A New Language Design for Prototyping Numerical Computation

THOMAS DERBY, ROBERT SCHNABEL, AND BENJAMIN ZORN

Department of Computer Science, Campus Box 430, University of Colorado, Boulder, CO 80309

ABSTRACT

To naturally and conveniently express numerical algorithms, considerable expressive power is needed in the languages in which they are implemented. The language Matlab is widely used by numerical analysts for this reason. Expressiveness or ease-of-use can also result in a loss of efficiency, as is the case with Matlab. In particular, because numerical analysts are highly interested in the performance of their algorithms, prototypes are still often implemented in languages such as Fortran. In this article we describe a language design that is intended to both provide expressiveness for numerical computation, and at the same time provide performance guarantees. In our language, EQ, we attempt to include both syntactic and semantic features that correspond closely to the programmer's model of the problem, including unordered equations, large-granularity state transitions, and matrix notation. The resulting language does not fit into standard language categories such as functional or imperative but has features of both paradigms. We also introduce the notion of language dependability, which is the idea that a language should guarantee that certain program transformations are performed by all implementations. We first describe the interesting features of EQ, and then present three examples of algorithms written using it. We also provide encouraging performance results from an initial implementation of our language. © 1996 John Wiley & Sons, Inc.

1 INTRODUCTION

As problems in any domain become more complex, they demand increasingly abstract notations with which to describe solutions. For example, in the domain of numerical computation, Fortran in its many forms has always been a favorite implementation language. Recently, however, many numerical analysts are using more expressive languages and language environments such as Matlab or Mathematica to initially express, evaluate, and debug their algorithms. While Matlab and Mathematica provide advantages over traditional languages for numerical computation, in their current form, they lack optimizing compilers.* As a result, many numerical analysts continue to implement their algorithms directly in Fortran.

The purpose of programming language design is to provide humans with simple and natural abstractions that simplify the translation from the human mind into a computable form of expression. While some would argue that Fortran provides such a set of abstractions for numerical ana-

Received March 1995
Revised January 1996

* Only recently has work on compiling Matlab been published [1].

lysts, our experiences and the widespread use of Matlab suggests otherwise. In particular, in interviewing local numerical analysts working in the area of numerical optimization and reviewing the published literature in the field, we observed that these scientists expressed their algorithms in pseudo-code at a level typically much higher than that of Fortran notation. Motivated in part by these observations, we have designed a programming language intended for use in prototyping linear algebra and numerical optimization computations. We have focused our work to date in these areas (avoiding the broad category of differential equations) in order to limit the scope of the research. Extension of EQ into the domain of differential equations is an obvious direction for future work.

Our language, EQ, provides several important features that directly reflect the syntactic and semantic ideas we observed being used. These features are:

1. Unordered expressions: The numerical analysts we interviewed described parts of computations out-of-order. Their "natural" semantics of the "=" operator was that it defined a relationship, and not that it updated a memory location. Specifically, they used a different notation to describe a variable definition (e.g., *tolerance* = 0.0001) and a recurrence relation (e.g., $x' = x - lambda \times H \times gx$).
2. Large-granularity state transitions: The numerical analysts used a notation that indicated an update to the state of the computation, as shown by the recurrence relation notation mentioned above. Thus, the analysts' model of computation was based on larger-grain state transitions than on the granularity of individual memory location updates.
3. Matrix notation: Uniformly, the analysts used standard matrix notations to describe computations on arrays.

Incorporating these ideas has led us to a language design that has features of both imperative languages, such as Fortran and C, and functional languages, such as Sisal.

Another aspect of the EQ design is related to the issue of performance. Traditionally, performance has been a critical aspect of numerical computation. One of the strongest arguments for using Fortran is that the performance of its programs is traditionally excellent and well understood. In this article, we introduce the principle of language dependability, by which we mean the guarantees the language provides in relation to issues of optimization. More specifically, dependability refers to the requirement that certain program transformations are performed by all implementations of the language. As an example of language dependability, tail-recursion elimination is a required transformation in all Scheme implementations [2]. We recognize that providing dependability in general is a difficult problem. As a result, in this article, we discuss what language transformations we would like to dependably provide in EQ, and why the EQ language design is amenable to such guarantees. What transformations can be provided dependably in EQ remains an open issue and will be considered further in future work.

In this article, we introduce the important concepts of EQ and illustrate its use with three examples representing small numerical computation kernels. We also show that our preliminary EQ implementation performs well for these programs. Unlike other languages in this domain, the expression of parallel computation with EQ, either explicitly or implicitly, is not a primary goal of the design at this time. Some of the features of EQ appear well suited to parallelization, but this aspect is not the thrust of our current research.

The remainder of this article is organized as follows. Our language is related to many other languages, and in Section 2 we describe some of those relationships in detail. In particular, we discuss in detail the differences between EQ and its closest relatives, Sisal, Id, and EPL. In Section 3, we describe the important features of EQ, including their syntax, semantics, and our motivation for including them. Three example EQ programs, with line-by-line commentary, are provided in Section 4. We provide some preliminary implementation results in Section 5, including timing measurements for our three example EQ programs. Section 6 discusses the issue of dependability in more detail, including our goals for dependability in EQ. Finally, we describe future research in Section 7, and summarize our conclusions in Section 8.

2 RELATED WORK

In this section, we discuss how our work with EQ relates to existing programming language designs. Because EQ contains side-effect-free features (e.g., unordered equations), large-grain state transitions, and array notations, there are many different languages that are related. This section is by no means a complete listing of relevant languages; instead, it tries to present at least one example of each language category we discuss. At the end of this section, we summarize the aspects of our design that make it unique.

2.1 Imperative Languages

Imperative languages are those that describe computations by giving an ordered sequence of steps, each of which changes the state of the computation. In these languages, the granularity of state transition is usually very small (i.e., a single memory location). This class of languages includes Fortran [3], C [4], Matlab [5], APL [6], Fortran 90 [7], as well as object-oriented languages such as C++ [8]. Scientific computation uses this class of languages almost exclusively.

Because they allow the modification of locations through the use of the assignment statement, imperative languages are generally good at expressing algorithms involving change. As we have noted, the concept of change (or state transition) is expressed explicitly in many numerical algorithms, and as a result, such algorithms are relatively easy to express in imperative languages. In contrast, many functional languages, while allowing state transitions to be modeled, do not allow such transitions to be expressed concisely and/or provide features to facilitate the use of such transitions (for exceptions, see below). This property alone may explain much of the imperative paradigm's popularity over the functional paradigm for scientific programming.

Imperative languages also typically use an explicit memory allocation scheme, through variable declarations, which allows them to easily be memory efficient, something with which most other language paradigms (including EQ) have more difficulty. This approach also has disadvantages, however. Forcing users to explicitly deal with memory allocation prevents them from conveniently creating temporary objects, or from using objects that have more logical dimensions than the storage needed to hold them.†

We discuss two broad classes of imperative programming languages below, giving examples of specific languages that have been suggested for use in scientific programming.

† An example of an array with extra dimensions is a matrix that is computed one column at a time, with all columns reusing the same storage.

Scalar Imperative Languages

Scalar imperative languages only allow the programmer to manipulate and update simple numerical quantities, rather than aggregate values such as arrays or records. Fortran is the foremost example of this class, and is also the most popular language for use in scientific computation. This fact has been true from its introduction right through to the present day, and its popularity seems likely to continue in some form (Fortran 90 is discussed in the next section). Despite criticisms by some language researchers [9], Fortran has many features that are desirable in scientific computing. In addition to its ability to represent change easily, Fortran also offers the ability to handle complex numbers and variably sized subprogram arguments, features that are not available in other common imperative languages such as C or Pascal. Excellent compilers, a simple execution model, and general familiarity have further added to Fortran's success.

Because scalar imperative languages by definition do not support expressions using larger aggregates such as arrays, they are not particularly concise; small numerical algorithms can explode into large amounts of code when written using languages such as Fortran. In conjunction, the resulting programs often do not correspond well with the user's understanding of the algorithms used by that program. This problem is often exacerbated by the requirement that the user perform explicit storage management. In an effort to minimize the storage allocation of their algorithms by hand, Fortran programmers often greatly complicate their code (e.g., by packing two triangular matrices into storage for a single square array).

Imperative Languages with Array Expressions

A number of imperative languages, including APL, Matlab, and Fortran 90, allow expressions to operate on entire arrays or slices of arrays. We discuss the features provided by these languages, and how these designs have influenced EQ.

The widespread use of APL illustrates the power of providing higher-level array operations in a programming language. While APL has been used in numerical applications, it also provides other very general array operations such as rotation. APL also encouraged the use of array computations to an extreme, popularizing the one-liner approach to problem solving. One reason that APL was so successful is that it provides a very natural and concise notation for describing many problems. As a result, APL has influenced many subsequent languages, including Backus' FP [10].

Matlab supports the use of mathematically sound operations, such as matrix multiplication. In Matlab, users are encouraged to write as much of their computation as possible in the array/matrix notation for several reasons. First, the array notations are more concise than the less concise iterative facilities. Second, the interpretive nature of existing Matlab implementations means that a significant performance penalty is taken when computations are written using control structures rather than the matrix notation. This problem occurs because while the underlying matrix computation routines are implemented in compiled libraries linked into the Matlab executable, any control structures are executed though intepretation, greatly reducing the performance. This efficiency problem can prevent Matlab from being used to perform algorithm comparisons using large data sets.

Fortran 90 extends Fortran 77 in a number of ways, including adding new facilities for manipulating pointers and records, a new module system, and some limited facilities for computation with complete arrays. The inclusion of array facilities in Fortran 90 supports our belief that numerical programmers benefit from such extensions. On the other hand, because Fortran 90 extends and must remain compatible with existing Fortran dialects, support for array expressions in Fortran 90 is not complete. In particular, in EQ we support the notion of a range variable which embodies the concept of an array index, and which is not supported in Fortran 90. Furthermore, the reference manual itself suggests a significant loss of efficiency when using Fortran 90's array features [7].

2.2 Functional Languages

Declarative programming languages are characterized by the absence of side effects and sequencing, essential characteristics of imperative languages. Functional languages, specifically, also typically provide support for higher-order, first-class functions. By the definition of the paradigm, these languages lack direct support for updates to locations. As a result, computations with state must be simulated by explicitly passing the state as an argument and updating it via copying (at least conceptually, if not in practice). For example, instead of implementing iteration via updates to a loop index, many of these languages require iteration to be specified using a recursive procedure.

Some functional languages are actually mostly functional in that they include a standard imperative assignment statement as an option (e.g., Scheme and Standard ML). While these semantics result in programming flexibility, they also prevent some of the significant advantages of the functional paradigm (e.g., referential transparency and lazy evaluation). Other functional languages introduce state in a semantically limited way. In particular, Backus' applicative state-transition (AST) systems [10] are similar to the state-transition semantics proposed for EQ in that they both propose larger-grain state transitions than single memory cells. Id (discussed below) also provides imperative features with restricted semantics [11].

A small number of functional programming languages (e.g., Sisal), have been designed specifically to effectively support scientific computation. These languages include syntactic and semantic constructs beyond those of purely functional languages such as Haskell [12] and Miranda [13], and are closest in spirit to EQ. In this section we will discuss similarities and differences between EQ and these languages.

Functional Languages with Explicit Looping Constructs

Since scientific programmers often seek to express algorithms that involve iteration, languages such as Sisal [14] and Id [15] add explicit looping constructs to support this goal. The general looping constructs in these languages are similar to the ones found in the imperative paradigm (while and for loops). Their bodies specify the new value of the variables in terms of the old one, with some syntax that looks very imperative, such as:

```
x = x + 1;
```

While Sisal and Id both support an imperative-style notation for iteration in similar ways, they support the notation of array modification in very different ways. In particular, Sisal only allows the user to change a whole object: it is not possible to change a single element of an array without conceptually constructing a whole new array, at least at the language level. While Sisal has been fairly successful at using optimization techniques to eliminate the run-time inefficiencies associated with updating whole arrays rather than their pieces, the language constructs do not correspond to the large-grain state-transition semantics that we observed programmers apply in solving scientific problems. In particular, we believe that programmers sometimes think in terms of "partial update" semantics. For example, to describe an identity matrix, a user can think of updating a zero matrix to place ones only on the diagonal. EQ differs from Sisal in many other ways as well. EQ's array manipulation mechanisms (the range variables discussed in Section 3.4) are based on implicit looping operators, rather than on explicit for loops, array operators, or vectors subscripted with index arrays. The distinctions between index arrays and range variables are further described in Section 3.4. Also, EQ's iterative looping notations are more general than Sisal's, permitting multiple exit conditions from a loop, an important concept in many numerical algorithms. Overall, the resulting programming style in EQ is quite different from that used in Sisal.

Unlike Sisal, the programming language Id supports partial updates to array objects. Id provides an imperative mechanism called an M-structure that allows values to be entered and removed from individual elements of an array [11]. M-structure semantics require that before a new value can be placed in an M-structure, the old value must first be removed. M-structures were added to Id because it was observed that threading explicit state variables sequentializes otherwise parallel scientific computation and can result in significant amounts of copying.

M-structures provide an interesting alternative approach to supporting imperative semantics in a mostly functional framework. They differ from the semantics of state transitions in EQ because state transitions remain fine-grained and updates are asynchronous. An important purpose for M-structures is to provide implicit synchronization in Id. In EQ, on the other hand, the purpose of our large-grain state-transition mechanism (i.e., see the "followed-by" operator in Section 3.7) is to provide a feature that naturally models the scientific programmer's intent. In particular, followed-by supports the notion of a synchronous partial update across all variables currently in scope. While these semantics limit parallelism as compared to the semantics of M-structures, the current intent of the EQ design is not to support a maximal amount of parallelism, but to provide more natural sequential language semantics.

Functional Languages with Implicit Looping Constructs

Rather than use standard control structures, some functional languages express loops implicitly. In Lucid [16], this is done by treating variables as representing a whole stream of values, and writing equations such as:

```
x = 1 followed-by x * 2;
```

which sets x to the sequence (1, 2, 4, 8, 16, ...).

EPL [17] uses a different mechanism; to describe a recurrence, one writes an array and defines each value in the array in terms of the previous values. So, for example, the above computation could be written:

```
subscript i;
x[1] = 1;
x[i] = x[i-1] * 2;
```

Both of these languages also have difficulty expressing the change of only part of an object. In addition, these constructions are more inefficient than those of Sisal and Id, because it may be very difficult to easily determine which values to compute first; this must be decided either at run-time (resulting in drastic loss of efficiency), or by using sophisticated compile-time analysis. This can be unreliable in the sense that compile-time analysis is not guaranteed to generate a solution for sufficiently complex sequence definitions. In such cases, the language must fall back on the run-time system. Furthermore, the program may contain cyclic definitions only detectable at run-time. Thus, it is difficult for the user to depend on the compiler of such languages to generate reasonable, or even consistent, performance. This relates to the concept of dependability, which is introduced in Section 6.

2.3 Object-Oriented Languages

Object-oriented programming has become a widely used important programming paradigm that is supported in many different languages. For the most part, existing applications of object-oriented approaches to scientific computing problems have been in the areas of class library definition (e.g., matrix and array classes, libraries to support specific scientific fields such as magnetic resonance [18], etc.). While object-oriented ideas relate to high-level structuring mechanisms (e.g., class hierarchies), our ideas are more relevant to the practice of programming-in-the-small. Finally, it is quite clear that object-oriented concepts such as inheritance, polymorphism, and dynamic dispatch are not yet in common use among scientists. The object-oriented paradigm may eventually greatly influence the way scientific programming is done, but the effective use of this paradigm for scientific computation is still a topic of research and beyond the scope of this article.

2.4 Summary of Relation of Existing Languages to EQ

While EQ shares syntactic and semantic features with many different languages (from different paradigms), it combines them in a way that makes it unique. First, it supports syntactic constructs intended to closely model those used by numerical analysts describing computations involving linear algebra. While a language such as Matlab also supports this goal, it does so in a purely imperative framework. Second, the semantics of EQ are mostly functional, with the addition of large-grain state transitions intended to closely support the conceptual model of the programmer. While other functional languages, such as Sisal and Id, also support numerical computation and side effects, they do so in different ways, and with different goals.

3 FEATURES OF EQ

This section describes the EQ language. Because this article is intended as a language overview and not a language reference, we focus here on the most important aspects of the language. A more complete language reference manual that provides further details of the language design is currently in preparation.

After a brief discussion of lexical issues, we describe the EQ language in two basic parts. First, EQ provides a set of definitional features for defining new values in terms of old ones. This part of the language is very similar to ideas from a variety of functional languages, except that it allows for partial definitions—the definition of only part of an array, or definitions that occur under only some circumstances. Portions of EQ that fall into the definitional category include unordered equations, statement grouping, the conditional statement, range variables, and matrix expressions.

Next, we describe EQ's unique support for iterative constructs, which involve a notation for future and past values, and then we describe explicit operations that cause time to "flow," moving values from the future into the past. The features in EQ that support change include the "next" and "prev" operators, the followed-by operator, and the do loop.

3.1 Lexical and Syntactic Issues

To date, our research has focused on semantic, rather than syntactic, issues. As a result, the formation of tokens, interpretation of numbers, and other similar issues are treated as they are in the C programming language. These issues do not affect the semantics of the language constructs presented below.

3.2 Unordered Equations

Mathematical notations, in general, distinguish between ordered and unordered equations. Unordered definitions are given as a set of equations. Some notations have extra words between these equations; "x = sin y, where y = . . ." and "let y = . . . in x = sin y" are common. Ordered definitions, involving a set of recurrences, are given using a special syntax (e.g., $x' = x + 1$). EQ semantics support both of these notions directly. Specifically, in this section we discuss expressing unordered notations, and in a later section we discuss recurrences. In EQ a use of "=" corresponds to the simple unordered model. For example:

```
a = 4;
b = a + correction;
correction = sqrt (a);
```

would compute `a` = 4, `correction` = 2, and b = 6. This unordered form of expression allows the programmer greater flexibility. For programming-in-the-small, unordered statements can lead to a more natural way of expressing algorithms; statements that logically belong together can be placed in proximity, even if they cannot be executed consecutively.

In addition, unordered equations correspond to a proof-oriented view of the program; each equation can be thought of as a fact about the values the program computes, rather than a computational rule. These facts can then be used to prove properties about programs. From this point of view, it is convenient to be able to view these statements as unordered.

As we have described so far, these features are fairly similar to the unordered facilities provided by many functional languages. However, in addition to allowing the user to assign an entire variable, EQ allows an assignment to only part of an array. The statement

```
a[3,] = r~;
```

would assign r to the third row of a (arrays are accessed in column-major order), assuming r is a one-dimensional column vector (the "~" operator transposes r into a row; see Section 3.5). In Section 3.7, we describe what happens to the undefined parts of an array such as a. Note that attempting to use a totally undefined variable (one which has no definitions at all) is a compile-time error.

EQ is a single-assignment language; this means that the same variable cannot be defined twice. EQ insists on this property even for arrays; the code fragment

```
a[1] = 1;
a[2] = 2;
```

is not permitted in EQ. By permitting only one assignment to an array variable, EQ avoids the problem of determining at run-time if any element of the array is defined twice (a double definition would mean that the order of execution of these unordered statements could affect the computed result). It is, however, sometimes desirable to be able to write code fragments such as the above; we show how to achieve the effect of multiple assignments in Section 3.7.

3.3 Conditionals

Conditional computations in EQ are handled in a very similar way to that used in imperative languages:

```
if (x >= y) {
  maximum = x;
  minimum = y;
}
else {
  maximum = y;
  minimum = x;
}
```

computes both the minimum and maximum of x and y. We note that braces are used, as in C, to group several individual statements into a single compound statement. Although there are multiple assignments to variables here, each branch of the conditional must assign to a variable at most once. This rule ensures that there is never more than one definition active, since only one branch of the conditional is executed.

By providing a conditional statement (as opposed to the conditional expression used by functional languages), we have avoided the need to have records or tuples.‡ Adding such types to EQ would not be difficult, but would be a violation of our direct representation design principle. More importantly, the conditional statement permits the programmer to assign to a variable in only one of the branches of an if statement; this will have a well-defined (and important) meaning when used with EQ's iterative statements (see Section 3.8).

3.4 Range Variables

Performing computations over a set of values is a very common activity in programming. In numerical computations, typically these are sets of subscripts. Using loop notation (as is required in languages such as Fortran) obscures the intent and potential parallelism of the program, in addition to expanding code volume. To address these issues, EQ provides the range variable, which takes on a consecutive set of integer values. The behavior of these range variables is strongly related to that of EPL's subscript variables, but used in a restricted form (see the end of this section). Definition statements that involve a range variable are performed for each possible value of the variable, by implicitly surrounding the statement with one for loop for each range variable used within the statement. For example:

```
i = 1..20;
v[i] = 0;
```

makes v a length 20 vector of zeros. It is equivalent to the pseudo-code:

```
for i = 1..20 {
  v[i] = 0;
}
```

Note that range variables behave just like any other variable with regards to EQ's unorderedness properties: code such as

‡ Use of a conditional expression to assign two values such as minimum and maximum would involve a notation using tuples such as (maximum, minimum) = if (x > =y) then (x, y) else (y, x).

```
i = 1..10;
...
a[i,j] = i+j;
...
j = 1..20:
```

is perfectly legal in EQ. This flexibility allows users to place range information where it is most convenient and understandable, and does not force them to place it directly before or after the code being looped over, as many other notations (such as explicit loops or array comprehensions) do.

We note that range variables are not equivalent to the vector subscripting available in languages such as APL and Sisal, where the semantics are equivalent to textual substitution. For example, the code

```
i = 1..20; j = 1..20;
x[i,j] = a[i] * b[j];
```

computes the "cross" product of vectors a and b (i.e., the full two-dimensional matrix ab^T); it is equivalent to the pseudo-code:

```
for i = 1..20 {
  for j = 1..20 {
     x[i,j] = a[i] * b[j];
  }
}
```

In contrast, the EQ statements:

```
i = 1..20;
x[i,i] = a[i] * b[i];
```

are equivalent to the pseudo-code:

```
for i = 1..20 {
  x[i,i] = a[i] * b[i];
}
```

which assigns only to the diagonal of x the product of corresponding elements of a and b (the meaning of such partial assignments to a variable in a single-assignment language will be clarified in Section 3.7). This distinction is not possible in a vector subscripting system, and requires additional constructs in other languages, such as the dot in Sisal or the transposition operator in APL. In addition to being more expressive, range variable constructions are easier to compile efficiently than vector subscripting.

EQ's range variables also have some advantages over the standard array comprehension notation used in many functional languages. The most important difference is the ability to reuse the same range for multiple computations, such as in

```
i = 1..n; j = 2..n-1;
x[i,j] = func1(i);
y[i,j] = func2(j);
z[i,j] = func3(i,j);
```

Because the definitions of x, y, and z use different subsets of the range variables, it would not be possible to express this code using a single comprehension. Instead, one comprehension would have to be written for each array. Writing such computations using comprehensions would require repeating the ranging information. In effect, EQ allows the user to create an abstract loop object, and to reuse it in a number of contexts within a given program.

Range variables can also be used in several other contexts within EQ as well, besides looping over a simple statement. For example, they can be used for performing reduction operations, such as a summation. The code

```
i = 1..20;
s = sum[i]: v[i];
```

adds up the elements of array v. The sum reduction operators (along with min, max, and several others) are built into the language definition itself. Our experience with numerical analysts suggests that user-defined reductions are not overly important to scientific computation. Providing built-in operators for these functions also makes the generation of high-quality executable code much easier.

Combining reductions with other uses of range variables can yield considerable expressive power:

```
i,j,k = 1..10;
c[i,k] = sum[j]: a[i,j] * b[j,k]
```

is a matrix multiplication program for 10×10 matrices (note that the sum operator binds more tightly than addition, but less tightly than multiplication. We believe this generally corresponds to mathematical notation).

In addition to summation, range variables can be used to find minimums and maximums, as well as the locations at which the extremum occurred:

```
i = 1..10; j = 1..20;
...
at max[i,j]: c[i,j]
  next c[i,j] = 0;
```

will replace the maximal element of c with zero (for more information on the next keyword, see Section 3.6). This construct more naturally expresses the programmer's intentions than would a loop like Fortran. Such a construct may be more elegant than using a location of maximum reduction operator.§

We note that range variables cannot be used to define recurrences; statements such as

```
x[i] = x[i-1] + 1;
```

are considered circular (x is defined in terms of itself) and are not permitted in EQ. This behavior contrasts with the behavior of EPL's subscript variables, where recurrences of this sort are legal, and in fact, are how all iterative computations must be specified. By avoiding statements such as these, an EQ compiler does not need to determine an appropriate order in which to loop through the range variable(s); any order (including fully parallel) will produce the correct results. When computations involving recurrences are needed, they can be expressed in EQ using its "over" loops (see Section 3.9).

3.5 Matrix Operations

As in Matlab, EQ provides the matrix operations that correspond with those usually used in standard mathematics. Thus, A * B, where A and B are two-dimensional matrices, represents a standard matrix-matrix multiplication, not the element-by-element computation that it would in APL or Fortran 90. We believe element-by-element multiplication is more clearly expressed by using subscripting and range variables instead of a matrix operator:

```
i = 1..n; j = 1..n;
C[i,j] = A[i,j] * B[i,j];
```

Currently, EQ supports addition ("+") subtraction ("−"), and multiplication ("*") of matrices, in addition to transposition, which is represented by the postfix "~" operator. We plan to eventually support less computational operations (such as matrix inverse) efficiently.

§ In order to use this sort of reduction style, tuples would have to be introduced. For example, the above example might read ''let (ii, jj) = maxloc (c [i,j] for i = 1..10, j = 1..20) in ...''.

3.6 Explicit Notations for Change

In an unordered notation, each name can represent only a single value; thus, a Fortran-style assignment statement such as x = x + 1 is not meaningful in such a context. To express such relationships in EQ, we turn once again to notation used for writing down algorithms that have iteration as a major component. In areas such as numerical optimization, one often finds notation such as:

$$x_+ = x_c + x_p$$

where the subscripts +, c, and p stand for the next value, current value, and previous value, respectively. EQ models this notation very closely:

```
next x = x + prev x;
```

is a statement that would be used in a program to compute a Fibonacci series. This notation corresponds more closely to the algorithmic ideas than Fortran code for the same computation, which would require a temporary to be introduced (and assignment statements carefully placed) to express this computation.

We note that there is nothing special about the current time; the above statement could also be written

```
x = prev x + prev prev x;
```

or as

```
next next x = next x + x;
```

provided that the rest of the code is modified similarly.

We expect that future EQ compilers will try to optimize away any extra storage associated with past copies of a variable. The issues are similar to those faced by the Sisal compiler, which has dealt with these issues successfully. We plan to use somewhat different techniques, which are discussed briefly in Section 6.5. These ideas have the advantage that they generalize nicely to cover cases where multiple past values are referenced, in addition to removing programmer-introduced storage inefficiencies.

In general, the next and prev prefixes allow references to past and future values of a variable. For convenience, the notation x′ can be used as an abbreviation for next x. Multiple prefixes can be used to reference values in the distant past or future (although this is rarely done). We show how to use these constructs to build programs that represent change in the following sections.

3.7 Followed-by Operator

In order to use the next and prev notations, there must be a way to move forward in time, so that the value that used to be called next x becomes x, x becomes prev x, and so on. EQ provides several mechanisms for incrementing the time step. The first is the followed-by operator "=>", which combines two individual statements A and B into a new statement A => B. Notice that this contrasts, for example, with the "---" barrier statement from Id, where the barrier applies to all statements within a pair of braces, not just the immediately adjacent ones. Informally, the statement A => B means to execute A, then move forward one unit in time and execute B. An example of this is:

```
...;
{x = 5; y = 2} =>
x = prev x * y;
...
```

which computes the value x = 10.

Generally, if we have the statement A => B, then the variables defined by A are accessible within B, but with one extra prev (or one fewer next) if B redefines that variable. Values defined by A but not redefined by B are accessed in B through their usual names (the variable y in the above example). The variables defined by A that are redefined by B are not accessible outside the followed-by statement (e.g., they appear either before or after B); i.e., only the last (textually) definition of a given variable is visible outside of the followed-by statement. In our example above, the value x=5 is not visible outside the followed-by statement, and could not be referenced in the sections of code labeled "..." in our example (uses of x would give the value 10 instead). This property ensures the side-effect-free nature of EQ's semantics, because the first assignment to the doubly assigned variable (x in our example) is only visible within the followed-by statement itself. As a result, execution of the followed-by cannot modify variables that are visible outside of the followed-by statement, only define new ones (such as x in our example, which has no other definition outside of the followed-by statement).

One additional aspect of the followed-by operator is important. If the statements in B make a partial definition of a variable (either by only defining some elements of an array or by defining a variable in only one branch of a conditional), then the undefined portions of that variable default to the value defined in A for the variable. This rule allows the followed-by operation to be used when an array must be built up out of several parts, because multiple definitions of the same array are not allowed. For example, one way of defining an n-by-n identity matrix is:

```
p = 1..n; q = 1..n;
I[p,q] = 0=>
I[p,p] = 1;
```

The first definition of I gives each element the value zero, and the second assigns ones to the diagonal. Since the second definition of I does not define all of I's elements, the values from the first definition are used to fill in the undefined spaces. Note that for all statements outside of the followed-by statement, the zero matrix assigned to I in the first part of the followed-by statement is inaccessible (although some of its zeros do "show through" the second definition). This notation (the partial assignment to I[a,a]) allows EQ to represent the change of only part of an object (e.g., the diagonal), a computation that many functional languages have trouble expressing at the language level.

The above example demonstrates in a practical example the contrast between EQ's range variables and subscripting with vectors, as is allowed in languages such as APL and Sisal. If the variables a and b were treated as ordinary vectors of integers, then I[a,a] would represent the entire matrix, not just its diagonal.

The followed-by operator is right associative; this makes code such as

```
a[1] = 1=>
a[2] = 4 =>
a[3] = 5;
```

have the correct behavior.

3.8 Approximation Loops—the do Statement and once Clause

Most loops in numerical codes iterate over a fixed set of integer values (like Fortran's DO loops). These loops are best described using EQ's range variables. Of the remaining loops in scientific computation, almost all fall into the category of approximation loops; these loops iterate, improving an initial approximation to some quantity on each execution of the loop, until either an adequate approximation to the answer is achieved or the approximation process fails. These loops often have multiple exit conditions, with different code that needs to be run for each exit case, and are not well modeled by standard while or repeat loops.

EQ supports approximation loops directly with its do statement. Each iteration of the loop advances time by one step. Consider the example:

```
x = n/2;
do {
   next x = (x + n/x) / 2;
   once (|next x - x| < 1e-7))
   sqrt = next x;
}
```

This program computes the square root of a number x and stores it in the variable sqrt. The first line initializes our loop. The do statement continues to compute values for next x and timeshifts them back into x until the boolean expression of the once statement becomes true (note the use of $|x|$ to compute absolute value), and which time sqrt is assigned the last computed approximation, which is next x (not x). Note that the once condition refers to both x and next x. In fact, any previous values of x can be referred to at any point within the loop body. For example, a statement such as

```
ave = (x + next x) / 2;
```

could be inserted anywhere inside the loop, including after the once statement, and the statement will compute the same values no matter where it is placed. This works because of EQ's unordered equations paradigm, and because the time-shift of next x into x does not occur until the loop itself iterates.

In general, a do loop body consists of an unordered set of statements that include one or more once clauses. The do loop repeatedly executes its body until one of its once clauses is triggered (has its condition become true). When this happens, the body of the once clause is executed, and the loop is finished. Note that a once clause has no effect on the current loop body execution; it only determines whether another time-shift is called for.

If multiple once clauses are used, their conditions are tested in order; the lexically first once clause that has a true condition is the one that is used. This is the only case in EQ where the order of statements can change the semantics of a program. We understand that this presence of ordered once clauses within unordered equations can be confusing; we hope to eliminate this problem in future versions of EQ.

3.9 Ordered Definite Iterators: The over Statement

Range variables cannot directly be used to write recurrences; code such as

```
i = 3..100;
fib [1] = 1 =>
fib [2] = 1 =>
fib [i] = fib [i - 1] + fib [i - 2];
```

is illegal; the last statement will be flagged by the EQ compiler as circular, because the last line defines fib in terms of itself. These kind of computations can be written using a do loop:

```
fib [1] = 1 =>
fib [2] = 1 =>
{
  i = 3;
  do {
    fib [i] = prev fib [i - 1] +
              prev fib [i - 2];
    i' = i + 1;
    once (i = 100) { }
  }
}
```

but the code is somewhat cumbersome.

To allow such computations to be written elegantly, EQ provides the over statement. This statement can be used to write an ordered loop (similar to the do loop), but which automatically loops over the values of a range variable. This statement permits us to write our example above as:

```
x = 3..100;
fib [1] = 1 =>
fib [2] = 2 =>
over x:
   fib [x] = prev fib [x - 1] +
             prev fib [x - 2];
```

which computes the first 100 elements of the Fibonacci sequence. The over statement runs its body as if it were a do loop; after each iteration, a time-shift is done, and x is changed to the next value in the range. Within the over body, the range variable is treated as an ordinary scalar variable, with the built-in definition

```
x = ⟨initial value in range⟩;
do {
  ⟨body of the over statement⟩
    next x = x + 1;
    once (x >= ⟨final value in range⟩)
         { }
}
```

In effect, the over statement "serializes" the implicitly parallel range variable.

By default, the over statement loops through the values of the range variable from smallest to largest; to reverse this direction, the keyword rev is placed before the variable name.

3.10 Summary

In summary, EQ borrows features and properties from both imperative and functional languages in an attempt to support rapid and natural construction of numerical programs. Its single-assignment nature and unordered definitions give it many of the advantages of functional programming, including a correspondence with standard mathematical notations. Range variable notation is an implicit looping construct that allows many computations on arrays to be expressed more simply than in systems that provide only element-wise or aggregate array operations. In addition to providing simple "doall" functionality, it also naturally extends to other operations, such as summations, minimizations, and ordered definite iteration. At the same time, EQ's explicit support for change, particularly of parts of an array, enables it to express algorithms that fundamentally involve the notion of updating values. The ability to easily describe such algorithms is a property it shares with the imperative paradigm.

4 EXAMPLE PROGRAMS

We present three examples of EQ programs, with explanations and observations about the EQ constructs used. The first example emphasizes the use of range variables, the second demonstrates iteration, matrix operations, and unordered equations, and the third leads into a discussion of dependability. All three examples include statements not implemented in the current EQ prototype (see Section 5). These examples show that the EQ constructs given in Section 3 allow a range of numerical algorithms to be expressed clearly and naturally.

4.1 LU Decomposition Example

The EQ in Figure 1 computes an LU decomposition with partial pivoting. The input matrix a is of size 1..n × 1..n, and the decomposition is computed "in place," and therefore returned in a′ (where a′ is the abbreviation for next a). The vector of pivoting information is stored in p′. For simplicity of presentation, our algorithm contains no singularity detection. The lines have been numbered so that we can refer to them. Line 1 sets up a range variable to loop over the entire matrix. Line 2 defines the initial state of the pivot vector p: Each row is in its original location. The assignment is done for each possible value of k. Line 3 begins the main loop. The over statement loops through each of the values of the range variable k, going from 1 to n. Unlike normal usage of a range variable, in an over statement the values are used sequentially. Each of lines 4 through 10 is executed once for each value of k, starting with 1 and going through n. These statements perform the pivoting and elimination steps. Lines 4 and 5 set up needed range variables for the computations at a single iteration of the LU decomposition; piv will be used to perform the pivot step, and i and j will perform the rank one update to a. Line 6 searches for the proper row with which to pivot, using the at statement, which goes through all of the values of piv, searching for the row with the biggest element in column k. Note the use of $|x|$ to

```
(1)     k = 1..n;
(2)     p[k] = k;

(3)     over k: {
(4)       i, j = k+1..n;
(5)       piv = k..n;

(6)       at max [piv]: |a[piv, k]| {
(7)         t1 = p[k] => p'[k] = p[piv] => p'[piv] = t1;
(8)         t2 = a[k,] => a'[k,] = a[piv,] => a'[piv,] = t2;
          } =>


(9)       a'[i,k] /= a[k,k] =>
(10)      a'[i,j] -= a[i,k] * a[k,j];
        }
```

FIGURE 1 LU decomposition example.

calculate the absolute value of x. Lines 7 and 8 perform the required exchanges, using the fixed value of `piv` that was computed by the `at` statement. Line 7 exchanges the pivoting information, and line 8 exchanges rows of the matrix. The followed-by operator is used because each set of three steps must be done in the specified order. Line 9 computes the multipliers, storing them in place. Note that this must be done after the pivoting, thus the followed-by operator is again used. Line 10 does the rank one update. Range variables are used to perform this operation without the use of loops. Since this operation uses the multipliers, the followed-by operator is placed between it and line 9.

This example makes very little use of one of the most important features of EQ, the ability to specify equations is an unordered manner. It does, however, illustrate the simplicity that comes from the index range features of EQ.

4.2 BFGS Example

The EQ code given in Fig. 2 finds a local unconstrained minimizer of an n-variable function, using a simple version of the popular BFGS quasi-Newton method. It takes as inputs the function `f ( )` and its gradient `gradf ( )`, as well as an initial estimate to the solution, `start_point`. The algorithm updates the inverse hessian, H, and uses a simple quadratic interpolation line search.

We note the heavy use of matrix notation in this algorithm. Lines 1 through 5 initialize the algorithm. The only use of range variables in this algorithm is in lines 5 and 6, to initialize H to the identity matrix. Lines 7 through 22 form the main loop, which produces better and better approximations to the minimizer of f.

Lines 8 and 9 compute the BFGS update. Note that they can be written at this point because EQ uses unordered equations. These lines will be evaluated by EQ only after the computations in line 18 have been completed, since `x'` and `gx'` are used in line 8.

Lines 10 through 18 form the line search which, like the loop within which it is nested, produces better and better approximations—this time to an acceptable step length lambda. Lines 12 and 13 compute the `x` and `fx` values for our current step length. Lines 14 through 16 compute the new trial step length. Line 17 checks the termination condition, and once the condition is satisfied, makes the trial values official in line 18.

Finally, we have the termination conditions for our main loop, of which there are two. The first, at lines 19 and 20, checks for small changes in function value. Notice that both `fx'` and `fx` are available for this test; the definition of `fx'` has no effect on `fx` until a time shift occurs because of a followed-by operator or `do` loop. If the test succeeds, the algorithm is successful, and we return our answer. The second termination test, in lines 21 and 22, checks for excessive iterations. As mentioned in Section 3.8, the once conditions are evaluated in their lexical order at each iteration of the loop.

This example illustrates the convenience and elegance of EQ's approximation loop facilities. Of particular note is the lack of temporary variables for holding "old" values of variables; the programmer was simply able to refer to `x'` and `x` at the same time. The example also shows the value of simple matrix operations, although that is not a feature unique to EQ. Finally, we have found that

```
            %Initialize the algorithm
(1)         iter = 1;
(2)         x = start_point;
(3)         fx = f(x);
(4)         gx = gradf(x);

(5)         i = 1..n;  j = 1..n;
(6)         H[i,j] = 0 => H[i,i] = 1;

(7)         do {

              % Inverse Hessian update
(8)           s = x' - x;  y = gx' - gx;
(9)           H' = H + ((s - H*y) * s~ + s * (s - H*y)~) / s~*y
                   - s*s~*(s - H*y)~*y/(s~*y)^2;

              % Step length computation
(10)          lambda = 1;
(11)          do {
(12)            trial_x = x - lambda*H*gx;
(13)            trial_fx = f (trial_x);

(14)            slope = -gx~*H*gx;
(15)            trial_lambda = -lambda * slope /
                        2 * (trial_fx - fx - lambda * slope);
(16)            lambda' = max (trial_lambda, lambda / 10);

(17)            once (trial_fx < fx + 1e-4 * slope) {
(18)              x' = trial_x;  fx' = trial_fx;  gx' = gradf (x');
                }
              }

              % Termination conditions
(19)          once (fx' - fx < 1e-8) {
(20)            answer = x';  success = TRUE;
              }

(21)          iter' += 1;
(22)          once (iter > 500) {
                success = FALSE;
              }
            }
```

FIGURE 2 BFGS unconstrained optimization example.

the use of unordered features demonstrated here corresponds to the way that optimization experts describe this algorithm. The placement of lines 8 and 9 early in the code highlights the heart of the algorithm, the BFGS update itself.

4.3 Nelder-Meade Simplex Example

We present EQ code for the Nelder-Meade simplex algorithm for unconstrained minimization in Figure 3. While somewhat less robust and slower than the BFGS algorithm, simplex algorithms do not require gradient information to be provided. In practice, simplex algorithms are used for some kinds of unconstrained minimization problems with small numbers of variables. Furthermore, they can also be useful in situations where computation of the function itself involves significant error. The routine in Figure 3 takes the dimension `n`, an `n` dimensional function `f ( )`, and an initial set of `n+1` points `s [, i]`, each of which is a vector of size `n`. The algorithm returns to its best approximation to the minimizer in the variable `result`.

The algorithm in a straightforward translation of the Nelder-Meade simplex algorithm. Line 1 sets up an array of function values for each of the points in the simplex. This array is created (rather than simply calling `f ( )` every time a function value is needed) for efficiency reasons. It is extremely difficult to optimize away the extra function calls in this program if this array is not provided. This is also true of currently popular languages such as Fortran or C.

Lines 2 through 33 are the major iteration, which attempts to improve the points in the simplex. Lines 4 through 9 compute the candidate points (`reflect`, `expand`, `contract`, `reflect_contract`) to update the worst point of the simplex (which must be in array position `n+1` because the simplex is sorted by function value). One of these is chosen based on its function values in lines 10 through 21. If none of these are a sufficient improvement, then the entire simplex is reduced in lines 22 through 24.

Then the simplex is sorted, using an insertion sort in lines 25 through 28, so that the points in the simplex are once again arranged in order of function value. We note that for typical applications, the number of variables (`n`) is very small, so the type of sort used does not affect performance significantly.

The termination conditions are given in lines 29 through 33. The first condition checks the largest difference between coordinates of the best point and the coordinates of all other simplex points. The second is a simple iteration limit.

The code given, if naively compiled, will exhibit several inefficiencies. First, multiple calls to the function `f` would be made with the same argument value (e.g., `f (reflect)` in lines 10, 13, 15, 16, and 18). These extra function calls can be optimized away fairly easily, using standard techniques. Second, the values of variables such as `contract` do not need to be computed on every iteration of the loop; only when their values are actually used (the `else` clause at line 19 is executed) does this definition need to be evaluated. This style of programming is encouraged by the unorderedness of EQ statements, and allows a significantly shorter source program. The sort of transformations required to address the efficiency issue that this raises are not commonly addressed in current languages, but we believe that they can be handled automatically by an EQ compiler. For more information on these transformations, see Section 6.

5 IMPLEMENTATION RESULTS

A prototype compiler for the EQ language has been implemented, which translates an EQ program into an equivalent C program, for compilation using a standard C compiler. We present some timing results comparing EQ programs wih hand-coded C versions of our LU, BFGS, and Nelder-Meade algorithms. These measurements were taken on a DECstation 5000/260 using its standard `cc` compiler, but we received similar performance ratios on other machines. The prototype EQ implementation does not currently support all of the EQ constructs used within the examples (in particular, the `at` and `over` statements). As a result, we have been forced to use alternative versions of these examples. We plan to add these constructs to our prototype in the future. Note that the missing features are necessary only for syntactic convenience, and including their implementation in our prototype should have no adverse impact on the performance measurements reported here.

For the LU algorithm, we computed 100 50-by-50 decompositions. For BFGS, we minimized a simple function of 64 variables—taking about 150 iterations of the main loop. The Nelder-Meade example optimized a more complex function for problem sizes from two to eight variables, each done 10 times.

In Table 1, we provide measurements of each

```
% Let fs be the value of the function at the simplex points
fs[i=1..n+1] = f(s[,i]);

% Begin the iteration
iter = 1;
do {

  % Define the relevant points
  centroid = (sum[i=1..n]: s[,i]) / n;
  reflect = 2*centroid - worst;
  expand = 2*reflect - centroid;
  contract = (worst + centroid) / 2;
  reflect_contract = (reflect + centroid) / 2;
  worst = s[,n+1];

  % Determine which point(s) to change in the simplex
  if (f(reflect) < fs[1] & f(expand) < f(reflect)) {
    s'[,n+1] = expand;
    fs'[n+1] = f(expand);
  }
  else if (f(reflect) < fs[n]){
    s'[,n+1] = reflect;
    fs'[n+1] = f(reflect);
  }
  else if (f(reflect) < fs[n+1] & f(reflect_contract) < f(reflect)) {
    s'[,n+1] = reflect_contract;
    fs'[n+1] = f(reflect_contract);
  }
  else if (f(reflect) >= fs[n+1] & f(contract) < fs[n+1]) {
    s'[,n+1] = contract;
    fs'[n+1] = f(contract);
  }
  else {
    s'[,i=2..n+1] = (s[,1] + s[,i]) / 2;
    fs'[i=2..n+1] = f(s'[,i]);
  } =>

  % Once finished, sort them
  over i=2..n+1:
    at min [j=1..i]: fs[j] {
      t1 = s[,i] => s'[,i] = s[,j] => s'[,j] = t1;
      t2 = fs[i] => fs'[i] = fs[j] => fs'[j] = t2;
    }

  % Check the termination conditions
  new_best = s'[,1];
  once (max[i=1..n, j=2..n+1]: |s'[i,j] - new_best[j]| <= 1e-8)
    result = new_best;

  iter' = iter + 1;
  once (iter > 500) {}
}
```

FIGURE 3 Nelder-Meade simplex example.

program compiled with optimization turned both on and off. The third column in the table contains the times of the EQ programs, while the fourth column shows the times of the hand-coded C implementation. The fifth column indicates the ratio of the EQ to C execution times. As Table 1 shows, the EQ programs are slower by a factor of 1.3 to 5.9, which we believe is reasonable for a first

Table 1. Comparison of Execution Times of Example EQ Programs against Hand-Coded C Versions

Problem	Optimization	EQ (s)	C (s)	Ratio (EQ/C)
PLU	OFF	9.0	3.4	2.6
	ON	4.7	0.8	5.9
BFGS	OFF	1.8	1.4	1.3
	ON	0.7	0.4	1.8
Simplex	OFF	34.1	13.6	2.5
	ON	15.1	5.5	2.7

implementation of a prototyping system. There is clearly room, however, for improvements in the efficiency of generated EQ programs.

Further investigation revealed that a large amount of the additional time required by the EQ version was caused by the calls to memcpy( ),|| which are produced by the EQ compiler. These calls are generated when a time step occurs in the EQ program (followed-by operations and do loops), and are used to copy the value of a variable into its previous variable. Many of these copies, however, were totally unnecessary and could be removed entirely. Others could be reduced in size to only one dimension. To examine the potential effects of copy elimination optimization, we removed the unnecessary memcpy( ) calls by hand from the code output by the EQ compiler, giving the results shown in Table 2.

In the optimized case, these EQ times are within 30 to 40% of the handwrittren C code. These timings are very promising for a first prototype containing very little optimization. The Nelder-Meade simplex algorithm is the worst of our test cases;

Table 2. Comparison of Execution Times of Example EQ Programs with memcpy()'s Removed against Hand-Coded C Versions

Problem	Optimization	EQ (s)	C (s)	Ratio (EQ/C)
PLU	OFF	4.8	3.4	1.4
	ON	1.0	0.8	1.3
BFGS	OFF	1.7	1.4	1.2
	ON	0.5	0.4	1.3
Simplex	OFF	26.2	13.6	1.9
	ON	7.8	5.5	1.4

|| The C library function memcpy( ) copies a block of memory from one location to another.

we suspect this is because the EQ version of the code computes some values that are not used in the computation (see Section 6.6). Overall, these data suggest that EQ programs can execute with efficiency very close to that of optimized C code, provided that the problem of copy elimination can be suitably solved. We discuss this and other optimization issues in the next section.

6 OPTIMIZATION AND DEPENDABILITY

Traditionally, many issues of implementation are avoided in language definitions, at least explicitly. On the other hand, truly successful languages such as Fortran, C, and C++ are very carefully designed so that efficient implementations can be provided, and thus implementation and optimization issues are implicit in their design. Such implicit treatment of implementation issues has drawbacks, however. For example, because different compilers provide different levels of optimization, to achieve the best performance, programmers often have to experiment with code sequences and examine the generated assembly instructions to achieve the best performance for a given compiler. Worse, because a programmer may not be able to rely on all compilers to provide a particular optimization (e.g., reusing a common subexpression), the programmer will be tempted to implement the optimization directly in high-level code, perhaps significantly increasing its complexity.

In order to prevent users from changing from representation *A* of a program to a less understandable representation *B* due to efficiency concerns, the language must guarantee that *A* and *B* will provide substantially the same performance for all implementations of the language. This leads us to propose the principle of language dependability—concrete guarantees of source program equivalences.

To provide dependability, the language definition must define explicitly what program equivalences the implementations of a language are required to provide. An application of this principle is the requirement in Scheme that all implementations perform tail-recursion elimination [2], thus making all tail-recursive programs equivalent to some nonrecursive program, at least to some degree. There are also similarities to the work of Skillicorn on congruence [19]. Congruence represents a guarantee of predictable performance on parallel machines as you move between different parallel architectures and different numbers of processors.

In other words, if the same source program is moved from one parallel machine to another, the performance changes are predictable. This is somewhat different from dependability, which guarantees equivalent performance of two different source programs on the same machine, but provides no cross-machine relationships.

We are particularly interested in applying the principle of language dependability to EQ for the following reasons:

1. Efficiency is a concern to scientific programmers. If a language feature cannot be provided along with some understanding of its expected performance consequences, then the feature is likely to remain unused by programmers who are concerned with performance.
2. EQ is intended to be used to compare prototype implementations of complex numerical algorithms. Because such comparisons are intended to identify the algorithms with the best performance, language implementation overheads that are not well defined will render such comparisons meaningless. Matlab [5] is an example of a language that does not meet this requirement. Efficiency of looping constructs in current implementations of Matlab is significantly less than that of array constructs. As a result, computations that do not conveniently fit into array form are penalized. In some extreme cases we've seen in Matlab, the difference in performance between array and nonarray programs for the same computation can be over a factor of 100.

To see how language dependability impacts the use of a language like EQ, consider the following code fragment from our Nelder-Meade example program:

```
if (f(reflect) < fs[1] &
    f(expand) < f(reflect)) {
. . .
}
```

It is clear that a naive implementation of the program will run the function f three times. Two of these will be passed the same argument (reflect), and therefore compute the same value (since EQ contains no side effects). The extra function call is unnecessary, and can be optimized away. Another way of looking at a language that provides this optimization in all cases is that it provides the programmer with the following rule:

Multiple computations of the same value by applying the same function or operator to the same arguments will not hurt efficiency; it is exactly equivalent to a program using a temporary variable to hold the intermediate value.

By the nature of being a guarantee, dependability is generally not easy to provide. Our current goal with regard to this issue is to identify the importance of this aspect of language design, and to do some preliminary investigations to determine how other language design decisions impact issues of dependability. While we see the principle of dependability as very important in the EQ design, we do not currently have sufficient experience with the implementation of EQ to definitively describe what source language equivalences EQ will be able to dependably provide.

In the remainder of this section, we list language features and/or programming styles that we hope to support in EQ, the optimizations that correspond to them, and our preliminary assessment of the viability of providing them dependably. Note that our focus here is on identifying what guarantees might be made to a programmer of the EQ language. We are not claiming that any of the suggested optimizations that support these dependability guarantees are new with this work. We only mention these optimizations to indicate that such guarantees can be provided with existing language and compiler implementation technology.

6.1 Irrelevance of Redundant Computations

As pointed out in the above example, the ability to reuse an expression rather than to create a temporary for it is an important language feature for prototyping. The full Nelder-Meade code becomes significantly more complex if the user must write temporaries for all of the appropriate function calls. Similar situations arise in many numerical algorithms.

Fortunately, since expressions in EQ have no side effects, this language feature (expression reuse) can be provided by the well-known common subexpression elimination (CSE) optimization. This optimization can be performed in reasonable time and eliminates redundant computations in all cases where the equivalence of two expressions is guaranteed by the equivalence of their parts.¶ This guarantee is a property that EQ shares with the functional programming paradigm, but imperative languages such as Fortran must perform more complex (albeit fairly standard) analysis to determine whether two expressions are actually redundant. As a result, the user of such a language may have difficulty recognizing when two expressions will be treated as redundant by any particular language or compiler. This implies that in these languages, the user might have to carefully choose those expressions that should be given temporary storage, and declare that storage appropriately. Even in cases where common optimization techniques can typically eliminate the unnecessary storage (such as simple scalar temporaries) from imperative languages, the lack of a guarantee to do so, and the very syntax and semantics of the language itself (the presence of storage declarations), is likely to compel users to choose their variables carefully. This extra user effort may occur even in cases where such decisions will not have any effect on the final output code.

6.2 Irrelevance of Temporaries

Irrelevance of temporaries means that replacing an expression with a temporary name, and defining that name elsewhere, does not have any effect on the efficiency or memory usage of the resulting executable. An example where this is a useful language property is given below (this time, taken from our BFGS example code):

```
trial_lambda = -lambda * slope /
              2 * (trial_fx - fx -
                   lambda * slope);
lambda' = max (trial_lambda,
               lambda / 10);
```

Here, the variable `trial_lambda` was simply introduced to shorten up the expression for `lambda'` and to make the program more readable. It also serves as a comment on the meaning of this part of the computation of `lambda'`. Unfortunately, doing so in many languages will result in wasted memory and slower execution.

By using an intermediate representation that is data flow oriented, the intermediate representations of programs that use temporaries will be exactly the same as those of equivalent programs that do not use temporaries. As a result, the final code produced will be the same for the two programs, and therefore, this optimization can be performed perfectly—the resulting executable will be exactly identical in the two cases.

¶ This includes such cases as f (x) and f (y), where x = y. This works because functions in EQ are side effect free.

6.3 Irrelevance of Full or Partial Copies

This property is in some ways a subset of the previous optimization, but it has additional implications. Consider the following code:

```
i = 1..n+1;
...
fs[i] = f(s[,i]);
best = s[,1];
if (f(reflect) < f(best)) ...
```

This example highlights the special problems of copies. Here, `best` is a partial copy of `s`. As a result, no storage is needed for `best`. Further, however, the expression `f(best)` is equal to `f(s[,1])`, which has already been computed in `fs[1]`. Therefore, an entire call to `f` can be avoided if the relationship between `best` and `x` is exploited.

Our example in fact comes from an early version of the Nelder-Meade simplex algorithm. To avoid the need for this optimization, the sample code in Section 4.3 replaces `f(best)` with `fs[1]`. This shows that partial copies are an important concept which must be optimized properly. The elimination of full copies (such as are created by the statement `a = b;`) is a fairly simple optimization, and we believe it will be possible to generalize the algorithm to handle partial copies. Once the partial copies are removed, the remaining issue is to deal with the partial CSE (in our example above, `f(best)` was equivalent to part of the variable `fs`). An extension of the CSE algorithm which keeps track of which parts of two objects have the same value (rather than treating values in aggregate) should allow it to handle this sort of optimization for almost all cases; we plan to investigate this optimization more thoroughly in future work.

6.4 Interchangeability of Matrix and Range Variable Notation

This language property is an attempt to avoid the tendency toward one liners that affects prototyping languages that have significantly faster performance when using their built-in matrix operators, such as APL and Matlab. Users of these languages are often forced into matrix notations, even when the computation could be more clearly described using control structures, because of efficiency or memory usage considerations.

To avoid this inefficiency, matrix notations should simply be a convenient shorthand for a scalar program involving looping notations (in the case of EQ, range variables). There should be no effect on speed or memory efficiency because of matrix notation, allowing users to choose the notation that corresponds to their mental conception of the problem.

EQ already handles this interchangeability by the simplest conceivable method; it translates all matrix operations into range variable notation during an early phase of the compilation process. Of course, it is still important to provide for the efficient execution of range variable notations.

6.5 Irrelevance of Extra Matrix Dimensions

Many times, a Fortran programmer will have a variable that conceptually has one or more dimensions, but will declare it to be of fewer dimensions. This can be done if the code can be arranged such that one part of the array is computed, its values are used, and then those parts of the array are no longer needed—so that they can be overwritten by the next section of the array. Examples of this are pervasive in many numerical codes (e.g., see [20]).

The process of manually controlling this overwriting makes programs harder to write and understand, and makes them further removed from their mathematical underpinnings. Support for arrays and matrix operations becomes difficult to use efficiently if these unnecessary matrix dimensions must be removed by the programmer. A good example of this is line 9 of the BFGS example in Section 4.2. The update to H' has been written in its natural matrix notation, but taken literally this will create several unnecessary `n by n` matrices. Writing the expression in a directly storage-efficient form, however, is difficult and unnatural, and obscures the mathematical meaning of the program. Thus, it is important for EQ to be able to optimize the temporary matrices created by such expressions down to their minimal sizes.

We refer to reducing the number of dimensions of an array's storage in memory as a rank reduction. This optimization cannot only improve storage efficiency, but can also produce execution speed improvements, if the storage for an array can be reduced to a scalar, and therefore be stored in a register. We give a simple example below:

```
i = 1..100;
a[i] = sin(i);
b[i] = a[i] * i;
```

In a naive implementation of this code, storage would be allocated for two 100-element real vectors. However, the storage for the array `a` is totally unnecessary; if we rewrite the code as

```
i = 1..100;
over i {
  a = sin(i);
  b[i] = a * i;
}
```

the same computation can be performed, using only a scalar quantity `a`. Furthermore, since `a` has been fully rank reduced (to a scalar), it can now be stored in a register, rather than being written to memory. This produces a significant execution speed savings.

This optimization can also be used as a major component of algorithms to ensure that statements such as

```
next x[i] = x[i] / 2;
```

allow `x` to be updated in place, without creating a separate array `next x`. The basic idea is that the storage for `next x` will be rank reduced to a scalar. Once this occurs, standard register allocation techniques can be used to place it in a register, eliminating the extra memory requirement altogether.

The rank reduction optimization is one of the key impediments to natural description languages for scientific computations that are array based, and is therefore an important problem to address. Unfortunately, the process of trying to choose which rank reductions to perform to achieve optimal results is very difficult, and is NP hard in the general case, as shown in EPL [17]. However, we hope to formulate some more restrictive rules under which a perfect optimization is feasible, such as reducing all objects that can be fully rank reduced or minimizing the maximum number of dimensions used by any array in the program. This aspect of EQ optimization will be an interesting facet of future research.

6.6 Irrelevance of Equation Placement

This property may sound somewhat redundant, since EQ already offers unordered equations which allow users to group their equations in the most natural and convenient way. However, the movement of equations into and out of control structures can have an effect on efficiency, even when the semantics of the program are unchanged. For example, consider the following code fragment (taken from the Nelder-Meade simplex algorithm in Section 4.3):

```
contract = (worst + centroid) / 2;
...
if ...
else if ...
else if ...
else if (f(reflect) >= fx[n+1] &
         f(contract) < fs[n+1]) ...
```

Here, the current implementation will evaluate `contract` on each iteration through the outer loop, even though it is only needed if the first three `if` clauses are false. The code could be more efficiently written as:

```
if ...
else if ...
else if ...
else {
  contract = (worst + centroid) /2;
  if (f(reflect) >= fx[n+1] &
  f(contract) < fs[n+1]) ...
}
```

This version solves the efficiency problem, but the code is more complex. Furthermore, it can be very cumbersome to do this optimization by hand in the general case. Another example of this sort of code movement is the lifting of invariant computations out of loops, which is a classical optimization technique.

We hope that future versions of EQ will make the placement of statements have as little effect on the speed or storage usage of compiled EQ programs as possible. Traditional code-motion optimizations can be used to achieve some of these effects in imperative languages. It may also be possible to extend these results to functional languages as well, in which case we will use them. We anticipate that there may be more opportunities to perform these optimizations in EQ code than in imperative languages due to EQ's single-assignment property and unordered semantics.

7 FUTURE WORK

The current prototype EQ implementation does not support syntactic conveniences such as the `at` and `over` operations. Correcting these deficiencies is one of the top priorities for our future work in this area. Once this is done, it will be possible to do further examples of EQ programming of a more lengthy and complex nature than those given here. We hope to use these experiences to continue to evolve and change the constructs of the EQ language.

Another area of interest is the efficient support for more complex matrix operations, such as matrix inversion, and support for some kinds of structured matrices, such as triangular and diagonal matrices. We hope to be able to add these kinds of constructs to EQ, using compile-time analysis to avoid any loss of efficiency. As an example of the sorts of issues that occur in providing matrix operators, consider an operator for inversion. While very nice from a mathematical point of view, very few scientific programs calculate an actual inverse matrix; instead, they use some sort of decomposition, such as an LU or Cholesky factorization. Then backsolves are used to complete the computation. Thus, a good implementation of matrix inverse in EQ would only compute the actual inverse in those cases where it had to, and would simply use a factorization otherwise.

Additionally, a study and implementation of specific transformations, including those discussed in Section 6, are important both to obtain an understanding of the impact of the EQ language constructs on efficiency, and to determine the effectiveness of language properties such as expression reuse and invariance to statement placement. This work will involve examining the typical effectiveness of the transformations (i.e., how much they improve performance) and looking at dependability issues. Due to its single assignment and unordered semantics, EQ is a rich environment for studying such techniques.

Finally, a detailed study of the opportunities for parallelism in EQ is justified. EQ has much potential data parallelism exposed directly to the user of the language through its range variables. Furthermore, its unordered equations express functional parallelism. The inherently parallel nature of these features suggests that a parallel version of EQ could be quite natural and easy to understand when compared with languages based on the basically nonparallel imperative paradigm. We plan to continue research into both implicit and explicit models of parallelism in EQ.

8 CONCLUSION

The purpose of this research is to propose and investigate a new programming language design based on identifying the abstractions used by numerical analysts when they solve problems. Our language, EQ, provides several important features that attempt to directly reflect the syntactic and semantic concepts expressed by numerical analysts whom we interviewed. Specifically, we designed EQ to allow scientific programmers to rapidly prototype complex new algorithms using a language that is both efficient (where optimizing transformations are applied in a dependable manner) and expressive.

In particular, our language supports unordered equations, large-grained state transitions, and high-level matrix notation. Unordered equations support defining a set of values and their relations. Large-grain state transitions allow programmers to express a form of structured large-scale change that corresponds to the concept of iteration or recurrence. Our matrix notation supports range variables, a powerful implicit looping construct. As a language, EQ falls in between the functional and imperative programming paradigms, providing elements of both. While EQ contains elements found in a number of diverse programming languages, including Sisal, Id, and EPL, no other language brings these elements together in the same way.

In EQ we also introduce the general principle of language dependability. This principle states that language definitions should specify what program transformations the implementations of the language are required to perform. Dependability is necessary because without such a guarantee in the definition, programmers are unlikely to assume such transformations will be done, and will potentially greatly complicate their code in an effort to make it more efficient. Dependability is especially important in a language like EQ because one goal of our design is to provide a high-level language that programmers can use to compare the performance of complex algorithms.

While dependability is an important principle to support in design, the logistics of supporting it are quite difficult. In this article, we have identified a set of transformation guarantees that appear possible with existing implementation technology. In the future, we intend to investigate these issues more closely and completely specify which guarantees we are able to provide.

We plan to refine the design of EQ and experiment more with its implementation. Our experiences with EQ have shown us the value of domain-specific language design and demonstrated that interesting new styles of expressing computations can arise from interacting with programmers in different application domains (e.g., that of numerical computation).

ACKNOWLEDGMENTS

Research supported by NSF grant ASC-9307315.

REFERENCES

[1] S. C. Johnson, "Compiling MATLAB," in *Proc. of the Symposium on Very High Level Languages*, pp. 119–128.

[2]. J. Rees and W. Clinger, "The revised^3 report on the algorithmic language Scheme," *ACM SIGPLAN Notices*, Vol. 21, pp. 37–79, Dec. 1986.

[3] ANSI, *American National Standard Programming Language Fortran ANSI X3.9-1978*. ANSI, 1978.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Engle Cliffs, NJ: Prentice-Hall, 1978.

[5] Math Works Inc., *Matlab User's Guide*, Math Works, Inc. 1992.

[6] K. E. Iverson, *A Programming Language*. New York, Wiley, 1962.

[7] ISO, *Fortran 90 Standard ISO/IEC 1539: 1991(E)*. ISO, 1991.

[8] B. Stroustrup, *The C+ + Programming Language*. Reading, MA: Addison-Wesley, 1986.

[9] D. Cann, "Retire Fortran? A debate rekindled," *Commun. ACM*, Vol. 35, pp. 81–89, Aug. 1992.

[10] J. Backus, "Can programming be liberated for the von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, Vol. 21, pp. 613–641, Aug. 1978.

[11] P. S. Barth, R. S. Nikhil, and Arvind, "M-structures: Extending a parallel, non-strict, functional language with state," In *Proc. on Functional Programming and Computer Architecture*, 1991, p. 538.

[12] P. Hudak, et al., "Report on the programming language Haskell: A non-strict, purely functional language version 1.2," *SIGPLAN Notices*, Vol. 27, pp. R-1—R-164, May 1992.

[13] D. Turner, "An overview of Miranda," in *Research Topics in Functional Programming*, D. Turner, Ed. Reading, MA: Addison-Wesley, 1990, pp. 1–14.

[14] A. P. W. Bohm, R. R. Oldehoeft, D. C. Cann, and J. T. Feo, *SISAL Reference Manual, Language Version 2.0*.

[15] R. S. Nikhil, *Id Language Reference Manual, Version 90.1*. Postscript available via FTP from Massachusetts Institute of Technology, Boston, MA, July 1991.

[16] W. W. Wadge, *Lucid, the Dataflow Programming Language*. London: Academic Press, 1985.

[17] B. K. Szymanski, "EPL—parallel programming with recurrent equations," in *Parallel Functional Languages and Compilers*, B. K. Szymanski, Ed. New York: ACM Press, 1991.

[18] S. A. Smith, T. O. Levante, B. H. Meier, and R. R. Erns, "Computer simulations in magnetic resonance: An object-oriented programming approach," *J. Magnetic Resonance*, Vol. 106A, pp. 75–105, Nov. 1993.

[19] D. B. Skillicorn, "Models for practical parallel computation," *Int. J. Parallel Program.*, Vol. 20, pp. 133–158, April 1991.

[20] D. Olander and R. B. Schnabel, "Preliminary experience in developing a parallel thin-layer Navier Stokes code and implications for parallel language design," University of Colorado at Boulder, Tech. Rep. CU-CS-;582-92, Feb. 1992.