Normalize, Transpose, and Distribute: An Automatic Approach for Handling Nonscalars

DANIEL E. COOKE, J. NELSON RUSHTON, BRAD NEMANICH,
ROBERT G. WATSON, and PER ANDERSEN
Texas Tech University

SequenceL is a concise, high-level language with a simple semantics that provides for the automatic derivation of many iterative and parallel control structures. The semantics repeatedly applies a "Normalize-Transpose-Distribute" operation to functions and operators until base cases are discovered. Base cases include the grounding of variables and the application of built-in operators to operands of appropriate types. This article introduces the results of a 24-month effort to reduce the language to a very small set of primitives. Included are comparisons with other languages, the formal syntax and semantics, and the traces of several example problems run with a prototype interpreter developed in 2006.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Programming Language**]: Language Classification—*Applicative (functional) language*; D.3.3 [**Programming Language**]: Language Constructs and Features

General Terms: Languages

Additional Key Words and Phrases: Automatic parallelisms, automatic loop generation

ACM Reference Format:

Cooke, D. E., Rushton, J. N., Nemanich, B., Watson, R. G., and Andersen, P. 2008. Normalize, transpose, and distribute: An automatic approach for handling nonscalars. ACM Trans. Program. Lang. Syst. 30, 2, Article 9 (March 2008), 49 pages. DOI = 10.1145/1330017.1330020 http://doi.acm.org/10.1145/1330017.1330020

1. INTRODUCTION

High-level functional languages, such as LISP and Haskell, are designed to bridge the gap between a programmer's concept of a problem solution and the

This research was supported by NASA-NNG06GJ14G, NASA-CAN NNJ06HE94A, and Abilene Community/Sheldon Fund grant 146144C571.
Authors' address: Department of Computer Science 8th and Boston, Texas Tech University, Lubbock, TX 79409; email: {daniel.cooke, nelson.rushton, brad.nemanich, robert.g.watson, per.andersen}@ttu.edu.

realization of that concept in code. In these languages, a program consists of a collection of function definitions, roughly isomorphic to their counterparts in nonexecutable mathematical notation. The language semantics then generate the data and control structures necessary to implement a solution. A great deal of the complexity of execution remains hidden from the programmer, making it easier and faster to develop correct code.

The ease and reliability afforded by high-level languages often comes at a cost in terms of performance. Common wisdom says that if a software product must perform better in terms of speed and memory, it must be written in a lower level language—with arcane looking optimized assembly code at the extreme end.

Over time, however, the trend is for more software to be developed in higher level languages. There are two reasons for this. The first is obvious: machine performance tends to improve over time, bringing more applications within the realm where high-level implementations, though perhaps slower than their low-level counterparts, are fast enough to get the job done. In other words the human costs in creating problem solutions are increasingly greater than the cost of the machines that carry them out.

The second reason is slightly less obvious: while the intelligence of human programmers in writing low level algorithms remains roughly constant over time, the intelligence of automatic code generators and optimizers moves forward monotonically. Today, we are beginning to see examples where a few lines of high-level code evaluated by a sophisticated general-purpose interpreter perform comparably with hand written, optimized code (e.g., see Lin and Zhao [2004]). This happens because optimization is accomplished at the level of the compiler, rather than on individual programs—focusing the optimization efforts of the programming community in one place, where they are leveraged together on a reusable basis.

Improvements in the performance of high-level codes may follow a course similar to what we have recently seen in the performance of chess programs. Not that long ago, these programs only occasionally defeated competent amateur human players. Swift progress in this domain has led to the current situation in which chess programs are outperforming the chess masters. Looking forward by this analogy, we expect the performance of high-level languages to continue to gain ground against their low-level counterparts. However, for reasons explained below, *we claim that most languages in existence today are not in a position to participate fully in this trend.*

The impressive optimization results in code generators cited in Lin and Zhao [2004] were obtained using the logic programming language A-Prolog [Gelfond and Lifschitz 1988, 1991], which has two distinguishing features:

(1) Its semantics are purely declarative, containing no commitments whatsoever regarding the data structures or algorithms underlying the execution of its programs, and
(2) The performance of the language has substantially exceeded the expectations of its designers and early users.

In light of the preceding discussion, we claim it is no coincidence that these two unusual traits are found together in the same language. Ironically, Item #1 effectively blocks programmers from optimizing their code. Though the A-prolog programmer knows what the output of his program will be, he cannot control, or even know on a platform-independent basis, how the results will be obtained. On the other hand, it is precisely this feature which frees the hands of the compiler-designer to effect optimizations. Without constraints on the representations or algorithms deployed, the creativity of those who implement a new language is unleashed. And it is precisely *this* feature, which allows them to write compilers in which concise, readable "executable specifications" can perform comparably with, or better than, handwritten algorithms.

Few other languages have taken the same path. Though their code is written at a high level of abstraction, the semantics of languages such as Haskell and Prolog make guarantees about their mechanisms of representation, computation, and inference. This approach has the advantage of allowing programmers to understand, control, and optimize the representation and execution of their programs. But it ties the hands of the designers of interpreters and compilers, bounding their capability to deploy and combine optimizations at a more general level.

This situation is one of mindset resulting in a tacit agreement among the language designers who provide low level semantics and the programmers who employ the languages. Even with a high-level language such as Haskell, programmers tend to perceive, for example, the list structure, as shorthand for a particular low-level representation. They make substantial efforts to optimize their code with respect to this representation, and compiler-designers deploy optimizations in anticipation of this mindset. Thus the programming community has an implicit (and in many places explicit) commitment to viewing programming constructs as a notation for objects related to algorithms and machine architecture—even in supposedly high-level languages.

Our position is that in many problem domains it is time, or will soon be time, for programmers to *stop* thinking about performance-related issues. This does not mean we discount the role of optimization and program performance. On the contrary, program performance is crucial for many problem domains, and always will be; and this makes it important to attack the problem by focusing efforts in places where they can be most effectively combined and reused—which is at the level of the compiler or interpreter. Then in 'ordinary' programs, the burden of optimization can be passed off to the compiler/interpreter, possibly with 'hints' from the programmer.

For more than a decade, we have been developing SequenceL, a Turing-complete, general-purpose language with a single data structure, the Sequence. [Cooke 1996, 1998] The goal of this research has been to develop a language, which allows a programmer to declare a solution in terms of the relationship between inputs and desired outputs of a program, and have the language's semantics "discover" the missing procedural aspects of the solution. The key feature of the language is an underlying, simple semantics termed Consume-Simplify-Produce and the Normalize-Transpose-Distribute. It is still an open question how well such a high-level language can perform. We conjecture that the performance of SequenceL can eventually equal or exceed performance of lower level languages such as C and C++ on average. This articles describes SequenceL and its semantics, and gives performance data on a commercial-scale application, which serves as a step toward substantiating this conjecture. We focus on SequenceL's NTD (normalize-transpose-distribute) semantic, which we envision as a substantial component of reaching this goal. We informally explain the NTD semantics, and compare it to similar constructs defined in related work. We also give a formal syntax and semantics of the current version of SequenceL, including NTD, show Turing Completeness of SequenceL, and illustrate its use with examples. Throughout this article, code and other information entered by a programmer are shown in `Courier` and traces of execution are shown in Times font.

2. MOTIVATING EXAMPLES AND INTIUTION ON SEMANTICS

Iterative and recursive control structures are difficult and costly to write [Mills and Linger 1986; Bishop 1990]. In some cases, these control constructs are required because the algorithm being implemented is intuitively iterative or recursive—say implementing Quicksort. However, most uses of iteration and recursion are not of this type. *Our experience is that in the majority of cases, control structures are used to traverse data structures in order to read from or write to their components.* That is, the control structures are typically necessitated not by intuitively iterative or recursive algorithms, but by the nonscalar nature of the data structures being operated on by those algorithms.

For example, consider an algorithm for instantiating variables in an arithmetic expression. The parse tree of an expression, for example,

$$x + (7 * x)/9$$

can be represented by a nested list (here we use the SequenceL convention of writing list members separated by commas, enclosed in square brackets):

$$[\mathtt{x}, +, [7, *, \mathtt{x}], /, 9]$$

To instantiate a variable, we replace all instances of the variable with its value, however deeply nested they occur in the parse tree. Instantiating the variable x with the value 3 in this example would produce

$$[3, +, [7, *, 3], /, 9]$$

Here's LISP code to do the job:

```
(defun instantiate (var val exp)
  (cond
    ( (and
        (listp exp)
        (not (equal exp nil)))
      (cons
         (instantiate var val (car exp))
         (instantiate var val (cdr exp))))
   ( (equal exp var) val)
   (t exp)))
```

Prolog gives a somewhat tighter solution:

```
instantiate(Var, Val, [H|T], [NewH|NewT]):-
       instantiate(Var, Val, H, NewH),
       instantiate(Var, Val, T, NewT).
instantiate(Var, Val, Var, Val).
instantiate(Var, Val, Atom, Atom).
```

Next We Have a Solution in Haskell:

```
inst v val (Seq s) = Seq (map (inst v val) s)
inst (Var v) val (Var s)
    | v==s = val
    | otherwise = (Var s)
inst var val s = s
```

We note three things about this algorithm and its implementation. First, our intuitive conception of the algorithm—"replace the variable with its value wherever it appears"—is trivial in the sense that it could be carried out by any schoolchild by hand. Second, explicit recursion does not enter into our basic mental picture of the process (until we have been trained to think of it this way). Third, the use of recursion to traverse the data structure obscures the problem statement in the LISP, Prolog, and Haskell codes.

Often, as in this example, the programmer envisions data structures as objects, which are possibly complex, but nevertheless static. At the same time, he or she must deploy recursion or iteration to traverse these data structures one step at a time, in order to operate on their components. This creates a disconnect between the programmer's mental picture of an algorithm and the code he or she must write, making programming more difficult. SequenceL attempts to ease this part of the programmer's already-taxing mental load. Our desideratum is this: *if the programmer envisions a computation as a single mental step, or a collection of independent steps of the same kind, then that computation should not require recursion, iteration, or other control structures.* In the remainder of this section, we will give code for two functions illustrating how this point can be achieved in SequenceL, and then, in the following sections, describe the semantics that allow the functions to work as advertised.

Variable instantiation, discussed above, is written in SequenceL as follows.

```
instantiate(scalar var,val,char) ::=
      val when (char == var) else char
```

The SequenceL variable instantiation solution maps intuitively to just the last two lines of the Prolog and LISP codes, and last three lines of Haskell, which express the base cases. This is the real meat of variable instantiation—the "schoolchild's picture" of the algorithm. The remainder of the LISP, Haskell, and Prolog code is dedicated to traversing the breadth and depth of the tree. This is the majority of the code, line for line, and the tricky part to write. When called on a nested data structure, SequenceL, in

```
Function matmul(Consume(s_1(n,*),s_2(*,m))),Produce(next)
            where next(i,j) =
       {compose([+([*(s_1(i,*),s_2(*,j))])])}
Taking[i,j]From cartesian_product([gen([1,...,n]),gen([1,...,m])])
```

Fig. 1. Prior version of SequenceL.

contrast, will traverse the structure automatically, applying the function to sub-structures of appropriate type. Furthermore, SequenceL will execute this problem solution using one simple rule, the *normalize-transpose-distribute*. NTD is repeatedly applied to any SequenceL construct and the data upon which it operates, until the data matches the type of argument expected by the function.

For another example, consider matrix multiplication. Here is a Haskell version:

```
matMul:: [[Integer]] -> [[Integer]] -> [[Integer]]
matMul a b = [[dotProd row col | col <-transpose b] | row<-a]

dotProd:: [Integer] -> [Integer] -> Integer
dotProd x y = sum [s * t | (s,t) <- zip x y]
```

and the corresponding SequenceL:

```
mmrow(vector a, matrix b) ::= dotProd(a,transpose(b))

dotProd(vector x,y) ::= sum(x* y)
```

Note once again the following observation: to a first approximation, the SequenceL code can be obtained from the Haskell code by *erasure of the Haskell syntax related to the traversal and composition of data structures*. In particular, here SequenceL eliminates the need for "dummy variables" (i.e., *row, col, s,* and *t*, which play the role of looping variables in procedural code), as well as the comprehension and "zip" constructs.

A couple of additional points concerning the SequenceL function warrant attention. First, compare the new version of the problem solution (seen immediately above) with the old, seen in Figure 1.

Figure 1 presents the form of the SequenceL solution as it was defined in Cooke and Andersen [2000]. The comparison between the solution today and the older version serves as a good, representative example of the simplifications and improvements made in the language during the last few years. The NTD semantics shield the programmer from having to develop much of the procedural aspect of a problem solution. As the reader will see, these semantics are surprisingly simple to understand.

2.1 The Consume-Simplify-Produce Semantics

In its present form, SequenceL has no facility for variable assignment and no input-output other than an ability to provide and inspect initial and final *tableaux*, respectively. Thus, SequenceL is a simple and pure functional language—capable of executing typical functional programs. For example, a recursive SequenceL solution to find the greatest common divisor appears below.

```
gcd(scalar m,n) ::=
      gcd(m-n, n) when m > n else gcd(n,m) when m < n else n
```

A special sequence, called a *tableau,* provides a workspace for the evaluation of SequenceL terms. To evaluate the function, above, one establishes an initial *tableau*, which references the function and supplies arguments (e.g., *gcd(200,100)*).

Given an initial tableau an evaluation step, called a Consume-Simplify-Produce (CSP) step is performed. A CSP step *consumes* the tableau, *simplifies* its contents, and *produces* the simplified result in the next tableau. For example, a single CSP step for the *gcd* function is shown below:

$$INITIAL = gcd\ (200, 100)$$
$$CSP = gcd(200 - 100, 100)\ \textbf{when}\ 200 > 100\ \textbf{else}\ gcd(100, 200)\ \textbf{when}\ 200 < 100\ \textbf{else}\ 100$$

In the example case, the simplification step simply grounds the variables of the function, leaving the grounded function body in the next tableau. The *CSP* in the trace so far represents one *Consume-Simplify-Produce* step. The subsequent *CSP* steps are:

$$CSP = gcd(200 - 100, 100)$$
$$CSP = gcd(100, 100)$$
$$CSP = gcd(100 - 100, 100)\ \textbf{when}\ 100 > 100\ \textbf{else}\ gcd(100, 100)\ \textbf{when}\ 100 < 100\ \textbf{else}\ 100$$
$$CSP = gcd(100, 100)\ \textbf{when}\ 100 < 100\ \textbf{else}\ 100$$
$$CSP = [100]$$
$$FINAL = [100]$$

The complete evaluation of *gcd(200,100)* is the concatenation of the *Initial*, the *CSP*, and the *Final* steps above. Notice that evaluation of tableaux continues until a fixpoint in evaluation is achieved.

The tableau is viewed as any other sequence in SequenceL. Sequences can be structured and contain any constants or SequenceL term—including conditional terms (i.e., function bodies). There are no restrictions on the manner in which terms can be combined. Consider the following function to check the boundaries on subscripts:

```
sub(? x, scalar i) ::= x(i when i>=1 and i=<length(x) else
                                   subscript_error)
```

The parameters in the function signatures are typed according to dimension or level of nesting. Types include **scalar, vector**, **vector(vector)**, etc. The type **?** specifies an argument of any dimension. Given an initial tableau of *sub([12, 3, 4, 5], 3)*, CSP steps lead to *x(3)* or *[12, 3, 4, 5](3),* which ultimately produces *4*. The complete set of evaluation steps are:

INITIAL = sub([[12, 3, 4, 5], 3])
CSP = [12, 3, 4, 5](3 ***when*** *(3 >= 1)* ***and*** *(3 =<* ***size*** *([12, 3, 4, 5]))* ***else*** *subscript_error)*
CSP = [12, 3, 4, 5](3 **when** *true* **else** *subscript_error)*
CSP = [12, 3, 4, 5](3)
CSP = 4

FINAL = [4]

Notice that *sub([10, 20, 30, 40], 7)* leads to *[10, 20, 30, 40](subscript_error).*

2.2 Normalize-Transpose-Distribute Semantics and Overtyping

The Normalize-Transpose-Distribute semantics are based upon the idea of *overtyping*. Overtyping occurs when an operator or function encounters operands or arguments that are of higher level of nesting (or dimension) than expected. For example, arithmetic operators are defined to operate on scalars. Therefore the sequence *[2 + 2]* gives the expected result, *4*. If the expression *[1,2,3] * 2* is to be evaluated, the following CSP steps are followed:

*INITIAL = [1, 2, 3] * 2* *ntd*
*CSP = [[1 * 2], [2 * 2], [3 * 2]]*
CSP = [2, 4, 6]
FINAL = [2, 4, 6]

Since the multiply also expects scalars, an NTD is performed as the simplification step of the CSP. The NTD includes a normalize, which makes *3* copies of the scalar *2*, since the nonscalar argument has *3* elements. This results in *[1,2,3] * [2,2,2].* A transpose on the arguments is performed, resulting in *[[1,2],[2,2],[2,3]].* Now the operator can be distributed among the binary scalars, resulting in *[[1 * 2], [2 * 2], [3 * 2]],* which supplies the multiplication operator the scalar operands for which it is defined. The final CSP step above obtains the desired product.

Overtyping exists anytime an operator has operands greater than the expected nesting level. In the example above the multiplication operator was overtyped by *1* level (i.e., one of the operands was a one-dimensional sequence of scalars, rather than a scalar as expected). Consider the situation when the plus operator acts on a *three-dimensional* sequence. In cases such as this, the

NTD and the CSP interact in a manner resulting in nested NTDs.

$$\begin{aligned}
&INITIAL = [[[1, 1, 1], [2, 2, 2]], [[11, 11, 11], [12, 12, 12]]] \\
&\qquad\qquad + [[[1, 1, 1], [2, 2, 2]], [[11, 11, 11], [12, 12, 12]]] \\
&CSP = [[[[1, 1, 1], [2, 2, 2]] + [[1, 1, 1], [2, 2, 2]]], [[[11, 11, 11], [12, 12, 12]] \\
&\qquad + [[11, 11, 11], [12, 12, 12]]]] \\
&CSP = [[[[1, 1, 1] + [1, 1, 1]], [[2, 2, 2] + [2, 2, 2]]], [[[11, 11, 11] \\
&\qquad + [11, 11, 11]], [[12, 12, 12] + [12, 12, 12]]]] \\
&CSP = [[[1 + 1, 1 + 1, 1 + 1], [2 + 2, 2 + 2, 2 + 2]], [[11 + 11, 11 + 11, 11 \\
&\qquad + 11], [12 + 12, 12 + 12, 12 + 12]]] \\
&CSP = [[[2, 2, 2], [4, 4, 4]], [[22, 22, 22], [24, 24, 24]]] \\
&FINAL = [[[2, 2, 2], [4, 4, 4]], [[22, 22, 22], [24, 24, 24]]]
\end{aligned}$$

The interaction of CSP and NTD in this trace results in adding corresponding elements of two three-dimensional structures.

When operators and functions are defined in SequenceL, type information is provided. The types indicate the dimension of an argument. The question mark**?**, the words **scalar**, **vector, vector(vector)** or **matrix**, **vector(matrix)**, etc. in function signatures indicate the type of structure the function expects. A question mark allows any structure, a **scalar** is order zero, a **vector** order one, a **vector(vector)** or **matrix** is order two, etc. For a parameter **P**, in a function's signature and a corresponding argument **A**, the following indicates the conditions under which **A** is overtyped, based on the order of **A**:

P's type	P's order	A's order
scalar	0	order(A) > 0
vector	1	order(A) > 1
vector(vector)	2	order(A) > 2
etc.		

If **P** is typed with the **?** then **A's** order can be any $\mathbf{N} \geq \mathbf{0}$ (i.e., there is no situation in which **A** is overtyped). A **vector(vector)** can be a vector containing a mixture of scalars and at least one vector or the special case **matrix**. An undertyped argument, or error, occurs whenever:

P's type	P's order	A's order
vector	1	order(A) < 1
matrix	2	order(A) < 2
vector(matrix)	3	order(A) < 3
etc.		

When provided arguments of the order declared, a function or operator is evaluated. When provided an overtyped argument, NTDs result. When provided an undertyped argument, a type error occurs. The following section provides an informal definition of the NTD semantics and more advanced examples of its effect.

2.3 Simple Translation Involving Normalize-Transpose-Distribute

Here we present informal definitions of the NTD semantics, which enable the shortcuts seen above, in more detail. We first define the NTD operations on sequences ("sequences" will be defined rigorously in Section 5.4; but for now we can think of them as ordered multisets).

Let E be a sequence of length L, and for $i \leq L$ let $E(i)$ denote the ith member of E. Let S be a subset of $\{1, \ldots, L\}$, and for all i in S suppose $E(i)$ are sequences of the same length L' (though E may contain other sequences of similar length, not in S). For any natural number n and any e, let *repeat(e,n)* denote the ordered multiset consisting of n copies of e. From E, we obtain the *normalization* of E *with respect to S,* denoted by, *normalize(E,S),* by replacing $E(i)$with *repeat(E(i), L')* for all i not in S, and leaving $E(j)$ unchanged for i in S. Intuitively, normalization is used to make all the arguments conformant in terms of number of elements. For example:

$$\textit{normalize}\,([[1,2],[2,3,4],5],\{1\}) = [[1,2],[[2,3,4],[2,3,4]],[5,5]].$$

Notice that the second and third members of the original sequence are repeated twice each, because the length of the first member, with respect to which we are normalizing, is 2. For another example,

$$\textit{normalize}([[1,2],3,[4,5],[6,7]],\{1,4\}) = [[1,2],[3,3],[[4,5],[4,5]],[6,7]].$$

For any sequence E whose members are sequences of the same length, the *transpose* of E consists of a sequence of all the first components of members of E, followed by a sequence of all the second components, etc. For example,

$$\textit{transpose}([[1,2,3],[10,20,30]]) = [[1,10],[2,20],[3,30]].$$

Finally, the *distribution* of an operation over a sequence is obtained by applying the operation to each member of the sequence (this is often called a *map).* For example,

$$\textit{distribute}(f,[1,2,3]) = [f(1), f(2), f(3)].$$

The *Order* of a SequenceL term is its level of nesting (scalars are of Order 0—denoted by *0* in the SequenceL function signatures; vectors are of Order 1—denoted by *1*; matrices are of Order 2—denoted by *2*; etc.). Any order can be accepted for a parameter given order *?*. Arguments of a SequenceL expression which are of higher order than indicated in the function signature are called *overtyped arguments,* and those whose order exceeds the expected order by a maximal amount (i.e., maximal among the parameters in the expression) are referred to as *maximally overtyped*. For example, both arguments of the expression [1, 2] + [[3, 4], [5, 6]] are overtyped, and the second argument is maximally overtyped. The key feature of SequenceL semantics, eliminating the need for control structures in many cases, is this: whenever any argument of an expression is of an order greater than that required by the function signature, the argument is normalized with respect to the collection of arguments that are maximally overtyped. The argument is then transposed, and the operation is distributed over the resulting sequence. This process continues recursively (through a succession of CSPs) until a base case is reached, in which the function or operator can be applied directly to its arguments.

It turns out that this simple semantics adjustment allows operators to locate and act on their intended arguments within a data structure, and synthesize the results into a new data structure, in a way that is both intuitive and flexible.

For example, in the base case, the infix + and * operators act in the usual way:

$$\begin{aligned} 3 + 10 &= 13 \\ 3 * 10 &= 30 \end{aligned}$$

Through repeated, implicit applications of NTD, we can multiply a vector by a scalar using the ordinary * operation:

$$\begin{aligned} &10 * [1, 2, 3] \\ &\text{(normalize)} \rightarrow [[10, 10, 10], [1, 2, 3]] \\ &\text{(transpose)} \rightarrow [[10, 1], [10, 2], [10, 3]] \\ &\text{(distribute)} \rightarrow [[10 * 1], [10 * 2], [10 * 3]] \\ &\rightarrow [10, 20, 30]. \end{aligned}$$

The same process can be applied to add vectors. The programmer simply writes, say, [1, 2, 3] + [10, 20, 30], which evaluates as follows:

$$\begin{aligned} &\text{(normalize)} \rightarrow [[1, 2, 3], [10, 20, 30]] \\ &\text{(transpose)} \rightarrow [[1, 10], [2, 20], [3, 30]] \\ &\text{(distribute)} \rightarrow [[1 + 10], [2 + 20], [3 + 30]] \\ &\rightarrow [11, 22, 33]. \end{aligned}$$

This works not because vector arithmetic is built into SequenceL, but because the usual operators scale up naturally via NTD.

NTDs also scale up to user-defined functions. One declares expected dimensions for function parameters in the function signature. For example, an identity function,

```
ident2 (matrix n) ::= n
```

is defined with a *two-dimensional* sequence for its argument. When provided a three-dimensional sequence to evaluate one NTD is performed:

Initial = ident2([[[1, 1, 1], [2, 2, 2], [3, 3, 3]], [[11, 11, 11], [12, 12, 12], [13, 13, 13]]])

CSP = [ident2([[1, 1, 1], [2, 2, 2], [3, 3, 3]]), ident2([[11, 11, 11], [12, 12, 12], [13, 13, 13]])]

Final = [[[1, 1, 1], [2, 2, 2], [3, 3, 3]], [[11, 11, 11], [12, 12, 12], [13, 13, 13]]]

Modifying the function to expect *one-dimensional* sequences `ident1(`**`vector`**` n) ::= n`, and providing the same three-dimensional argument, results in nested NTDs in two CSP steps. The first two steps and the final result are identical to the trace above. Only the non-italicized step below showing the nested NTD differs from the `ident2` trace.

Initial = *ident1([[[1, 1, 1], [2, 2, 2], [3, 3, 3]], [[11, 11, 11], [12, 12, 12], [13, 13, 13]]])*
CSP = [*ident1([[1, 1, 1], [2, 2, 2], [3, 3, 3]]), ident1([[11, 11, 11], [12, 12, 12], [13, 13, 13]])]*
CSP = [[ident1([1, 1, 1]), ident1([2, 2, 2]), ident1([3, 3, 3])], [ident1([11, 11, 11]), ident1([12, 12, 12]), ident1([13, 13, 13])]]
Final = *[[[1, 1, 1], [2, 2, 2], [3, 3, 3]], [[11, 11, 11], [12, 12, 12], [13, 13, 13]]]*

Modifying the function to expect *scalar* sequences `ident0(`**`scalar`**` n) ::= n`, and providing the same three-dimensional arguments, results in an additional nested NTD. Only the non-italicized step indicating the additional NTD varies from the `ident1` trace above.

Initial = *ident0([[[1, 1, 1], [2, 2, 2], [3, 3, 3]], [[11, 11, 11], [12, 12, 12], [13, 13, 13]]])*
CSP = [*ident0([[1, 1, 1], [2, 2, 2], [3, 3, 3]]), ident0([[11, 11, 11], [12, 12, 12], [13, 13, 13]])]*
CSP = [[*ident0([1, 1, 1]),ident0([2, 2, 2]),ident0([3, 3, 3])],* [*ident0([11, 11, 11]),ident0([12, 12, 12]),ident0([13, 13, 13])]]*
CSP = [[[ident0(1), ident0(1), ident0(1)], [ident0(2), ident0(2), ident0(2)], [ident0(3), ident0(3), ident0(3)]], [[ident0(11), ident0(11), ident0(11)], [ident0(12), ident0(12), ident0(12)], [ident0(13), ident0(13), ident0(13)]]]
Final = *[[[1, 1, 1], [2, 2, 2], [3, 3, 3]], [[11, 11, 11], [12, 12, 12], [13, 13, 13]]]*

Notice that all of the identity functions, *ident2, ident1,* and *ident0* gradually pull the nonscalars apart in a logical fashion and furthermore, put the nonscalars back together again. These are simple demonstrations of the power of the NTD combined with the CSP. Operations could have been performed at any level of dissection with an assurance that the nonscalar would reform in subsequent CSP steps. The NTD/CSP can be used to perform operations on dissected structures in an orderly manner.

For a further demonstration of the semantics hard at work (in lieu of the programmer), consider the evaluation of the variable instantiation code mentioned above. Recall the code:

```
instantiate(scalar var,val,char) ::=
      val when (char == var) else char.
```

In the case of a user-defined function, like *instantiate,* the user has indicated that the three arguments *var, val,* and *char* are scalars. Thus the function, as written, expresses only the base case in which a single character is instantiated. However, as in the above examples, this function will automatically "scale up" to handle arbitrarily deeply nested expressions. For example, suppose the function is called with arguments x, 3, and *[x,+,[[7,*,x],/,9]]:*

```
instantiate(x, 3, [x,+,[[7,*,x],/,9]]).
```

Since the third argument *char* expects a scalar, but has been given a list of length 3, the other two arguments are normalized to obtain: $[x, x, x]$, *[3,3,3], [x,+,[[7,*,x],/,9]]*. The results are then *transposed,* and the operation *distributed* among the resulting sequences, resulting in *3* function references, which may be evaluated in parallel (writing *ins* for *instantiate*):

$$[(\text{ins}(\text{x}, 3, \text{x}), \text{ins}(\text{x}, 3, +), \text{ins}(\text{x}, 3, [[7, *, \text{x}], /, 9])].$$

This step is hidden from the user—it is a result of the internals of the language translator. The first two subterms are now ready for evaluation. The third subterm, since its final argument is of greater order than expected by the function, undergoes two more rounds of NTD, obtaining

$$[[\text{ins}(\text{x}, 3, 7), \text{ins}(\text{x}, 3, *), \text{ins}(\text{x}, 3, \text{x})], \text{ins}(\text{x}, 3, /), \text{ins}(\text{x}, 3, 9)].$$

Note how the repeated use of normalize-transpose-distribute in successive CSP steps allows the function to descend implicitly through the data structure, "finding" its appropriate arguments without any additional effort from the programmer. Also note that the actual computations in one part of the evaluation may proceed in parallel with additional applications of NTD in other parts.

At this point, the arguments finally match what the function expects, and can therefore be evaluated based on the user's specification of the function body, leading to the final, desired result:

$$[3, +, [[7, *, 3], /, 9]].$$

The execution of the function descends recursively into the tree, and is essentially similar to the execution of the LISP, Haskell, or Prolog versions presented earlier. But in SequenceL the recursion falls automatically out of the semantics and is not explicit in the source code. As in this example, the internal workings of the normalize-transpose-distribute semantics can be fairly complex. However, the effect on code and its output is generally natural and intuitive, and often corresponds to simply omitting iterative and recursive constructs that would otherwise be needed.

The advantage SequenceL brings here is not a matter of smaller codes—keystrokes are cheap, all else being equal. But designing and debugging loops and recursive functions is expensive. To do so, the programmer must learn and master the appropriate constructs (Haskell has several: comprehension, map, zip, zipWith, , zipWith2, . . . , zipWith7, filter, etc.), and then apply them in the correct configuration with respect to order and nesting. *These configurations are often greatly constrained, or even determined, by the structure of the data along with the input and output types of the operators used*. Thus, the use of control structures is not as flexible as it first appears—they are often derived rather than chosen. For the human programmer, this derivation is taxing and error prone. SequenceL, by contrast, often yields the derivation automatically through NTD.

Not all instances of recursion can be eliminated—or need to be. As noted above, some algorithms are most naturally thought of and coded in this manner—and SequenceL allows arbitrary recursion when needed. However, the use of iteration or recursion merely to traverse and synthesize data structures, *which includes the majority of cases*, is a distracting and sometimes strenuous nuisance [Cooke and Gates 1991]. In many cases, SequenceL relieves this nuisance, allowing the programmer to write code closer to his or her mental picture of the problem solution. SequenceL's implicit normalization and transpose evaluation steps significantly diminish the need for iteration and recursion. In particular, recursion is typically not needed in SequenceL and is replaced by iterative operations implicit in the SequenceL code. These iterative operations *could be* performed in parallel (whether they are actually performed in parallel or not). By analogy with Haskell, this includes all recursive patterns covered by *zip, map*, and list comprehension, but not those covered by *foldr* and *foldl*. NTD also covers some cases, which are not covered by any of the standard Haskell constructs, as discussed in Section 7.

3. EXAMPLES RUN USING THE SEQUENCEL INTERPRETER

The goal of the SequenceL effort is to reduce the programmer's obligation to specify the procedural part of a problem solution. In this section, we focus on how SequenceL applies to different kinds of problems.

In Section 5, the semantics of SequenceL are given as a theory of first order logic. Building on the informal definitions presented in Section 2, one may view the complete "evaluation" of a SequenceL expression T_1 to be a series of Tableaux:

$$T_1 = T_2 = \cdots = T_n,$$

where $T_i = T_{i+1}$ is a theorem of Meta-SequenceL (see Section 5) for $1 \leq i < n$, and T_n is a term written using only scalars, commas, and square brackets.There is a SequenceL interpreter, which generates these theorems. This interpreter is the source of all the traces shown in this section and was used to evaluate all of the examples in this article. For purposes of this section, we will call each T_i a *Tableau,* as we did in Section 2, which overviewed the CSP.

3.1 Matrix Multiplication

NTD is first demonstrated on matrix computations. Here we recall the SequenceL *mmrow* and *dp* functions from Section 2. The *mmrow* function computes the matrix product of a vector and a matrix, while *dp* computes the dot product of two vectors.

```
mmrow(vector a, matrix b) ::= dp(a, transpose(b))
dp(vector x,y) ::= sum(x * y)
```

Given the matrix,

$$\mathtt{M1} = \begin{pmatrix} 1 & 2 & 4 \\ 10 & 20 & 40 \\ 11 & 12 & 14 \end{pmatrix}$$

An initial tableau for matrix multiply is:

$$\text{mmrow}\ (\text{M1}, \text{M1}) \tag{1}$$

We note that *mmrow's* first argument is expected to be of order *1*. Thus, we identify the three constituent rows making up *M1*.

```
R1:   <1    2    4>

R2:   <10  20   40>

R3:   <11  12   14>
```

Normalize then makes three copies of the second matrix:

$$\text{mmrow}([\text{R1}, \text{R2}, \text{R3}], [\text{M1}, \text{M1}, \text{M1}]) \tag{2}$$

and transpose and distribute yield the next tableau:

$$(\text{mmrow}(\text{R1}, \text{M1}), \quad \text{mmrow}(\text{R2}, \text{M1}), \quad \text{mmrow}(\text{R3}, \text{M1})). \tag{3}$$

Now the language interpreter instantiates the body of the *mmrow* function;

$$\begin{aligned} &[\text{dp}(\text{R1}, \textbf{transpose}(\text{M1})),\\ &\ \text{dp}(\text{R2}, \textbf{transpose}(\text{M1})),\\ &\ \text{dp}(\text{R3}, \textbf{transpose}(\text{M1}))]. \end{aligned} \tag{4}$$

Next true matrix transposes are performed forming M1^{T}, where

$$\texttt{M1}^{\texttt{T}} = \begin{pmatrix} 1 & 10 & 11 \\ 2 & 20 & 12 \\ 4 & 40 & 14 \end{pmatrix}$$

After the transposes, the *dp* functions are eligible for evaluation:

$$\begin{aligned}&[\mathrm{dp}(\mathrm{R1}, \mathrm{M1}^{\mathrm{T}}),\\ &\mathrm{dp}(\mathrm{R2}, \mathrm{M1}^{\mathrm{T}}),\\ &\mathrm{dp}(\mathrm{R3}, \mathrm{M1}^{\mathrm{T}}),].\end{aligned} \tag{5}$$

The *dp* function takes two order-1 sequences as input, but in (5) the second argument of each *dp* reference is a two-dimensional structure. Thus, we note the rows of the transposed *M1*:

$$\begin{aligned}\texttt{R1}' &= \langle 1 \quad 10 \quad 11\rangle\\ \texttt{R2}' &= \langle 2 \quad 20 \quad 12\rangle\\ \texttt{R3}' &= \langle 4 \quad 40 \quad 14\rangle\end{aligned}$$

Therefore, another *NTD* is performed on each *dp* resulting in 9 *dp* references:

$$\begin{array}{lll}[[\mathrm{dp}(\mathrm{R1}, \mathrm{R1}'), & \mathrm{dp}(\mathrm{R1}, \mathrm{R2}'), & \mathrm{dp}(\mathrm{R1}, \mathrm{R3}')],\\ [\mathrm{dp}(\mathrm{R2}, \mathrm{R1}'), & \mathrm{dp}(\mathrm{R2}, \mathrm{R2}'), & \mathrm{dp}(\mathrm{R2}, \mathrm{R3}')],\\ [\mathrm{dp}(\mathrm{R3}, \mathrm{R1}'), & \mathrm{dp}(\mathrm{R3}, \mathrm{R2}'), & \mathrm{dp}(\mathrm{R3}, \mathrm{R3}')]].\end{array} \tag{6}$$

At this point, the *dp* functions are instantiated and operator-level NTDs distribute operators to produce the final result:

$$[[65, 90, 140], [650, 900, 1400], [285, 430, 720]]. \tag{7}$$

The procedural aspects of Matrix Multiplication are fully discovered through the *NTD*. Furthermore, examination of the SequenceL trace reveals opportunities for parallel evaluations, which can lead to design decisions in developing concurrent codes. In Cooke and Rushton [2005], we show how these traces can be used to discover improved concurrent algorithms to be implemented in JAVA. Two observations come to mind in reviewing this trace:

(1) Parallel computations are a result of the evaluation automatically decomposing the operand sequences. This is the significant distinction between

SequenceL's evaluation strategy when compared with competitors like dataflow machines. In a dataflow machine, the programmer must decompose the data structures, resulting in more complicated functions—ones the programmer must write. See, for example, the pH matrix multiply immediately following this list of observations.

(2) The programmer did nothing to indicate where the parallel or iterative/recursive elements of the computation exist—these are found automatically via NTDs as the function and operations are evaluated according to the SequenceL semantics.

Now for the pH example promised in the first item of the list just read. pH (a dialect of parallel Haskell) functions to multiply two two-dimensional matrices are shown here as they appear in Nikhil and Arvind [2001].

```
row i x = let ((li, lj), (ui, uj))   =    bounds x
                 fill k      =       x!(i,k)
          in     mkArray (lj,uj) fill
col j x = let ((li, lj), (ui, uj))   =    bounds x
                 fill k  = x!(k,j)
          in     mkArray (lj,uj) fill

ip ar bc k1 k2 = let    s = 0.0
          in       for k <- (k1..k2)
                   do next s = s + ar!k * bc!k
                   finally s
matmul a b =  let   ((1,1),(m,n))      = bounds a
                    ((1,1),(_,l))   = bounds b
                    fill (i,j) = ip (row i a) (col j b) 1 n
              in    mk Array ((1,1),(m,l)) fill
```

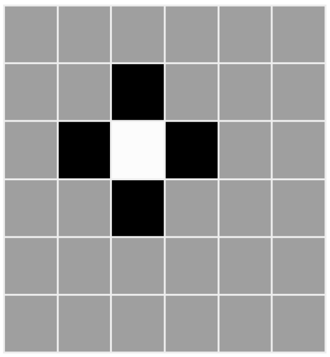
Even though there are no directives to indicate parallel processing opportunities, it is indeed incumbent upon the programmer to break the matrices apart. The parallelisms are only then discoverable.

3.2 Jacobi Iteration

For a more complex matrix computation, consider the Jacobi Iteration solution of a Partial Differential Equation according to the discretized formula:

$$\mu'_{j,k} = \frac{1}{4}(\mu_{j+1,k} + \mu_{j-1,k} + \mu_{j,k+1} + \mu_{j,k-1}) - \left(\rho_{j,k}\left(\frac{1}{4}\Delta^2\right)\right).$$

An important difference between Jacobi and Matrix Multiplication is that, in Jacobi, the computed value of a matrix element involves *only* its four neighbors: above, below, and to each side. Therefore, one must be able to select the appropriate values to compute the new value in an interior position. For example, to compute the *(3, 3)* element below (the lightened box), the darkened boxes must be used as indicated in the equation above.



To select the darkened elements requires a capability to call out row and column indices, much the way the *taking* clause did in the prior versions of SequenceL (see Figure 1). Here is one SequenceL solution to compute one iteration, defining the next matrix μ'.

```
jacobi(matrix a, scalar delta, matrix b)::=
    neighbors([1..length(a)],a, transpose(a)) - (b*(delta2))/ 4 ]
 neighbors(scalar i,vector a, matrix b)::=
    helper(a,b,i,[1..length(b)])

helper(vector a,b scalar i,j)::=
      a(j) when (i=1 or length(a)=i) or
                (j=1 or length(a)=j)
            else
     a(i+1) + a(i-1) + b(j+1) + b(j-1)/4
```

In the SequenceL solution, each of *μ's* rows is combined with each of its columns, with *neighbors* playing the role of the *mmrows* function of matrix multiply and *helper* playing the role of *dp*. Besides forming the desired Cartesian product of the row-order and column-order versions of the input matrix, the *NTD* also captures the row indices in *neighbors,* and the column indices in *helper.* Ultimately, there is a point in the resulting trace where each row/column combination is applied to *helper* with its respective subscripts. Let's consider the *neighbors* and *helper* functions. Assume *M1* and its transpose *M1*$^{\mathrm{T}}$ as previously defined for the matrix multiply example:

$$\boxed{M1 = \left\{ \begin{array}{l} R1 : \\ R2 : \\ R3 : \end{array} \begin{pmatrix} 1 & 2 & 4 \\ 10 & 20 & 40 \\ 11 & 12 & 14 \end{pmatrix} \right.} \quad \boxed{M1^{T} = \left\{ \begin{array}{l} R1' : \\ R2' : \\ R3' : \end{array} \begin{pmatrix} 1 & 10 & 11 \\ 2 & 20 & 12 \\ 4 & 40 & 14 \end{pmatrix} \right.}$$

and let R1, R2, and R3 be the rows of M1, and R1′, R2′, and R3′ be the rows of $\mathrm{M1}^{\mathrm{T}}$. When *neighbors* is referenced in *jacobi*, it includes arguments providing the subscripts of all the rows, via the generative *[1..length(M1)],* and all rows and all columns of the matrix a:

$$\text{neighbors}([1, 2, 3], \mathrm{M1}, \mathrm{M1}^{\mathrm{T}})$$

NTDs result in pairing each row and its respective subscript with all of the columns:

$$\begin{aligned}&[\text{neighbors}(1, \mathrm{R1}, \mathrm{M1}^{\mathrm{T}}),\\&\text{neighbors}(2, \mathrm{R2}, \mathrm{M1}^{\mathrm{T}}),\\&\text{neighbors}(3, \mathrm{R3}, \mathrm{M1}^{\mathrm{T}})].\end{aligned}$$

Likewise, the interaction between *neighbors* and *helper* combine each row and its respective subscript with each column and its respective subscript, resulting in:

$$\begin{aligned}&[[\text{helper}(\mathrm{R1}, \mathrm{R1}', 1, 1),\\&\text{helper}(\mathrm{R1}, \mathrm{R2}', 1, 2),\\&\text{helper}(\mathrm{R1}, \mathrm{R3}', 1, 3)],\\&[\text{helper}(\mathrm{R2}, \mathrm{R1}', 2, 1),\\&\textbf{helper}(\mathbf{R2},\mathbf{R2}', \mathbf{2}, \mathbf{2}),\\&\text{helper}(\mathrm{R2}, \mathrm{R3}', 2, 3)],\\&[\text{helper}(\mathrm{R3}, \mathrm{R1}', 3, 1),\\&\text{helper}(\mathrm{R3}, \mathrm{R2}', 3, 2),\\&\text{helper}(\mathrm{R3}, \mathrm{R3}', 3, 3)]].\end{aligned}$$

Only the second row, second column element fails to meet the condition of the *when* clause. This results in the average value of its neighbors above and below, and to the left and the right. The helpers produce each element of the matrix. As a result of a sequence of *NTDs* in the evaluation of the *jacobi* function, all elements of the matrix produced by *neighbors* are subtracted from corresponding elements of the *rho* matrix. The *rho* matrix is computed as a result of a series of NTDs in the subexpression of the *jacobi* function $\rho_{j,k}(1/4\Delta^2)$), culminating in the final result:

$$\begin{aligned}&[[0.999958, 1.99992, 3.99983],\\&[9.99958, 15.9992, 39.9983],\\&[10.9995, 11.9995, 13.9994]].\end{aligned}$$

The trouble with the forgoing solution to Jacobi iteration is that one must know about and skillfully deploy the NTD semantics to solve the problem. Such requisite knowledge is counterproductive to our purpose of shielding the user from technical details in a problem solution. Consequently, with virtually no change to the SequenceL syntax, we have introduced the concept of variable subscripts whose values are computed—rather than obtained as function arguments. The concept is not unlike the polynomial time backtracking one can set up for assertional databases in Prolog.

The prior definition of Jacobi in SequenceL produces the Cartesian product via the nested *NTDs,* which also produced the needed subscripts. In other words, nested NTDs can produce the desired combination of subscript pairs defined by a Cartesian product. The NTD semantics combine respective elements of operand sequences and side-steps the need to provide subscripts when their only purpose is to break apart and rebuild a nonscalar. There are times, however, when subscripts are a natural part of the problem solution. For example, recall the necessary use of subscripts in the Jacobi equation seen earlier and repeated here:

$$\mu'_{j,k} = \frac{1}{4}(\mu_{j+1,k} + \mu_{j-1,k} + \mu_{j,k+1} + \mu_{j,k-1}) - \left(\rho_{j,k}\left(\frac{1}{4}\Delta^2\right)\right).$$

Free variable subscripts in SequenceL range over the sizes of structures they subscript and operate in a manner similar to the evaluation of free variables in Prolog. When more than one free variable is specified, nested NTDs produce the Cartesian product of subscript values. With the use of free variables, the complete Jacobi solution in SequenceL is improved and closely matches the specifying equation:

```
jacobi_{j,k}(matrix a, scalar delta) ::=
     a(j,k) when (j=1 or length(a)=j) or (k=1 or length(a)=k)
            else
((a(j+1,k)+a(j-1,k)+a(j,k+1)+a(j,k-1))/4)-(a(j,k)*delta ^2)/4
```

Likewise, matrix multiply is improved:

```
matmul_{i,j} (matrix m1,m2) ::= sum(m1(i, all) * m2(all,j)).
```

There are times when subscripts are part of the basic concept of a problem solution as they are in the definition of Jacobi Iteration and Matrix Multiplication. The programmer can identify and provide these natural uses of subscripts, while leaving it to the NTD semantics to handle their processing. When subscripts are not a natural feature of a specification, but instead, are required in the iterative framework specifying *how* to solve a problem, the subscripts and their management is handled by the NTD semantics. An example of a subscript, absent in the SequenceL Matrix Multiplication, is the extra subscript (k in the example below) which is required by procedural definitions:

```
For i := 0 To rows do
   begin
      For j := 0 To cols do
      begin
         val := 0;
         For k := 0 To cols do
         begin
             val := val + (m1[i, k] * m2[k, j]);
         end;
         mm[i, j] := val;
      end;
   end;
```

3.3 Fast Fourier Transforms

The previous examples demonstrate how one declares intuitive solutions in SequenceL and the manner in which the NTD effectively discovers the procedural aspects of algorithms. The Discrete Fast Fourier Transform follows this trend as well. A Discrete FFT involves two computations defined by:

$$\begin{aligned} a(\omega^{j}) &= \mathit{fft(odd\ elements\ of\ a)} + \mathit{fft(even\ elements\ of\ a)} \bullet \omega^{2j} \\ a(\omega^{j+N}) &= \mathit{fft(odd\ elements\ of\ a)} - \mathit{fft(even\ elements\ of\ a)} \bullet \omega^{2j} \\ & \mathit{where}\ 0 \leq j \leq N-1. \end{aligned}$$

In the following SequenceL FFT definition, *c, /c, ^ce, and ^c are complex multiply, divide, e raised to a complex number, and a complex number raised to a real number, respectively.

```
fft(scalar pi2,e,n, matrix a) ::=
       fft(pi2,n/2,a(([1..n/2]*2-1))         +
       fft(pi2,n/2,a(([1..n/2]*2))           *c
       (e,0) ∧ce ((0,pi2) /c (n,0))^c (([0..n/2-1],0)
       ++
       fft(pi2,n/2,a(([1..n/2]*2-1))         -
       fft(pi2,n/2,a(([1..n/2]*2))           *c
       (e,0) ∧ce ((0,pi2) /c (n,0))^c (([0..n/2-1],0)
                    when length (a) > 1
                           else
                    a(1)
```

We will now comment on the subexpressions appearing in the solution and their relationship to the FFT definition. One term of the mathematical definition of FFT is $\omega = e^{(2\pi i) \div N}$. In SequenceL this is obtained by

$$\mathit{[2.71828,0]\ {}^\wedge ce\ ([0,pi2]\ /c\ [n,0])}, \qquad (A)$$

where *pi2* is instantiated with the argument *6.283185,* which is 2π, $\wedge$ *ce* is complex exponentiation, and /*c* is complex division.

When a is subscripted

$$\texttt{a([1..n/2] *2-1)}, \text{ where } n \text{ is the size of } a. \qquad (B)$$

NTDs on the multiplication and then the subtraction operator yields a vector of the odd-numbered elements of a.

The even vector is obtained in a similar SequenceL operation:

$$\texttt{a([1..n/2]*2)}, \text{ where } n \text{ is the size of } a. \qquad (C)$$

The Fast Fourier Transform is now definable recursively, where two sets of values are obtained by the equations (employing the expressions *A, B, and C,* above):

$$\begin{aligned} a(\omega^{j}) &= \mathit{fft(B)} + \mathit{fft(C)} \bullet A^{2j} \\ a(\omega^{j+N}) &= \mathit{fft(B)} - \mathit{fft(C)} \bullet A^{2j}\ \mathit{where}\ 0 \leq j \leq N - 1, \end{aligned}$$

which leads to the SequenceL definition given previously. Procedural aspects of these two equations are discovered and handled by the NTD.

3.4 Gaussian Elimination

The next problem we consider is Forward Processing in Gaussian Elimination. Our first SequenceL definition of Gaussian elimination is:

```
gauss(matrix e, scalar i) ::= e when length(e)=1 else
                e(1) ++ gauss(ztail(tail(e), e(1) i), i+1)

ztail(vector e2,e1, scalar i) ::= e2-(e1 * e2(i)) / e1(i)
```

An example of the matrix parameter *e* for the Gaussian Function follows:

eq1 = (1, -3, -2, 6)
eq2 = (2, -4, 2, 18)
eq3 = (-3, 8, 9, -9)

Parameter *i* selects an equation as the basis for processing. The function *zTail* performs the most significant computations. It returns the matrix obtained by subtracting an appropriate multiple of e1 from each row of e2, giving all 0's in the *i*th column. NTDs accomplish a considerable amount of the work. For example a typical call to ztail is made with respective parameters of a matrix M, vector v, and scalar s:

$$\text{ztail}(M, v, s).$$

An NTD is performed because the first argument is overtyped, resulting in

$$\begin{aligned}&[\text{ztail}(M(1), v, s),\\ &\text{ztail}(M(2), v, s),\\ &, \ldots,\\ &\text{ztail}(M(k), v, s)],\end{aligned}$$

where k is the number of rows of M. Instantiation of the body of ztail now gives

$$\begin{aligned}&[\mathbf{M}(\mathbf{1}) - (\mathbf{v} * M(1, s))/v(s),\\ &\mathbf{M}(\mathbf{2}) - (\mathbf{v} * M(2, s))/v(s),\\ &\ldots,\\ &\mathbf{M}(\mathbf{k}) - (\mathbf{v} * M(k, s))/v(s)].\end{aligned}$$

Since the operations -, *, and / act on scalars and the arguments shown in bold are vectors, another round of NTD occurs, resulting finally in the matrix

$$\begin{aligned}&[[M(1, 1) - (v(1) * M(1, s))/v(s), M(1, 2) - (v(2) * M(1, s))/v(s), \ldots]\\ &[M(2, 1) - (v(1) * M(2, s))/v(s), M(2, 2) - (v(2) * M(2, s)/v(s), \ldots]\\ &\cdots,\\ &[M(k, 1) - (v(1) * M(k, s))/v(s), M(k, 2) - (v(2) * M(k, s))/v(s), \ldots]].\end{aligned}$$

The complex pattern of subscripts to M and v appearing above reflects the thought pattern behind the design of the Pascal and NESL versions of the ztail function that follow.

```
function ztail(eq:matrix;r,c:integer):matrix;
/*r is the number of equations and c is the number of coefficients */
       var m:integer;
       begin
       i:=1;
       for j:=i+1 to r do
          begin
              m:=eq[j,i];
              for k:=1 to c do
                      eq[j,k]:=eq[j,k]-((eq[i,k]*m) / eq[i,i])
              end;
       return eq
       end;
```

Apart from the complexity of establishing the nested iterative control structures, care must be taken to manage the subscripts of the matrix containing the equations. Managing the interaction of the control structures and the subscripts provides a mental burden for the programmer, distracting him or her with technical details that tend to distract one from efforts to focus on the essence of the problem solution. Notice that the only subscript in the SequenceL *ztail* function identifies the row of coefficients being processed, which is a natural part of the intuitive picture of the problem solution. The additional subscripts in the Pascal-like solution above and the NESL solution below are what we claim to be the technical distractions arising from the more algorithmic detail involved in stating the *how* of the solution.

There is a slight improvement (over the Pascal version) in the NESL version, which uses list comprehension. Note however that the NESL solution still requires programmer effort to manage subscripts and the nested application of subscripts is not unlike the nested for-loops above:

```
ztail(eq,i)=
{ {eq[j][k]-eq[i][k]*eq[j][i])/eq[i][i]
     :k in [1:{#}eq[1]]
  }
  : j in [1:{#}eq]
};
```

The thought process behind the SequenceL codes deploying NTD downplays the numerical subscripts required by the Pascal and NESL solutions, and corresponds to a more visio-spatial picture of how the final matrix is formed. Recall the previous discussion in Section 3.2 concerning the natural and unnatural use of subscripts. In the case of the *ztail* function, the unnecessary subscripts are not even present in the SequenceL solution, not to mention the iterative or recursive scaffolding required to process them. In the case of the free variable version of the *jacobi* function, subscripts that are a natural part of the problem solution are visible, but once again the control structures required to manage them are not.

A common optimization in Gaussian Elimination is *pivoting,* in which the matrix of coefficients is rearranged to prevent the potential for division by zero. Using the built-in operator for set difference (i.e., the \ ), SequenceL pivoting can be accomplished by two helper functions:

```
pivot(matrix coef, scalar j) ::=
     [max(coef,j)] ++ \coef  max(coef,j)
            when length (coef)>1 else coef
max_I(matrix coef, scalar j)::= coef(I) when and (abs(coef(I,j)) >
     =abs(coef(all,j)))
```

There is no change to *ztail,* and a minor one to Gaussian to take care of the pivoting:

```
gauss(matrix e, scalar i) ::= e when length(e)=1 else
       e(1) ++ gauss(ztail(pivot( tail(e),i), e(1) i), i+1)
```

3.5 Quicksort

Recall that Quicksort is pivot-based. A pivot is selected from a list to be sorted; all items less than the pivot are placed in front of it; and all items greater than the pivot are placed after (++ denotes an append operation).

```
quick(vector a) ::=
     a when length(a) <= 1 else
     quick(less(a,a(length(a) div 2))++
     equal(a,a(length(a) div 2)) ++
     quick(great(a,a(length(a) div 2)))
```

The SequenceL functions for identifying elements less than, equal, and those that are greater than the pivot are quite intuitive and rely on NTDs to break apart data structures for the desired comparisons:

```
less(scalar a,b) ::= a when a < b
equal(scalar a,b) ::= a when a = b
great(scalar a,b) ::= a when a > b
```

For Example, Given the Initial Tableau, `Less`([5, 7, 2, 9], 7), NTDs Result in:

[Less (5, 7), Less (7, 7), Less , (2, 7), Less , (9, 7)]

Since Only the Tuples Containing Arguments *2* and *5* Satisfy the Comparison, They are the Only References to *Less* Returning Values. The Final Result is:

[5, 2]

Here is the NESL solution:

```
function Quicksort(A)=
if({#}A <= 1) then A
else
         let  b = S[rand({#}S)];
              S1 = {e in A|e<b}
              S2 = {e in A|e>b}
              R = {Quicksort(v); v in [S1,S2]};
         in R[0] ++ [e] ++ R[1];
```

NESL's comprehension construct ({`<var> in <sequence> | <condition>`}) distributes the elements of a referenced set among an operator. We claim that the NTD generalizes this concept, and is the only semantics required of SequenceL beyond the grounding of function parameters and the evaluation of built-in operators.

Note that the quicksort in SequenceL, though written in a functional syntax, truly is an *algorithm* in the sense that it is devised to execute quickly—as opposed to simply being declared to deliver a specified result. A more declarative sort might be written as follows:

```
Sort(vector List) := S where
     bag_equal(List,S) &
     S(1) <=···<= S(|S|)
```

In this code, *bag_equal* is a Boolean function that returns "true" if its arguments are lists which are equal as bags (i.e., multisets), and the ellipsis operator "··· ", acts with infix operators in an "intelligent" fashion. The *where* construct automatically results in a search for values of all uninstantiated variables (in this case, S), which make its right-hand-side true. We are currently implementing these extensions to the language, but the implementation is beyond the scope of this paper and should be considered future work as of this publication.

4. SCALABILITY: LARGE-SCALE APPLICATIONS OF SEQUENCEL

Traditionally, when new languages are introduced in the literature a wide variety of relatively small problem solutions are used to indicate the comparative strengths and weaknesses of the language abstraction. The problem solutions presented so far are also relatively small. In this section, we present evidence of the scalability of SequenceL.

Throughout the foregoing we have pointed out that the NTD accomplishes heavy lifting from a semantics point of view. We have recently developed a SequenceL interpreter in Haskell that centers all translation around the NTD. In fact, except for the declaration of the data structure used to implement the sequence and the definition of basic operations (e.g., arithmetic with scalars) all other language constructs are implemented in terms of the interaction between the NTD and the CSP. What this truly means is that all of the translation functions are treated like other user-defined SequenceL functions. Thus, the grounding of arguments and the evaluation of subscripts, free variables, conditionals, function bodies, and function references are governed by the CSP-NTD semantics. As a case in point, recall the SequenceL *instantiate* function and the manner in which the NTD was used to ground the variables of an arithmetic expression. The bulk of this NTD-centric version of SequenceL is shown in the Appendix. Consequently, once basic core capabilities (primarily the NTD-CSP) are available, building a translator is a simple matter.

4.1 NASA Applications of SequenceL

In the introduction, we observed that the human costs for developing codes are increasing to the point that high-level, language solutions are becoming competitive to lower level codes, especially given the decreasing cost of computational power. For many critical applications, the balance is beginning to tip in favor of declarative languages. At NASA, there is a growing need to solve problems more rapidly in the future. In many situations taking several days to fine tune a program that runs in 10 microseconds is unacceptable when an equivalent program in a high-level language takes 20 microseconds to run, but can be written and executed in fewer days. The overall delay recommends the higher-level language.

NASA's future exploration missions will not be as scripted as they have been in the past. On the long-distance and long-duration missions of the future, an ability to rapidly and dependably modify software capabilities is needed. Current approaches to software development and modification are unlikely to meet the NASA's future needs, [Cooke et al. 2006]. One possible step toward addressing NASA needs is to test declarative language approaches for prototyping requirements.

Recently, SequenceL was used to prototype the requirements of an onboard system for NASA's Space Shuttle [Cooke et al. 2005]. The system is called the Shuttle Abort Flight Manager (SAFM). An example SAFM requirement as specified by NASA Guidance, Navigation, and Control (GN&C) engineers is presented in Figure 2. Clearly, the requirement looks a lot like SequenceL. In the past, NASA developed prototypes apart from the requirements exemplified in Figure 2. With minor changes to the requirements—depicted in Figure 3—the requirement itself is a prototype since it is executable by the SequenceL interpreter.

The major change in the SequenceL version of the requirement is adding the *matrix multiply definition* and adding nested *[ ]'s* to denote the rows of the matrix. Since matrix computations (including matrix multiply) are the gist of the requirement, the NTD performs all the work in terms of the procedural aspects of the problem solution. In short, the NASA requirement, with only minor modifications, is executable in SequenceL.

We used SequenceL to develop the requirements for the Shuttle Abort Flight Management System. This is a large system—the requirements are 200 pages in length. Due to the success of the SAFM effort, we have recently developed a SequenceL prototype of the "flight rules checker" for NASA's Orion Crew Exploration Vehicle Onboard Abort Executive (CEV-OAE-FRC). This prototype

3.7.4.13.1 Functional Requirements

3.7.4.13.1.1 The signature of the Earth Fixed to Runway Transformation utility shall be as follows:

M_EFTo_Rw = EF_TO_RUNWAY(Lat, Lon, RW_Azimuth)

3.7.4.13.1.2 The Earth Fixed to Runway Transformation utility shall perform the following algorithm:

$$\mathbf{M} = \begin{bmatrix} \text{Cos(RW\_Azimuth),} & \text{Sin(RW\_Azimuth),} & 0 \\ \text{-Sin(RW\_Azimuth),} & \text{Cos(RW\_Azimuth),} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{MEFTopdet} = \begin{bmatrix} \text{-Sin(Lat) * Cos(Lon),} & \text{-Sin(Lat) * Sin(Lon),} & \text{Cos(Lat)} \\ \text{-Sin(Lon),} & \text{Cos(Lon),} & 0 \\ \text{-Cos(Lat) * Cos(Lon),} & \text{-Cos(Lat) * Sin(Lon),} & \text{-Sin(Lat)} \end{bmatrix}$$

M_EF_To_Rw = (**M**) • (**MEFTopdet**)

*Rationale: **M** is the Topodetic to RW matrix.*

Fig. 2. Example SAFM requirement.

```
M_EF_To_Runway(scalar Lat,Lon,Rw_Azimuth) ::= M_EF_ToRw(
[       [Cos(RW_Azimuth),           Sin(RW_Azimuth),            0],

        [-Sin(RW_Azimuth),          Cos(RW_Azimuth),            0],

        [0                          0                           1]

        ],

[       [-Sin(Lat) * Cos(Lon),  -Sin(Lat) * Sin(Lon),       Cos(Lat) ],

        [-Sin(Lon),             Cos(Lon),                   0        ],

         [-Cos(Lat) * Cos(Lon),   -Cos(Lat) * Sin(Lon),       -Sin(Lat)]
        ] )

M_EF_ToRw i,j(matrix M,MEFTopdet) ::=  sum(M(i,all) * MEFTopdet(all,j))
```

Fig. 3. Executable SAFM requirement in SequenceL.

software automates certain mission support decisions for the Orion vehicle, using a small set of "flight rules", identical to those written and used by human mission controllers. The SequenceL prototype was developed in parallel with a handcoded CEV-FRC written by NASA Guidance, Navigation, and Control Engineers in C. The SequenceL requirements are compiled into C++.

It is worth noting that the SequenceL version and the GN&C version were developed by different individuals, to some extent in competition with each other.

Furthermore, the NASA CEV Guidance, Navigation, and Control Functional Area Manager chose the portion of the CEV OAE that we would collaborate on because he viewed the FRC as the most challenging part of the OAE. Thus, we were not in a position to find a part of the system that would favor SequenceL. We completed the effort in June 2006.

The strengths of the SequenceL FRC include the ease of development and how well the SequenceL specifications match the engineers' intuitive understanding of the requirements. Notably, one error and one ambiguity in the specification were discovered in implementing the specification in SequenceL. These were *not* discovered in the process of implementing the specification in C. Both of these involved assigning values to variables that were later reassigned. The lack of assignment of variables in SequenceL was shown to be an important strength of the language allowing our effort to avoid errors not seen in the procedural approach. Subsequent work involving aerodynamic calculations has pointed out the strengths obtained by the NTD and free variables.

A weakness in the SequenceL approach in this context has to do with the way in which GN&C engineers have traditionally approached requirements. They typically view requirements from a procedural, event-driven point of view, whereas SequenceL provides a declarative, functional paradigm. We are now working on ways in which to ease the transition from their traditional approach to the approach we have taken as a result of the SequenceL language. NASA engineers see value in our approach since the SequenceL view pointed out issues and additional decisions that needed to be made as they developed the requirements and their prototype. We will soon begin developing a prototype of a larger portion of the same control system, which is to play a role in the acceptance testing of the flight software developed by CEV contractors. Our SequenceL codes are automatically compiled to C++ code, which runs in the same NASA simulation environment used to perform trade studies on GN&C systems.

Although runtime performance of the SequenceL-generated codes were slower than the GN&C hand-coded prototype, from an academic viewpoint, we were pleased with our results. Our generated codes had fewer decision points, but ran on average 1.5 to 2 times the time that the GN&C C prototype took. Figure 4 shows the average execution time for a set of 960 unit tests that performed a semi-black box test of FRC functionality.

The FRC has multiple output values. The NASA implementation computes all of the output values at once. SequenceL has a "compute on demand" characteristic. Since there is no assignment statement in SequenceL the burden of storing intermediate values to avoid redundant computation falls on the SequenceL compiler. Our current compiler is a prototype, which does not implement this optimization and furthermore, does not detect other common subexpressions or units of code. Some of the common units of code are many lines in length and therefore, are not detected by an optimizing C or C++ compiler. Future improvements to the compiler should substantially improve the performance of our codes and drastically reduce a lot of redundant code execution.

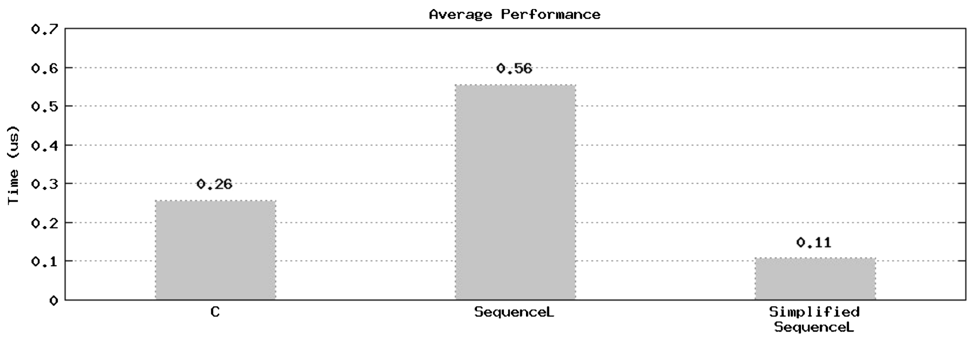


Fig. 4. Relative execution times.

To obtain a rough estimate of the impact of these planned optimizations we ran some tests that reduced the redundant code execution. We estimate that if the translator produced code that saved necessary intermediate values, the performance of our generated codes will be between the Average SequenceL (0.56) and the Average Simplified SequenceL (0.11), and be roughly equal to the hand written C.

4.2 Problems with SequenceL

SequenceL has a somewhat minimalist approach when it comes to the static checking of function arguments. This can and does lead to problems in the interpreted versions of SequenceL. For example, consider the following function:

```
m(vector x,y scalar z) ::= x * y / z.
```

Suppose the intended use of the function required a scalar value for z. SequenceL would still compute a result if z was provided a vector:

$$\begin{aligned} \textit{Initial} &= m([1, 2, 3], [2, 3, 4], [1, 2, 3]) \\ \textit{CSP} &= [m(1, 2, 1), m(2, 3, 2), m(3, 4, 3)] \\ \textit{CSP} &= [1{*}2/1, 2{*}3/2, 3{*}4/3] \\ \textit{CSP} &= [2/1, 6/2, 12/3] \\ \textit{CSP} &= [2, 3, 4] \\ \textit{Final} &= [2, 3, 4]. \end{aligned}$$

This unintended and possibly incorrect use of the function produces a result of the same form produced by a correct use of the function:

$$\begin{aligned} \textit{Initial} &= m([1, 2, 3], [2, 3, 4], 2) \\ \textit{CSP} &= [1, 2, 3]{*}[2, 3, 4]/2 \\ \textit{CSP} &= [1{*}2, 2{*}3, 3{*}4]/2 \\ \textit{CSP} &= [2, 6, 12]/2 \\ \textit{CSP} &= [2/2, 6/2, 12/2] \\ \textit{Final} &= [1, 3, 6]. \end{aligned}$$

Thus, an incorrect use of a function might go unnoticed by a user. In other languages the programmer could produce safeguards to prevent errors through stronger approaches to type checking. We are addressing this issue in our plans to produce an Intelligent Development Environment where programmers would execute their functions with "intended uses" that would then infer stronger type checking rules for SequenceL codes. In the code for function m above, either execution is a valid intended use that could serve as the basis for type checking.

5. SYNTAX AND SEMANTICS

In this section, we present the syntax and semantics of SequenceL—that is, a complete, rigorous description of the language and its meaning.

5.1 Syntax of SequenceL

Let U be a fixed set of user defined function symbols. The syntax of SequenceL over U is as follows (in practice, U is taken to be the set of symbols appearing as principal operators on the left-hand side of function definitions appearing in the program, together with the built-in operators):

constants:

```
Scalar ::= true | false | Numeral
Const ::= nil | Scalar
```

terms:

```
Prefix  ::= abs | sum | transpose| Term | length
Prefix2 ::= ~
Infix ::= +  | -| * | / | // | ^ | % | <  | > | <=  | >= | =
        |'|' | & | , | when | else | ++ |  .. | ,
Term  ::= Const | Prefix(Term)| Prefix2 Term
        | Term Infix Term | [Term] | U(Term) | Identifier
```

function signatures:

```
Simpleorder ::= s | ? | [Simpleorder]
Order ::= nil | Simpleorder | Simpleorder * Order
Signature(u) ::= u: Order -> Order, where u ∈ U
```

function definitions:

```
Arglist ::= ∈ | Identifier | Identifier Argtail
Argtail ::= ,Identifier | ,Identifier Argtail
Definition(u) ::= u(Arglist) ::= Term, where u ∈ U
Function ::= Signature(u) Definition(u), where u ∈ U
```

programs:

```
Program  ::= Function | Program Function
```

precedence classes (highest to lowest):

```
1. ~
2. ^
3. * / %
4. + -
5. > < <= >= = ≠
6. &
7. |
8. ++
9. when
10.   else
11.   ..
12.   ,
```

association,
Infix operators within a precedence class associate from left to right, except *else* and ',' which associate right to left.

grammar
The above rules for precedence and association give a unique parse for any program formed in accordance with the BNF rules. Function signatures like the ones shown in the grammar are derived from type information for function parameters by a simple pre-processing macro.

5.2 Syntax of Meta-SequenceL (MS)

The semantics of SequenceL are given as a formal theory of first order logic, which we will call MS (for meta-SequenceL). The language of MS contains the logical symbols $\forall \exists \neg \rightarrow \vee \wedge$ ( ) with their usual syntax, along with countably infinitely many variables *a, b, c, d* $a_1, a_2, a_3, \ldots$. MS contains the following binary infix predicate symbols

$$\begin{array}{cc} \in & \leq \\ = & \geqslant \\ < & > \end{array}$$

MS contains the following infix function symbols

$$\begin{array}{cc} + & / \\ - & \% \\ * & \wedge \end{array}$$

and the following prefix function symbols and constants (numbers following slashes denote arities)

$-/1$	(*unary minus*)
$\mathbf{R}/0$	(The real numbers)
$\mathbf{Seq}/0$	(The set of all sequences)
$floor/1$	(greatest integer function)
$true/0$	
$false/0$	
$undef/0$	
$max/1$	(maximum element of a list of numbers)
$domain/1$	(domain of a mapping)
$g/5$	(helper function for list processing, see below)
$h/3$	(helper function for list processing, see below)
$range/1$	(range of a mapping)
$openlist/0$	(atomic constant for list processing)
$tmap/3$	(helper for normalize-transpose, see below)
$s/2$	(denotation of a term with respect to a program)
$s/1$	(denotation of a term using only built-in operators)
$\emptyset/0$	(the empty set)
$order/1$	(level of nesting of a data structure)
$dist/2$	(distribute, aka map, an operator to a list of values)
$trans/1$	(transpose a list of lists)
$norm/3$	(normalization, see below)
$numeral/1$	(maps numbers to their numeral representation not terminating in 9's)

In addition, any SequenceL term enclosed in chevrons ($\langle\langle . \rangle\rangle$), with 0 or more subterms replaced by MS variables, is a term of MS, and any base-10 numeral not terminating in 9's is a constant symbol of MS. For example, 0.5 is a constant symbol but the numeral 0.499999..., which has the same value as 0.5, is not. This is to give real numbers unique representations.

5.3 Signatures of Built-In Functions

The signatures discussed in this section are considered to be implicitly part of every program. The signatures of the built-in functions are as follows:

Every built-in prefix operator p has the signature

$$p : \mathtt{s}-> \mathtt{s}$$

++ is the infix append operator; it operates on sequences and returns a sequence. "," operates on arbitrary terms and returns a sequence. "=" operates on arbitrary arguments and returns a scalar. Hence:

```
++   : [?]*[?] -> [?]
,    : ?*? -> [?]
=    : ?*?->s.
when : ?*s -> ?
```

Every other infix operator *i* operates on a pair of scalars and returns a scalar, and so has the signature

$$i : \mathrm{s}^*\mathrm{s}- > \mathrm{s}.$$

Every sequence *c* is considered an operator that maps *i* to its *i*th member, and so carries the signature

$$c : \mathrm{s}- >?.$$

5.4 Interpretation of the Predicate and Function Symbols of MS

The symbols of MS are interpreted in a semantic domain U, which is taken to be the smallest set containing the real numbers, the atoms *true, false, undef,* and *openlist,* and closed under the operations of Zermelo–Frankle set theory iterated through finite ordinals. Tuples are realized as sets in the usual way, as in Cohen [1966]. Functions are allowed to be partial, though some are made total by extending them to map to the atom *undef* where they are intuitively undefined.

The symbols $\in$ and $=$ are given their usual interpretations on sets and atoms, and the other infix predicates and functions are given their usual interpretations in the real numbers.

The prefix function symbols are interpreted as follows:

R denotes the real numbers

$-$ denotes unary negation

floor(x) denotes the greatest integer not exceeding the real number x, or denotes *undef* if x is not a real number.

true, false, undef, and *openlist* denote their corresponding atoms.

max(x) denotes the maximum element of the set x of numbers if it exists, and *undef* otherwise

domain(f) and *range(f)* denote the domain and range, respectively of the mapping f

$\emptyset$ denotes the empty set

order(x) $= 0$ if x is an atom or number, or n if x is a mapping from a finite set of integers to a set of items whose maximum order is $n - 1$.

numeral(x) is the base-10 decimal representation of the real number x not terminating in 9's. If x is not a real number, *numeral(x)* is *undef*.

The function *g(u,v,n,m,i)* gives the ith element of the concatenation of two lists u and v, of length n and m respectively. Formally,

$$\begin{aligned} g(u, v, n, m, i) \;=\; & \lambda i.u(i), 1 \leq i \leq n \\ & v(i-n), n < i \leq n+m \end{aligned}$$

The function h is used as a helper function to translate indices of lists:

$h(m, n, i) = i - m + 1, m < i < n$
$\qquad\qquad$ *undef*, otherwise

$denotation(\ll s \gg) = \{0\}$

$denotation(\ll ? \gg) = \{0, 1, 2 \cdots\}$

$denotation(\ll [x] \gg) = \{t + 1 : t \in denotation(x)\}$, where x is any simple type.

$denotation(\ll a_1 * \cdots * a_n \gg)$ is the vector $< t_1, \ldots, \; t_n >$, where t_i is the denotation $of \;\; a_i$.

$denotation(\ll \texttt{nil} \gg) = \emptyset$

tmap(i, f, P) is the set of argument types admissible in the ith argument of function f in program P:

$$tmap(i, \, f, P) = \{\texttt{k} : \ll f \; : x_1^* \cdots ^* x_n - > \; y \gg \in P \wedge k \in denotation(x_i)\}.$$

$extend(t, k)$ returns k copies of t in a sequence. That is, $extend(t, k)$ denotes the function mapping i to t for $i = 1\,..\,k$.
Suppose t maps i to a_i for i $= 1\,..\,n$, k is a positive integer, and S is a subset of $\{1, .., \;\text{n}\}$. Then $norm(t, k, S)$ maps i to a'_i for i=$1\,..\,n$, where

$$\begin{aligned} a'_i = a_i, & \qquad i \in S, \\ extend(a_i, k) & \qquad \text{otherwise} \end{aligned}$$

$trans(t) = \lambda\text{i. } \lambda\text{j. } t(\text{j})(\text{i})$ where $order(t) > 1$,

$dist(p, T) = \lambda i.p(t(i))$

If $t(x_1, \ldots x_n)$ is a SequenceL term with 0 or more subterms replaced by the MS variables $\mathrm{x}_1, \ldots \mathrm{x}_\mathrm{n}$, then $\ll t(x_1, \ldots x_\mathrm{n}) \gg$ denotes the function which maps the n-tuple $(t_1\,..\,t_\mathrm{n})$ of SequenceL terms to the SequenceL term obtained by replacing x_i with (t_i) respectively.

The symbol s is the denotation function from SequenceL terms to their denotations in the semantics domain. In cases involving only built-in operators, the semantics are independent of any program and s is a function of a single parameter consisting of a SequenceL expression. In general, the denotation of an expression depends on a program containing definitions of user-defined functions, and so s is binary. Intuitively, if e is a SequenceL expression and P a SequenceL program, then $s(P, \ll \text{e} \gg)$ denotes the value of the expression e with respect to the program P—that is, the evaluation of e obtained using built-in operators along with function definitions appearing in P. This interpretation is described formally by the axioms in Section 5.6.

5.5 Axioms of MS for Symbols other than s

If $c_1, .. c_{n+1}$ are constant terms of MS, f is an n-ary prefix function symbol of MS other than s, and $f(c_1, .. c_n) = c_{n+1}$ is true in the interpretation of Section 5.4, then $f(c_1, .., c_n) = c_{n+1}$ is an axiom of MS. If c_1, c_2, c_3 are constant terms of MS and Op is an infix function symbol of MS where $c_1 Op\ c_2 = c_3$ is true in the interpretation then $c_1 Op\ c_2 = c_3$ is an axiom of MS. If c_1, c_2 are constant terms of MS and Op is an infix predicate symbol of MS where $c_1\ Op\ c_2$ is true in the in the interpretation, then $c_1\ Op\ c_2$ is an axiom of MS. This covers trivial axioms like $1+2=3$, $7<8$, etc. The usual symmetry, transitivity, and substitution axioms for equality are also axioms of MS.

5.6 Axioms for s

This section is the heart of the matter—we give the axioms for the interpretation function s from SequenceL terms to the semantic domain. The axioms for built-in operators, Axioms 1–14, are written using the unary semantic function $s/1$. These are extended to cover semantics with respect to a given program by Axiom 15. Finally, the axiom for user-defined programs appears as Axiom 16.

1. arithmetic operators

$(\forall a \forall b)(s\ll a\gg \in \mathbf{R} \wedge s\ll b\gg \in R \rightarrow s\ll a+b\gg = s\ll a\gg + s\ll b\gg)$

Similarly for $-$, $*$, $/$, $\%$, `floor`

2. equality

$(\forall a \forall b)\,(s\ll a\gg = s\ll b\gg \rightarrow s\ll a = b\gg = \mathit{true})$

$(\forall a \forall b)\,(s\ll a\gg \neq s\ll b\gg \rightarrow s(s\ll a = b\gg) = \mathit{false})$

3. arithmetic comparison

$(\forall a \forall b)(s\ll a\gg \in \mathbf{R} \wedge s\ll b\gg \in \mathbf{R} \wedge s\ll a\gg < s\ll b\gg \rightarrow s\ll a < b\gg = \mathit{true})$

$(\forall a \forall b)(s\ll a\gg \in \mathbf{R} \wedge s\ll b\gg \in \mathbf{R} \wedge \neg(s\ll a\gg < s\ll b\gg) \rightarrow s\ll a < b\gg = \mathit{false})$

Similarly for >, <=, >=, <>

4. boolean operations

$(\forall a \forall b)(s\ll a\gg = \mathit{true} \wedge s\ll b\gg = \mathit{true} \rightarrow s\ll a\ \texttt{and}\ b\gg = \mathit{true})$

$(\forall a \forall b)(s\ll a\gg = \mathit{false} \vee s\ll b\gg = \mathit{false} \rightarrow s\ll a\ \texttt{and}\ b\gg = \mathit{false})$

Similarly for `or` and `not`

5. when

$(\forall a \forall b)(s\ll b\gg = \mathit{true} \rightarrow s\ll a\ \texttt{when}\ b\gg = s\ll a\gg)$

$(\forall a \forall b)(s\ll b\gg = \mathit{false} \rightarrow s\ll a\ \texttt{when}\ b\gg = \mathit{undef})$

6. else

$(\forall a \forall b)(s\ll a\gg = \mathit{undef} \rightarrow s\ll a\ \texttt{else}\ b\gg = s\ll b\gg)$

$(\forall a \forall b)(s\ll a\gg \neq \mathit{undef} \rightarrow s\ll a\ \texttt{else}\ b\gg = s\ll a\gg)$

7. append

$$(\forall a \forall b)(s\ll a\gg \in \mathrm{Seq} \wedge s\ll b\gg \in \mathrm{Seq} \wedge max(domain(s\ll a\gg)) = n \wedge max(domain(s\ll b\gg)) = m \rightarrow$$
$$s\ll a ++ b\gg = \lambda i.g(s\ll a\gg, s\ll b\gg, \mathrm{n}, \mathrm{m}, \mathrm{i}))$$

where, recall,

$$g(u, v, n, m, i) = \lambda i\, u(i),\ 1 \leq i \leq \mathrm{n}$$
$$v(i - \mathrm{n}),\ \mathrm{n} < i \leq \mathrm{n} + \mathrm{m}$$

$$(\forall a \forall b)(s\ll a\gg \notin \mathrm{Seq} \vee s\ll b\gg \notin \mathrm{Seq} \rightarrow s\ll a ++ b\gg = undef)$$

8. transpose

$$(\forall a)(s\ll a\gg \in \mathrm{Seq} \wedge (\forall \mathrm{x} \forall \mathrm{y})p(a, x, y) \rightarrow s\ll \mathtt{transpose}(a)\gg = \lambda \mathrm{i}.\,\lambda \mathrm{j}.\, s\ll a\gg(\mathrm{j})(\mathrm{i}))$$

where $p(a, x, y)$ denote the formula:

$$x \in \mathrm{range}(\mathrm{s}\ll a\gg) \wedge y \in \mathrm{range}(\mathrm{s}\ll a\gg) \rightarrow x \in \mathrm{Seq} \wedge y \in \mathrm{Seq} \wedge$$
$$\mathrm{max}(\mathrm{domain}(\mathrm{x})) = \mathrm{max}(\mathrm{domain}(\mathrm{y}))$$

9. '..'

$$(\forall a)(\forall b)(s\ll a\gg \in \mathrm{Num} \wedge s\ll b\gg \in \mathrm{Num} \rightarrow s\ll a..b\gg = \lambda i.\, h(s\ll a\gg, s\ll b\gg, i)$$

Where, recall,

$$\mathrm{h}(\mathrm{m}, \mathrm{n}, \mathrm{i}) = \mathrm{i} - \mathrm{m} + 1,\ \mathrm{m} < \mathrm{i} < \mathrm{n}$$
$$undef,\ \text{otherwise}$$

10. sequences

$$\mathrm{s}\ll \mathtt{nil}\gg = \emptyset$$
$$(\forall a)(s\ll [a\gg = (\mathrm{openlist}, s\ll a\gg))$$
$$(\forall a)(\forall b)(s\ll a, b\gg = (\mathrm{openlist}, s\ll a ++ [b]\gg))$$
$$(\forall a)(\forall x)(s\ll a\gg = (\mathrm{openlist}, x) \rightarrow s\ll a]\gg = x)$$

11. function calls

$$\ll f(x_1, .., x_\mathrm{n}) := T\gg \in P \rightarrow$$
$$(\forall a_1) \cdots (\forall a_\mathrm{n})((\forall i)(i \leq \mathrm{n} \rightarrow \mathrm{order}(c_i) \in tmap_i(f, P)) \rightarrow$$
$$\mathrm{s}_\mathrm{P}\ll f(a_1, .., a_\mathrm{n})\gg = \mathrm{s}_\mathrm{P}\ll T[x_1 \backslash (a_1), .., x_\mathrm{n} \backslash (a_\mathrm{n})]\gg)$$

12. sequence subscripting

$$(\forall a)(\forall b)(\mathrm{s}\ll a\gg \in \mathrm{Seq} \wedge \mathrm{s}\ll b\gg \in \mathrm{domain}(\mathrm{s}\ll a\gg) \rightarrow \mathrm{s}\ll a(b)\gg = \mathrm{s}\ll a\gg(\mathrm{s}\ll b\gg))$$
$$(\forall a)(\forall b)(\mathrm{s}\ll a\gg \notin \mathrm{Seq} \vee \neg \mathrm{s}\ll b\gg \in \mathrm{domain}(\mathrm{s}\ll a\gg) \rightarrow \mathrm{s}\ll a(b)\gg = undef))$$

13. For each program P and function f of arity n defined in P, we have the normalize-transpose-distribute axiom:

$$
\begin{array}{ll}
(\forall a_1)\cdots(\forall a_{\mathrm{n}})(\forall S)(\forall k)(\forall m)(& \\
\quad S \subseteq \{1, \ldots, n\} & \wedge \\
\quad (\forall i)(i \in S \rightarrow ex(\mathrm{s}\ll a_i\gg, \mathrm{i}, \mathrm{f}, \mathrm{P}) = m) & \wedge \\
\quad (\forall i)(0 < \mathrm{i} \leq \mathrm{n} \wedge \mathrm{i} \notin S \rightarrow \mathrm{ex}(\mathrm{s}\ll a_{\mathrm{i}}\gg, i, f, P) < m) & \wedge \\
\quad (\forall i)(\forall j)(i \in S \rightarrow max(domain(s\ll a_{\mathrm{i}}\gg) = k & \\
\rightarrow & \\
\quad s_{\mathrm{P}}\ll f(a_1, .., a_{\mathrm{n}})\gg = s_{\mathrm{P}}(dist(\ll f\gg, trans(norm((a_1, .., a_n), k, S)))) &
\end{array}
$$

where $ex(x ,i ,f ,P)$ is an abbreviation for $order(x)$ - $max(tmap(i, f, P))$.

Note $ex(x)$ is allowed to range over extended integers, that is, integers including positive and negative infinity. In particular, $max(tmap(i, f, P))$ will be infinite in case the signature of f in P has a '?' in the ith argument.
The antecedent of the main implication essentially says that S is the set of indices for which the expression $(\ll f(a_1, .., a_{\mathrm{n}})\gg$ is maximally overtyped, and that the maximally overtyped sequences are all of the same length k. The consequent says that we perform normalize-transpose-distribute, as explained informally in Section 2.

14. Constants
$\mathrm{s}(\ll\mathrm{true}\gg) = true$
$\mathrm{s}(\ll\mathrm{false}\gg) = false$

$(\forall a)(a \in \boldsymbol{R} \rightarrow s\ll a\gg = numeral(a))$

15. For every SequenceL program P and every SequenceL expression e containing only built-in operators, the following is an axiom.

$$\mathrm{s}(\ll e\gg) = \mathrm{s}(P, \ll e\gg)$$

16. User-defined functions
Suppose P is a SequenceL program containing a function definition

$$f(x_1, \ldots, x_n) = exp(x_1, \ldots, x_n)$$

where f is an identifier, $x_1, \ldots, x_n$ are SequenceL variables, and $exp(x_1, \ldots, x_n)$ is a SequenceL expression containing no variables except possibly $x_1, \ldots, x_n$. Then

$$s(P, \ll f(x_1, \ldots, x_n)\gg) = \mathrm{s}(P, \ll exp(x_1, \ldots, x_n)\gg)$$

is an axiom of MS.

5.7 Specification of Interpreters

The following definitions give the specification with respect to which SequenceL interpreters are verified:

Definition (*Soundness of an Interpreter*). A SequenceL interpreter I is said to be *sound* if whenever I returns value v for the term t with respect to program P, $s(P, \ll t \gg) = v$ is a theorem of MS.

Definition (*Completeness of an Interpreter*). A SequenceL interpreter I is said to be *complete* if whenever $s(P, \ll t \gg) = v$ is a theorem of MS, I returns the value $v$for the term t with respect to program P.

Since MS is a theory of standard predicate calculus, evaluation of most constructs is lazy in principal. However, keeping with the philosophy described in Section 1, SequenceL makes no commitment to how interpreters and compilers actually perform computations, provided they are sound. Similarly, we make no general commitments to the internal representation of sequences, for example, as lists, arrays, dynamic arrays, etc. Different compilers may work differently; the same compiler may work differently for different sequences appearing in the same program, or even the same function definition.

We have implemented an interpreter, which we believe is sound and complete. Verification of the interpretation is a direction for further work.

5.8 SequenceL—Is Turing Complete

We show the Turing Completeness of SequenceL through an implementation of the Universal Register Machine (URM). A URM consists of an ordered pair (P, R) where P is a *program* (defined below) and R is an ordered multiset of positive integers known as the *registers* of the machine. The URM language is known to be Turing Complete.

A URM program can be any one of the following strings enclosed in chevrons:

—$\ll$a$n\gg$, where n is an integer. This program increments the nth register.
—$\ll$s$n\gg$, where n is an integer. This program decrements the nth register.
—$\ll x;\, y\gg$, where y is a URM program and x is a URM program not of the form $x_1; x_2$. This program executes $x$and then y.
—$\ll(x)n\gg$, where x is a URM program and n is a positive integer. This program executes x while the nth register is nonzero.
—$\ll$halt$\gg$. This program halts the machine.

We will represent URMs in SequenceL by strings generated by the following grammar:

$$\begin{aligned} \text{URM} &::= ((\text{M}), \text{Regs}) \\ \text{M}_0 &::= (\text{‘a’}, \text{integer})| \end{aligned}$$

$$
\begin{aligned}
& (\text{`s'}, \text{integer})| \\
& (\text{`lp'}, \text{integer}, (\text{M}))\,| \\
& \text{`halt'} \\
\text{M} &::= \text{M}_0|\text{M}_0, \text{M} \\
\text{Regs} &::= ((\text{integer}, \text{integer})\,\text{R1}) \\
\text{R1} &::= , (\text{integer}, \text{integer})\,\text{R1}\,|\,\in
\end{aligned}
$$

Note that the memory of the machine is represented by a sequence of ordered pairs (n, v), where n is the register number and v is the value stored in that register.

In reviewing the URM operations, we note that a and s are the only operations that directly modify the register store, and that they both reference the register being incremented or decremented. Our organization of the registers is a sequence of sequences, where the nested sequences provide a register number followed by the current value contained in that register.

In SequenceL, the a and s operations are carried out in a single function *indecr,* which operates on a single register. When presented with a sequence of registers, the NTD isolates the individual registers to which individual *indecr's* are to apply. Furthermore, the NTD reconstructs the sequence of registers once the various *indecr* activations complete.

```
indecr(scalar i, vector r, scalar op) ::=
        [i, r(2) + 1] when i = r(1) & op = ‘a’
                else
        [i, r(2) - 1] when i = r(1) & op = ‘s’
                else r
```

Axioms 1, 5, 6, and 13 of Section 5.6 can be used to show that the input *(i, r, ‘a’),* where r is a sequence of register-value pairs, will return a version of r with the ith register incremented and no other change. Similarly, the input *(i, r, ‘s’)* will return a version of r with the ith register decremented and no other change.

The *urm* function handles the control operations: sequencing, looping, and halting. It has two arguments—a sequence of instructions of unknown (?) nesting comprising a machine and a sequence of registers.

```
urm(? m, matrix r) ::=
        [urm( m([2,..., length(m)]),indecr(m(1,2),r,m(1,1))]     (1)
              when m(1,1) = ‘a’ or m(1,1) = ‘s’ else
        [urm( m(1,3)++m ,r) when r(m(1,2),2) > 0]                (2)
              when m(1,1) = ‘lp’ else
        [urm( m([2,...,length(m)]), r) when r(m(1,2),2) = 0]     (3)
              when m(1,1) = ‘lp’ else
        r     when m(1) = ‘halt’.                                (4)
```

We will now sketch the proof that the above function implements the semantic operations of a URM. To do this, it must be shown that each of the bulleted URM operations is faithfully carried out when its SequenceL counterpart is supplied as an argument to the *urm* function. This will show that SequenceL is Turing complete.

Let $m = (p, r)$ be a SequenceL representation of a URM program. From the grammar rule for M, it follows that p is a sequence whose first member m_1 is an M_0. If m is a singleton, it must then be of one of the following four forms: *(a, integer)* , *(s, integer)*, *(lp, integer, (M))* or '*halt*'. The first two cases are handled by clause (1) and *indecr*, but the machine fails to reach a halt state. Consider the third case, say, $m =$ ('lp', n, (P)) for some integer n and URM program P. If the nth register of r is nonzero (i.e., if r contains $(n,\ k)$ where $k > 0$), Case (2) of the *urm* function will fire by Axioms 5 and 6 of the SequenceL semantics, executing P on r and then calling the machine again, by Axioms 3, 4, 7, 12, and 13. If r contains $(n,\ 0)$, or if r does not contain $(n,\ k)$ for any k, Case (3) is satisfied and the machine fails to halt. In the final case (4) where $p =$ 'halt', the machine halts and returns r as desired, by Axioms 2, 5, 6, and 12.

In case P is not a singleton, again its first member m_1 must be of one of the four forms *(a, integer)*, *(s, integer)*, *(lp, integer, (M))* or *halt*. The last case is just as above. In the first two cases Axioms 12 and 13, together with the analysis of the *indecr* function, guarantee that the result of the machine is the result of executing its first instruction in p, followed by executing the tail of p,as desired. In the third case, where m_1 is =('lp', n, (B)) for some integer n and URM program B, either r contains $(n,\ k)$ where $k > 0$, r contains $(n,\ 0)$, or r does not contain $(n,\ k)$ for any k. If r does not contain $(n,\ k)$ for any k then the machine terminates without reaching a halt state by Axioms 2, 3, 5, and 6. If r contains $(n,\ 0)$, then the tail of the machine is executed upon the existing registers by axioms 5, 6, 12, and 13. Finally, if r contains $(n,\ k)$ where $k > 0$, Axioms 3, 5, 6 and 12 imply that the machine executes the body of the loop B on r and calls P itself again. This completes the proof.

6. AUTOMATIC PARALLELISMS

High-performance computing can often achieve gains in performance by exploiting the power of multiple processors running in parallel. However, these gains come at a price in terms of coding. Parallel algorithms typically require the programmer to learn and use *yet another* set of programming constructs for directing parallel execution, on top of those required for specifying sequential iteration, which lie again on top of those required to execute basic operations. If this sounds hard, it is: it has been estimated that the cost of developing parallel programs averages \$800 per line of code [Pancake 1999]. We have seen how SequenceL can diminish the programmer's burden by reducing the need for sequential control structures [Cooke 1996, 1998]. Our recent research suggests that the language can be even more effective in reducing the burden of orchestrating parallel execution [Cooke and Andersen 2000; Cooke and Rushton 2005]. As control structures are derived through SequenceL's normalize-transpose-distribute process, control and data parallelisms can be automatically detected and implemented without explicit direction by the programmer.

In Cooke and Andersen [2000], different classes of parallel problems were presented. Three classes were considered and the implicit parallelisms inherent in the problem solutions were discovered by SequenceL. These three classes are represented by:

(1) Matrix multiplication (in which parallel paths are independent),
(2) Gaussian Elimination (where there are dependencies among paths), and
(3) Quicksort (where the parallel paths cannot be predicted apriori, but unfold dynamically).

The parallelisms discovered are now based on the NTD and the nested NTDs when implementing the free variables. These simplifications have significantly improved our ability to generate sequential codes for the NASA applications. We are now exploring our ability to generate C/MPI codes. This remains current and future work.

7. RELATED WORK

Work with goals similar to ours, include efforts on NESL [Blelloch 1996], and Lämmel and Peyton-Jones's papers on "boilerplate elimination" [Lämmel and Peyton-Jones 2003, 2004].

NESL's comprehension operation is similar to NTD, except that it is triggered by an explicit syntax, whereas NTD is triggered by overtyped arguments. For example, addition of vectors u and v would be accomplished in NESL by

```
{x + y : x in u; y in v}
```

and in SequenceL by

```
u + v.
```

At this level the NESL and SequenceL syntax are comparably readable, given a small amount of practice in each. However, the NESL comprehension syntax becomes cluttered if we must traverse deeper, nested data structures. Replacing vectors u and v with matrices a and b, in NESL we write

```
{{x + y : x in u; y in v} : u in a; v in b}
```

compared with SequenceL's

```
a + b.
```

The SequenceL is still readable at a glance. We claim the NESL is not. We do not claim the NESL code is *hard* to read; a competent NESL programmer can grasp it, with only a miniscule probability of error, by looking at the code for just a few seconds. But this is typically true of any single line of code in any language. Now make it one of ten thousand lines, and give the programmer the distraction of having to understand the abstract algorithm he is implementing on top of the code syntax, and these miniscule probabilities and few seconds are liable to add up to real errors and real delays. This is why we claim *readability at a glance* is important.

If the depth of the data structures is not known at compile time, as with the instantiation algorithm of Section 2, NESL must deploy recursion, and the code becomes comparable with the solution in LISP or Prolog. This is a direct consequence of the *explicitness* of NESL's syntax for distributing an operation over a data structure: if the depth of the structure is unknown, the number of nested levels of comprehension syntax is unknown as well.

Haskell's comprehension structure has the same feature of explicitness, as well as a static typing system, which can make generalized maps (such as the instantiation algorithm) clumsy to write. Lämmel and Peyton-Jones [2003] attack this problem by supplying syntax and semantics for an `everywhere` construct, which simplifies the implementation of generalized maps. Instantiation can be implemented in their framework as follows:

```
instantiate1:: var -> val -> token -> token
instantiate1 x v t | x=v = v otherwise =t

instantiate x v exp = everywhere
      (mkT (instantiate1 exp))
```

This is a marked improvement over the plain Haskell solution given in Section 2. The comparison with SequenceL on this problem is analogous to NESL vs. SequenceL on vector addition: SequenceL uses one small function instead of two. The extra function in the Haskell version is necessary to deploy the `everywhere(mkT (...))` control construct. This construct marks the deployment of a generalized mapping, which is implicit under SequenceL's NTD.

In a separate paper, Lämmel and Peyton-Jones [2004] approach the problem of generalized zips—that is, performing operations on respective leaf nodes of trees with identical structure. For an example of a generalized zip operation, we might add respective leaf nodes of the trees *[1, [2, [3, 4]], [5]]* and *[10, [20, [30, 40]], [50]]*, to obtain *[11, [22, [33, 44]], [55]]*. Using their `gzipWithQ` operator, whose semantics are defined in Lämmel and Peyton-Jones [2004], this operation may be written as

```
gzipWithQ   +    [1,[2,[3,4]],[5]]    [10,[20,[30,40]],[50]].
```

Giving semantics for `gzipWithQ` is an interesting result, because Lämmel and Peyton-Jones [2004, p. 1] point out that generalized zips "at first appear to be somewhat tricky in our framework."

In SequenceL, generalized zips come free of charge, just like generalized maps. They are obtained automatically from the same semantics—the NTD. We write:

```
[1,[2,[3,4]],[5]] + [10, [20, [30,40]], [50]]
```

and the recursive "zipping" operation occurs automatically, including the traversal of the input data structures, and assembly of the output structure.

The semantics of Haskell's `everywhere` and `gzipWithQ` operators are difficult largely because of Haskell's static type system. However, static typing brings certain advantages in performance and error checking. Current implementations of SequenceL use dynamic typing, but it is an open question whether NTD can be used with static typing to achieve performance gains (while possibly sacrificing some flexibility and convenience in coding). We consider this a question for future research.

In Cooke [1996, 1998] and Cooke and Andersen [2000], we compare SequenceL to other languages including FP [Backus 1978] and APL [Iverson 1962]. Included here is a further comparison to APL, since the earlier papers pre-date the NTD semantic.

7.1 APL and SequenceL

In this section, we show how many of the advanced operations in APL are covered by the SequenceL NTD. First we show APL code to find prime numbers up to some limit R:

$$\mathrm{PRIMES} : (\sim \mathrm{R} \in \mathrm{R}^{\circ}. \times \mathrm{R})/\mathrm{R} \leftarrow 1 \downarrow \iota\mathrm{R}.$$

Clearly, this definition requires very little in terms of keystrokes, but does require some documentation to decipher. APL is right associative. The ιR generates the numbers from 1 to the limit R. If R is 6, then the list is [*1, 2, 3, 4, 5, 6].* The down-arrow on the list strips off the 1 and the left arrow assigns the resulting vector to R. From there R$^{\circ}$.× R generates the *outer product* of the vector, which presents a matrix of the values obtained by multiplying the vector times itself:

×	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36

Next, using set membership an APL *selection vector* is constructed. Each element of the selection vector indicates whether a corresponding number in R is in the table. The vector produced in our example in which R is 6 is [0, 0, 1, 0, 1]. The selection vector is negated and then, using the /-operator, it is used to select the corresponding elements from the original vector: [1, 1, 0, 1, 0] / [2, 3, 4, 5, 6] yields [2, 3, 5].

The equivalent functions in SequenceL are offered below. First, the outer product is produced by:

```
table_{I,J} (vector N) ::= N(I) * N(J).
```

The free variables are obtained via nested NTDs. The NTDs occur because the SequenceL function to ground free variables is defined on scalar values for the indices and is provided with vectors of values from 1 to the length of N. The remainder of the selection results from the comparison of each element of N with the elements of the table:

```
primes2_I (scalar N) ::=
        [2,...,N](I) when and(and([2,...,N](I) =\=
                                   table([2,...,N])).
```

This comparison (=\= for not equals) also involves two rounds of NTDs. A pair of *and-reduces* indicate whether the item is in the table.

A more efficient and intuitive definition of *primes* in SequenceL is:

```
prime(scalar N) ::= N when and (0 ≠ N mod [2,..., sqrt(N)]).
```

To obtain the prime numbers up to some limit, one would reference *primes* with *prime([2,..., Limit])* and again NTDs would be performed at the function reference level.

In general, the equivalent of APLs selection vector can be easily accomplished in SequenceL, because NTDs are also performed on function bodies. A *when*-clause expects single Boolean values in its conditional part. If multiple values exist, NTDs are performed. Consider the definition of even numbers $\{x | x \in N \;\&\; x \textbf{ mod } 2 = 0\}$. In SequenceL one obtains the evens with the function:

```
evens(vector N) ::= N when N mod 2 = 0.
```

The NTD does the heavy lifting as usual:

$$
\begin{aligned}
\mathit{INITIAL} &= \mathit{evens([1, 2, 3, 4, 5, 6])}\\
\mathit{CSP} &= [1, 2, 3, 4, 5, 6] \textbf{ when } [1, 2, 3, 4, 5, 6] \textbf{ mod } 2 = 0\\
\mathit{CSP} &= [1, 2, 3, 4, 5, 6] \textbf{ when } [1 \textbf{ mod } 2, 2 \textbf{ mod } 2, 3 \textbf{ mod } 2, 4 \textbf{ mod } 2,\\
&\qquad 5 \textbf{ mod } 2, 6 \textbf{ mod } 2] = 0\\
\mathit{CSP} &= [1, 2, 3, 4, 5, 6] \textbf{ when } [1, 0, 1, 0, 1, 0] = 0\\
\mathit{CSP} &= [1, 2, 3, 4, 5, 6] \textbf{ when } [1 = 0, 0 = 0, 1 = 0, 0 = 0, 1 = 0, 0 = 0]\\
\mathit{CSP} &= [1, 2, 3, 4, 5, 6] \textbf{ when } [\mathit{false, true, false, true, false, true}]\\
\mathit{CSP} &= [1 \textbf{ when } \mathit{false}, 2 \textbf{ when } \mathit{true}, 3 \textbf{ when } \mathit{false}, 4 \textbf{ when }\\
&\qquad \mathit{true}, 5 \textbf{ when } \mathit{false}, 6 \textbf{ when } \mathit{true}]\\
\mathit{CSP} &= [\mathit{empty}, 2, \mathit{empty}, 4, \mathit{empty}, 6]\\
\mathit{CSP} &= [2, 4, 6]\\
\mathit{FINAL} &= [2, 4, 6].
\end{aligned}
$$

Other operators from APL, including the transpose and rotates are easily defined in SequenceL, and once again nested NTDs do the bulk of the work:

```
transpose_{I,J} (matrix N) ::= N(J,I)
rotate_right_{I,J} (matrix N) ::= reverse(N)(J,I).
```

8. SUMMARY AND CONCLUSIONS

The fundamental questions that have driven the research resulting in SequenceL include the following:

—Can we eliminate more of the procedural aspects of a problem solution?

—Can we develop a simple, but precise language that declares an intended problem solution and have the language's semantics "discover" the missing procedural aspects of the solution?

—How simple can we make this semantic?

Very quickly, we settled on the goal of eliminating the iterative control structures that process nonscalar data structures. We did not completely eliminate the need for recursion nor should we have. The key feature of the language is its Consume-Simplify-Produce and the Normalize-Transpose-Distribute semantics.

The NTD results in the discovery of the procedural aspects of many concurrently or iteratively solved problems, particularly those involving the decomposition of nonscalar data. Recursion comes free of cost, in that it requires no formal or operational semantic definitions. In particular, the Consume-Simplify-Produce semantics permit a function to "leave work" in the subsequent tableau. Since the remaining work could be a reference to the function itself, all that is required for the next execution of the function is to ground the variables assuming no NTDs are needed first. Return addresses or traditional activation records are not kept or managed in any of the SequenceL interpreters. Another point worth making is that assignment of values to variables, either through input-output or through traditional assignment statements, is simply not supported in SequenceL.

The beginning point of the SequenceL effort is discussed [Cooke and Gates 1991], which introduced the fact that iterative algorithms involve producing scalars from nonscalars, scalars from scalars, nonscalars from scalars, and nonscalars from nonscalars. Language constructs in SequenceL were formed based upon these classes of problem solutions. A better but similar way to classify these algorithms was also presented, first in Meijer et al. [1991]. The idea presented there involves catamorphisms (similar to *nonscalars to ?*), anamorphisms (similar to *? to nonscalars*), and two additional morphisms one of which involves compositions of the cata- and anamorphisms. These compositions can be achieved using the SequenceL constructs.

The early work concerning the classes of iterative problem solutions, led to a 15 year effort leading to SequenceL [Cooke 1996, 1998]. In addition to automatically deriving many iterative and recursive algorithms to traverse data structures, we have also investigated the use of SequenceL to automatically discover and evaluate parallelizable subtasks [Cooke and Andersen 2000]. In many ways, the independent investigation resulting in SequenceL has uncovered many of the same opportunities for parallelisms as they are discovered in other functional language approaches [Sipelstein and Blelloch 1991; Blelloch 1996; Banatre and Le Metayer 1993; Loidl and Trinder 2006; Nikhil and Arvind 2001]. SequenceL, however, benefits further from the fact that the CSP-NTD semantics discovers many inherent parallelisms when functions and operators are applied to nonscalar data structures. Our recent results and experiments lead us to believe that the SequenceL approach should allow scientists and engineers to express problem solutions that have greater appeal to their intuition.

Recent efforts to apply SequenceL to NASA GN&C systems, led to the introduction of free variables, and have also led us to construct SequenceL interpreters strictly in terms of the CSP-NTD. These recent results not only show the scalability of SequenceL and the CSP-NTD, but are also providing clear paths to code generation. In the near-term our goal is to automatically compile readable and well performing C code from SequenceL definitions. From these iterative codes we are also planning to investigate the generation of C-MPI codes to run in conventional parallel computing environments.

APPENDIX

```
module Eval where
import Sl_elem

-- NTD Functions

max_length (Seq s) = foldr max 0 lengths where lengths = map s_length s
max_length _ = -1

expand len (Seq s) = (Seq (expand' len s s))
                        where expand' l (h:t) r | l <= length(r) = r
                                                | otherwise = expand' l (t++[h]) (r++[h])
expand (-1) x = x
expand len x = expand len (Seq [x])

normalize (Seq s) = Seq (map (expand len) s) where len = max_length (Seq s)

transpose (Seq s)
        | ((s_depth (Seq s)) == 1) = (Seq s)
        | transposeable = Seq (transpose' s)
        | otherwise = (Seq s)
                where   transposeable = (foldr min (s_length (head s)) (map s_length s)) ==
                                (max_length (Seq s))
                        transpose' ( (Seq h):t)
                                | (h == []) = []
                                | otherwise = (Seq (map s_hd s2)): transpose' (map s_tl s2)
                                        where s2 = (Seq h):t
nt st num s
        | num == 0 = s
        | otherwise = Seq (map (nt st (num - 1)) normtran)
                --where  (Seq normtran) = transpose( normalize s)
                where   (Seq normtran) = transpose( normalize added_depth)
                        added_depth =  add_depth s st

add_depth (Seq ((Function f):args)) st = Seq ((Function f):(add_depth' args correct_depths))
        where   correct_depths = which_match f args st
                add_depth' _ [] = []
                add_depth' (arg:t1) (correct:t2)
                        | correct = (Seq[arg]):(add_depth' t1 t2)
                        | otherwise = arg:(add_depth' t1 t2)

remove_epsilons (Seq s)
        | removed == [] = Epsilon
        | otherwise = (Seq removed)
                where   removed = filter (not.isEpsilon) s
                        isEpsilon (Epsilon) = True
                        isEpsilon _ = False
remove_epsilons s = s
```

```
clean_up s = remove_epsilons s

num_functions (Seq s) = length [a| a<-s, (is_function a)]
        where   is_function (Function d) = True
                is_function _ = False

contains_vars ((Var h):t) = True
contains_vars ((Function "ground"):_) = False
contains_vars ((Seq h):t) = (contains_vars h) || (contains_vars t)
contains_vars [] = False
contains_vars (h:t) = contains_vars t

contains_indefs ((Indef h):t) = True
contains_indefs [] = False
contains_indefs (h:t) = contains_indefs t

ground_indefs (Seq s) = ground_indefs' s 0
        where   ground_indefs' s1 num= Seq[ Function "|", Seq[Function "ground",
                Seq[Function "values", Var ("indef_temp"++(show num)), (Seq indefs)],
                                        new_seq]]

                                where   (Indef indefs) = head ([i| i<-s1, (is_indef i)])
                                        is_indef (Indef x) = True
                                        is_indef _ = False
                                        replace_first x y (h:t)
                                                | x==h = (y:t)
                                                | otherwise = (h:(replace_first x y t))
                                        replaced = replace_first (Indef indefs)
                                                (Var ("indef_temp"++(show num))) s1
                                        new_seq
                                                |(contains_indefs replaced) =
                                                        ground_indefs' replaced
                                                       (num+1)
                                                | otherwise = (Seq replaced)
ev st (Seq ((Function f):t))
--      | (f /= "values") && (f /= "ground") && (contains_vars t) = (Seq ((Function f)
        :evaled_args))
       | (f /= "|") && (f /= "is_a") && (contains_indefs evaled_args) = ev st (ground_indefs (Seq
                ((Function f):evaled_args)))
        | (matching_args f evaled_args st) = call_func f evaled_args st
        | otherwise = clean_up(Seq (map (ev st) normtran))
                where   evaled_args
                                        | f == "when" = (t!!0):(ev st (t!!1)):(drop 2 t)
                                        | f == "ground" = (ev st (t!!0)):(t!!1):[]
                                        | otherwise = map (ev st) t
                        (Seq normtran) = nt st 1 (Seq ((Function f):evaled_args))
ev st (Seq (h:(Function f):t)) = ev st (Seq ((Function f):(h:t)))
ev st (Seq s) = clean_up(Seq (map (ev st) s))
ev st (Arg v) = get_value st v
ev st s = s

-- Symbol Table Implementation

type SymbolTable = [ST_entry]
type ST_entry = (Name, Args, ArgNames, Func)
type Name = [Char]
type Args = [SL_elem]
type ArgNames = [[Char]]
type Func = [SL_elem]->[SL_elem]

call_func nm args st = ev (add_symbols st arg_syms) (clean_up (f args))
        where   (f,arg_names) = get_func nm args st
                arg_syms = make_arg_syms args arg_names
                make_arg_syms _ [] = []
                make_arg_syms (a:as) (n:ns) = (n, [], [], (\([])->a)):(make_arg_syms as ns)

get_value st nm = call_func nm [] st

   get_func nm args st = last [(f,arg_names) | (n,a,arg_names,f)<-st, n==nm, (args_match args a)]

   args_match args header = matches args header
        where   matches [] [] = True
                matches [] yt = False
                matches xt [] = False
                matches (x:xt) ((I (-1)):yt) = matches xt yt
                matches ((Seq x):xt) ((Seq y):yt) = (matches x y) && (matches xt yt)
                matches (x:xt) ((I y):yt)
                        | ((s_depth x) == y) = matches xt yt
                        | otherwise = False
                matches _ _ = False

   which_match f args st = which_match' header args
        where   header = head [a | (n,a,_,_)<-st, n==f, ((length a) == (length args))]
                which_match' [] [] = []
                which_match' ((Seq h1):t1) ((Seq h2):t2) = (foldr (&&) True
                        (which_match' h1 h2)):(which_match' t1 t2)
                which_match' ((Seq h1):t1) (h2:t2) = False:(which_match' t1 t2)
                which_match' ((I h1):t1) (h2:t2) = (h1 == s_depth(h2) ||
                        h1==(-1)):(which_match' t1 t2)


   matching_args nm args st = or (map (args_match args) headers)
        where   headers = [a | (n,a,_,_)<-st, n==nm]


   add_function name args func st = (name,args,0,func):st
   add_symbols new_syms st = new_syms ++ st
```

ACKNOWLEDGMENTS

We are extremely grateful to the reviewers and editorial staff of this journal. Their efforts made the final paper much better than it would have been had we been left to our own device. Many thanks for the contributions of former students, particularly to the amazing Changming Ma. Comments and guidance from friends and colleagues Ken Ford, Michael Gelfond, and William Marcy contributed much to our efforts. Most of all we appreciate the support and efforts of our NASA collaborators: Howard Hu, Ryan Proud, and Jeremy Hart.

REFERENCES

BACKUS, J. 1978. Can programming be liberated from the von neumann style? *Comm. ACM 21*. 8 (Aug.), 613–641.

BANATRE, J. AND LE METAYER, D. 1993. Programming by multiset transformation. *Comm. ACM 36*. 1, (Jan.), 98–111.

BISHOP, J. 1990. The effect of data abstraction on loop programming techniques. *IEEE Trans. Soft. Eng. 16*, 4 (Apr.), 389–402.

BLELLOCH, G. 1996. Programming parallel algorithms. *Comm. ACM 39*, 3 (Mar.) 98–111.

COHEN, P. 1966. *Set Theory and the Continuum Hypothesis.* WH Benjamin, New York.

COOKE, D. 1996. An introduction to SEQUENCEL: A language to experiment with nonscalar constructs. *Softw. Pract. Exper. 26*, 11 (Nov.), 1205–1246.

COOKE, D. 1998. SequenceL provides a different way to view programming. *Comp. Lang. 24*, 1–32.

COOKE, D. AND ANDERSEN, P. 2000. Automatic parallel control structures in SequenceL. *Softw. Prac. Expe. 30*, 14 (Nov.), 1541–1570.

COOKE, D., BARRY, M., LOWRY, M., AND GREEN, C. 2006. NASA's exploration agenda and capability engineering. *Computer 39*, 1 (Jan.), 63–73.

COOKE, D. AND GATES, A. 1991. On the development of a method to synthesize programs from requirement specifications. *Int. J. Softw. Eng. Knowl. Eng. 1*, 1 (Mar.), 21–38.

COOKE, D., GELFOND, M., RUSHTON, N., AND HU, H. 2005. Application of model-based technology systems for autonomous systems. In *American Institute of Aeronautics and Astronautics Infotech@Aerospace Conference.* AIAA Press, Reston, VA. 8 pages.

COOKE, D. AND RUSHTON, N. 2005. Iterative and parallel algorithm design from high level language traces. Lecture Notes in Computer Science, vol. 3516. Springer-Verlag, New York. 891–894.

GELFOND, M. AND LIFSCHITZ, V. 1998. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*. Seattle, WA, Aug. MIT Press, Cambridge, MA, 1070–1080.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comp. 9*, 365–385.

IVERSON, K. 1962. *A Programming Language*. Wiley, New York.

LÄMMEL, R. AND PEYTON-JONES, S. 2003. Scrap your boilerplate: A practical design patterns for generic programming. *ACM SIGPLAN Notices 38*, 3, 26–37.

LÄMMEL, R. AND PEYTON-JONES, S. 2004. Scrap more boilerplate: Reflection, zips, and generalised cates. *ACM SIGPLAN Notices 39*, 9, 244–255.

LIN, F. AND ZHAO, Y. 2004. Assat: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence 157*, 1–2 (Aug.), 115–137.

LOIDL, H. AND TRINDER, P. 2006. A Gentle introduction to GPH. http://www.macs.hw.ac.uk/~dsg/gph/docs/Gentle-GPH/gph-gentle-intro.html.

MEIJER, E. AND FOKKINGA, M., AND PATERSON, R. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. Lecture Notes in Computer Science, vol. 523. Springer-Verlag, New York. 124–144.

MILLS, H. AND LINGER, R. 1986. Data structured programming: Programming without arrays and pointers. *IEEE Trans. Soft. Eng. 12*, 2 (Feb.), 192–197.

NIKHIL, R. AND ARVIND. 2001. *Implicit Parallel Programming in pH.* Morgan Kaufmann, San Francisco.

PANCAKE, C. 1999. Those who live by the flop may die by the flop. Keynote Address, 41st International Cray User Group Conference, Minneapolis, MN.

SIPELSTEIN, J. AND BLELLOCH, G. E. 1991. Collection-oriented languages. *Proc. IEEE, 79*, 4.

Received July 2005; revised November 2006; accepted June 2007