, 38, 43 \\
\\
49 & \longleftarrow & 48 \\
49 & \longleftarrow_{17} & 50 \\
51 & \longleftarrow & 50 \\
\\
54 & \longleftarrow & 52 \\
54 & \longleftarrow_{21} & 64 \\
55 & \longleftarrow & 52 \\
55 & \longleftarrow_{21} & 64
\end{array}
$$

$$
\begin{aligned}
56 &\longleftarrow 52, 53 \\
56 &\longleftarrow_{23} 58 \\
56 &\longleftarrow_{23} 59 \\
56 &\longleftarrow_{21} 64 \\
57 &\longleftarrow 52, 53 \\
57 &\longleftarrow_{23} 58 \\
57 &\longleftarrow_{23} 59 \\
57 &\longleftarrow_{21} 64 \\
60 &\longleftarrow 52, 58, 59 \\
60 &\longleftarrow_{21} 64 \\
61 &\longleftarrow 52, 58, 59 \\
61 &\longleftarrow_{21} 64 \\
62 &\longleftarrow 52, 53, 58, 59, 64 \\
62 &\longleftarrow_{21} 64 \\
63 &\longleftarrow 52, 53, 58, 59, 64 \\
63 &\longleftarrow_{21} 64 \\
65 &\longleftarrow 52, 58, 59, 64 \\
66 &\longleftarrow 52, 58, 59, 64 \\
67 &\longleftarrow 52, 58, 59, 64
\end{aligned}
$$

$$
\begin{aligned}
68 &\longleftarrow_{26} 69 \\
70 &\longleftarrow 69 \\
71 &\longleftarrow 69 \\
72 &\longleftarrow 69 \\
73 &\longleftarrow 69
\end{aligned}
$$

References

[1] Oliver Aberth. A precise numerical analysis program. *Communications of the ACM*, 17(9):509–513, September 1974.

[2] Oliver Aberth. Precise scientific computation with a microprocessor. *IEEE Transactions on Computers*, C-33(8):685–690, August 1984.

[3] Oliver Aberth. *Precise Numerical Analysis.* Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[4] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *SIGPLAN 90*, pages 246–256, June 1990.

[5] Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, New York, 1967.

[6] Hans-J. Boehm. Constructive real interpretation of numerical programs. *SIGPLAN Notices*, 22(7):214–221, July 1987.

[7] Hans-J. Boehm, Robert Cartwright, Mark Riggle, and Michael J. O'Donnell. Exact real arithmetic: A case study in higher order programming. In *Proceedings of the 1986 Lisp and Functional Programming Conference*, pages 162–173, 1986.

[8] Hans-J. Boehm, Alan Demers, and James Donahue. A programmer's introduction to russell. Technical Report 85-16, Rice University, Houston, Texas, 1985.

[9] Preston Briggs, Keith D. Cooper, and Linda Torczon. ILOC '87. $\mathbf{R}^{\mathbf{n}}$ Programming Environment Newsletter #44, Rice University, Houston, Texas, September 1987.

[10] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *SIGPLAN 89*, pages 13–27, June 1989.

[11] J. Donahue and A. Demers. Data types as values. *ACM Transactions on Programming Languages and Systems*, 3(7):426–445, July 1985.

[12] J. Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, pages 319–349, July 1987.

[13] Bill Gosper. Continued fraction arithmetic. HAKMEM Item 101B, AI Memo 239, MIT, February 1972.

[14] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[15] Vernon A. Lee Jr. and Hans-J. Boehm. Optimizing programs over the constructive reals. In *SIGPLAN 90*, June 1990.

[16] K. Kennedy and J. R. Allen. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, pages 491–542, October 1987.

[17] Ken Kennedy. Use-definition chains with applications. In *Computer Languages*, volume 3, pages 163–179. Pergamon Press, 1978.

[18] Ken Kennedy. A survey of data flow analysis techniques. In S. Muchnik and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, Inc., 1981.

[19] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, Massachusetts, second edition, 1969, 1981.

[20] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[21] Fritz Krückeberg. Arbitrary accuracy with variable precision arithmetic. In *Interval Mathematics 1985: Proceedings of the International Symposium*, pages 95–101, September 1985.

[22] D.B. Loveman. Program improvement by source to source transformation. *Journal of the ACM*, 24(?):121–145, January 1977.

[23] R.E. Moore. Methods and applications of interval analysis. In William F. Ames, editor, *SIAM Studies in Applied Mathematics*. SIAM, 1979.

[24] J. Myhill. What is a real number? *American Mathematical Monthly*, 79(7):748–754, August-September 1972.

[25] Michael J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.

[26] H. G. Rice. Classes of recursively enumerable sets of positive integers and their decision problems. *Trans. Amer. Math. Soc.*, 74(2):358–366, 1953.

[27] P.L. Richman. Automatic error analysis for determining precision. *Communications of the ACM*, 15(9):813–817, September 1972.

[28] Jerry Schwartz. Implementing infinite precision arithmetic. In *9th Symposium on Computer Arithmetic*, Santa Monica, CA, 1989.

[29] Donald F. Stanat and David F. McAllister. *Discrete Mathematics in Computer Science*. Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[30] A. Turing. On computable numbers, with an application to the *entscheidungsproblem* . *Proc. London Math. Soc.*, pages 230–265, 1936.

[31] Jean Vuillemin. Exact real computer arithmetic with continued fractions. Research Report 760, INRIA, 1987.

[32] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[33] E. Wiedmer. Computing with infinite objects. In *Theoretical Computer Science*, volume 10, pages 133–155. North-Holland Publishing Company, 1980.

[34] J.M. Yohe. Software for interval arithmetic: A reasonably portable package. *ACM Transactions on Mathematical Software*, 5(1):50–63, March 1979.