# SequenceL – An Overview of a Simple Language

Daniel E. Cooke and J. Nelson Rushton
Department of Computer Science
Texas Tech University
Lubbock, TX 79409

Abstract - SequenceL is a concise, high-level language with a simple semantic that provides for the automatic derivation of many iterative and parallel control structures. The semantic repeatedly applies a "Normalize-Transpose" operation to functions and operators until base cases are discovered, which include the grounding of variables and the application of built-in operators to operands of appropriate types. This paper introduces the language from an intuitive point of view, indicating the scalability of the normalize-transpose, the different forms of recursion supported by the language and its abstraction, and new shorthand methods for specifying additional iterative/recursive problem solutions.

Keywords: Declarative Language, Language Design

#### 1.0 Introduction

During the past fifteen years work on a purely functional language, called SequenceL has been under way (CG91, C96, C98, CA00), culminating in major simplifications discovered during the past eighteen months. SequenceL is a small language with a simple semantic, which discovers and evaluates most iterative parallel algorithms implied by a very high level specification. The beginning point of this effort can be traced to (CG91), 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 very similar way to classify these algorithms was also presented, first in (MFP91). 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.

This paper focuses on the recent results, which significantly simplify the syntax and semantics of this language and introduces the Normalize-Transpose (NT) Semantic. The complete SequenceL grammar can be found in the Appendix at the end of this paper. Intuitively, the semantics involve the pairing of function arguments and parameters (i.e., grounding function variables) and performing built-in operations (e.g., arithmetic operations). The terms of SequenceL are evaluated in a data flow environment. When terms are ready to execute, they are removed from a "tableau," which is an abstraction of a shared memory, similar to the structures found in GAMMA (BL93) and NESL (B96). For example, consider the following distinct steps in language's evaluation of (4+5)\*(2+3):

```
Initial tableau = ((4+5) * (2+3))

Next tableau = (9 * 5)

Final tableau = (45)
```

The workhorse semantic of SequenceL, and what we claim to be one of the major results of this research is the NT semantic operator. This semantic achieves a goal similar to that of the Lämmel and Peyton-Jones, boilerplate elimination. (LPJ03, LPJ04) In our effort and in theirs, the goal is to focus on problems where the "meat" of a computation is performed on a nonscalar data structure, but the bulk of the program to perform the computation is a "boilerplate" traversal algorithm. The Lämmel and Peyton-Jones effort focuses on the type of a data structure and the automatic derivation of an appropriate traversal algorithm. Ours is based on the expressed type of an operator. When data, not matching this expected

type, appears in the data-flow-based tableau, the NT semantic will break it apart to apply the specified computation.

This paper will concentrate on the language abstraction and the application of NT's to a problem solution beginning at the operator level and proceeding to the functional level. All examples have executed in an interpreter written in December, 2003. The convention we follow here is to present all items a SequenceL programmer inputs in courier text. *Italicized* text signifies all of the translation steps taken, internally, by the SequenceL interpreter.

## 2.0 The Data Structure and Normalize/Transpose.

The only data structure available in SequenceL is the sequence. The sequence is capable of representing any data structure. A sequence differs from a set in two ways: a sequence can have multiple occurrences of the same item (e.g., sequence  $\{1,2,3\} \neq \{1,2,2,3\}$ ) and the ordinal position of an item matters (e.g., sequence  $\{a,f,a,d\} \neq \{f,a,d,a\}$ ). The low level implementation of sequences is allowed to be compiler-dependent

The normalize-transpose (NT) semantic becomes more intuitive if we first consider its analog in natural language. For example, suppose you hear the following sentences: *Sam, Bill, and Dan moved the table*, and *Sam, Bill, and Dan passed the bar exam*.

The natural conclusion here is that the individuals moved the table as a collective, but each of them passed his own exam. Note the syntactic structures are identical, and the difference in meaning – the "distribution" of the action over members of a collection – is based on the semantic consideration that the predicate *passed the bar exam* is normally applied to individuals and not groups.

This is the basic intuition behind the NT and when to apply it. Most built-in operators in SequenceL are defined to operate on scalars. When such an operation is applied to something other than two scalars, the Normalize-Transpose (NT) rule is applied. Consider the following initial tableau:

$$(10,20,30) * 1.1$$

In this example, the first operand is a sequence, considered to be of order 1, and the second operand is scalar, considered to be of order 0. The \* operator is defined in SequenceL to work only on scalar operands in both places. In this case, the expression is first *normalized* which constructs a sequence of the second operand and a sequence of the operator, where the two new sequences are the same size as the nonscalar operand. (10,20,30) (\*,\*,\*)\* (1.1,1.1,1.1)

Next a transpose is performed, which forms a new expression using the *ith* element from each of the normalized sequences, where, in this case, *i* ranges from *I* to 3, resulting in: (10 \*1.1, 20 \*1.1, 30\*1.1)

Since the operators now have two atoms, the three expressions can be evaluated either iteratively or in parallel: (11, 22, 33)

The semantic scales well. For nested structures, nested iterative or parallel evaluations are a byproduct of successive applications of *NT*:

$$((10,20,30), (200,300)) * 1.1 = NT$$
 (1)

$$((10,20,30)*1.1, (200,300)*1.1) = ^{NT}$$
 (2)

$$((10*1.1, 20*1.1, 30*1.1), (200*1.1,300*1.1)) = (3)$$

$$((11,22,33),(220, 330))$$

In step (2) one begins to notice that the application of the NT rule itself can also be evaluated iteratively or in parallel. In step (3) there are 5 multiplication operations that can be evaluated iteratively or in parallel. Steps (2) and (3) indicate how the application of the NT's and the evaluation of arithmetic expressions are interwoven. Notice that, even though the application of the NT can be viewed as a translation step, it effectively does the work that an application program written in other languages must do anyway. The difference with SequenceL is that the programmer is not burdened with designing and writing the nested iterative or parallel algorithms – they are discovered and executed at runtime. The algorithms that would be required in other languages even for the simple example above involves nested loops or nested parallel program structures, both being costly and error-prone types of code to write.

Steps 1-3 and the final result shown above indicate how the tableau changes to reflect the evolution of the originally specified computation. The tableau serves as an abstraction of a global, shared memory. Like the PRAM of NESL (B96) and the multiset of GAMMA (BL93), a single or multiple processors can reference and modify elements of the tableau. In SequenceL, each of these idealized processors capable of accessing the tableau are also capable of performing NT's, grounding variables, and otherwise evaluating SequenceL built-in operations.

As previously mentioned, NT applies to any SequenceL construct. Consider the following relational expression, where the operator again expects to operate on two scalars:

```
(10, 50, 30, 40) > 23 = ^{NT}

(10 > 23,50 > 23,30 > 23,40 > 23) = ^{NT}

(false, true, true, true)
```

This effect can be leveraged to emulate logical quantification with only conservative additions to the language. For example, to test a natural number n for primality, we can test all k strictly between 1 and n to see if  $n \mod k = 0$ . Along with SequenceL's generative construct a..b, which returns a list of integers between a and b inclusive, NT provides an elegant expression of this quantified test:

```
nor(n mod [2..n-1] = 0)
```

A function in SequenceL requires a signature that indicates the order, or expected level of nesting, of its operand(s) and a simple function body. For example:

```
less: s * s \rightarrow s
less(a,b) ::= a when a<b
```

The s in the signature indicates a scalar or atom. The signature above denotes that less maps a scalar and a scalar to a scalar (or a null value). To indicate a sequence of nesting level or order (1), one states [s], of nesting level or order (2) or a matrix one states [[s]], etc.

SequenceL functions are paired with arguments in the tableau. As previously seen in the operator examples, the successive applications of NT's are performed with reference to the tableau. For example:

```
(less((10,60,30,40),35))
```

Since *less* takes two arguments it immediately consumes the two arguments to its right, since they do not need evaluation themselves (i.e., they are comprised strictly of constants, and do not contain operators or function symbols). Since *less* requires two scalars, an *NT* is performed on the arguments and the function symbol:

```
((less, less, less, less), (10,60,30,40), (35, 35, 35, 35)) = (less(10,35), less(60,35), less(30,35), less(40,35))
```

These occurrences of *less* can be evaluated iteratively or in parallel. Since the arguments are now of the expected level of nesting the function bodies are grounded in the next occurrence of the tableau:

```
(10 \text{ when } 10 < 35, 60 \text{ when } 60 < 35, 30 \text{ when } 30 < 35, 40 \text{ when } 40 < 35) = (10 \text{ when true, } 60 \text{ when false, } 30 \text{ when true, } 40 \text{ when false)} = (10, null, 30, null] = (10,30)
```

In the absence of an *else* in the function body, *false* conditions in the *when* clauses result in *nulls*, which disappear in the final result.

Consider the flexibility in the *less* function. Same function, different arguments:

```
(less((10, 20, 30, 40), (12, 24, 35)), 25)) = {}^{NT} \\ (less((10, 20, 30, 40), 25), less((12, 24, 35), 25)) = {}^{NT} \\ (((less(10,25), less(20,25), less(30,25), less, (40,25)), (less(12,25), less(24,25), less(35,25))) = \\ (((10 when 10 < 25, 20 when 20 < 25, 30 when 30 < 25, 40 when 40 < 25), (12 when 12 < 25, 24 when 24 < 25, 35 when 35 < 25))) = \\ (((10 when true, 20 when true, 30 when false, 40 when false), (12 when true, 24 when true, 35 when false))) = \\ (((10, 20), (12, 24)))
```

The *less* function and its *greater* counterpart form helper functions for Quicksort:

```
great: s * s → s
great(a,b) ::= a when a>b

quick: [s] -> [s]
quick(a) ::=
    a when length(a) <= 1 else
    quick(less(a,a(1))) ++ [a(1)]++quick(great(a,a(1)))</pre>
```

Now let's consider how SequenceL operates when faced with more formidable tasks. Consider an algorithm for instantiating variables in an arithmetic expression. The parse tree of an expression, such as,

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

can be represented by a nested list,

$$[x,+,[[7,*,x),/,9]]$$

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

Here is a solution in Haskell. This solution treats the expression as a list of strings and includes directives (e.g., 'par') to parallelize the evaluation:

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, recursion does not enter into our basic mental picture of the process. Third, the use of recursion to traverse the data structure obscures these facts in the Haskell code.

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*.

The Lämmel and Peyton-Jones boilerplate-scraping approach almost achieves this goal. Using this approach, the instantiate algorithm becomes much less intimidating, along the lines of:

Here, a large amount of boilerplate recursion has been distilled down to a single readable control structure, expressed in the everywhere and mkT operators. But it's SequenceL that truly *scraps* the boilerplate:

```
instantiate s * s * s \rightarrow s
```

```
instantiate(var,val,exp) = val when (exp == var) else exp
```

Note that this solution maps intuitively to just two lines of the Haskell code, which express the base cases. This is the real meat of variable instantiation – the "schoolchild's picture" of the algorithm. The remainder of the Haskell code is dedicated to traversing the breadth and depth of the tree. SequenceL, in contrast, will traverse the tree automatically. Here is a trace of the function:

```
(ins, x, 3, (x, +, (7, *, x))) =
```

Since the third argument is expected to be a scalar, but instead contains three items (the third being a nested sequence), three copies of the function symbol, the first, and second arguments are made. A transpose is performed on these new sequences and the third argument resulting in:

```
((ins,x,3,x),(ins,x,3,+),(ins,x,3,(7,*,x))) =
```

The first two function references contain arguments matching the function signature and, consequently, require no further applications of NT. These two sets of function bodies can be grounded. The third function reference requires a further (nested) application of NT and indicates how SequenceL discovers nested algorithmic control structures, control structures that must be written by the programmer in other languages. All of the grounding and NT's can be performed in parallel.

```
(3 \text{ when } x=x \text{ else } x,3 \text{ when } +=x \text{ else } +,
((ins,x,7),(ins,x,*),(ins,x,x))) =
```

The first two function bodies are continuing to be evaluated as the third set of function bodies can be grounded - all of these evaluation and translation steps can be performed in parallel:

```
(3 when true else x,3 when false else +,

(3 when 7=x else 7, 3 when *=x else *, 3 when x=x else x)) =
```

At this point, the first two function evaluations have provided their answers, and the next three function bodies (applied to the nested formula) are evaluated in parallel:

(3, +, (3 when false else 7, 3 when false else \*, 3 when true else x)) = Here is the final answer/tableau:

```
(3, +, (7, *, 3))
```

The NT indeed discovers all parallel and most iterative algorithms automatically. There do remain problems, however, like Quicksort, for which recursion is required.

# 3.0 Recursion in SequenceL

Due to the manner in which the SequenceL translation iteratively rewrites tableaus (until there is no change in the tableau), recursion requires no additional semantic definition. Furthermore, continuations (SW00) are a byproduct of this approach. In SequenceL, a function is unconditionally (in the absence of a *when* clause) or conditionally (in the presence of *when* clause(s)), indicating what will appear in the next tableau. A function can place in the tableau a reference to any SequenceL operator or user-defined function symbol. Thus, a function can place itself in the next tableau. Because there is no true calling or returning in a function, conventional activation records are not built. All intermediate information is contained in subsequent tableaus, including the place where continuations of computations are to begin execution once the recursive steps have completed. Continuations are a natural byproduct of the SequenceL abstraction. Consider a recursive definition of *factorial* in SequenceL:

```
fact: s \rightarrow s

fact(n) ::= fact(n-1) *n when n>0 else 1

Here is a trace of the evaluation of 4!

(fact(4))

(fact(4-1) * 4 when 4 > 0 else 1) =

(fact(3) * 4) = (fact(3-1) * 3 when 3 > 0 else 1) * 4) =

(fact(2) * 3 * 4) = (fact(2-1) * 2 when 2 > 0 else 1) * 3 * 4) =

(fact(1) * 2 * 3 * 4) = (fact(1-1) * 1 when 1 > 0 else 1) * 2 * 3 * 4) =

(fact(0) * 1 * 2 * 3 * 4) =

(fact(0-1) * 0 when 0 > 0 else 1) * 2 * 3 * 4) = (1 * 1 * 2 * 3 * 4) = (24)
```

Notice that the build up of multipliers 4, 3, 2, and 1 to the right (in the highlighted steps) in the tableau is effectively like the building of continuations. Again, no conventional activation records are needed so none are built. As residuals (or continuations) can be computed they indeed are computed, and often in parallel with other evaluations.

Even in the earliest versions of SequenceL, (C96) we noticed that beyond the powers of the NT many other problems solved recursively might be circumscribed in a shorthand notation. In particular, as seen in the *factorial* example, there is often a desire to perform a computation on a list that is recursively generated. We use an ellipse notation to indicate the generation of lists. For example, in the *factorial* function, we would generate the values from I to n, and then multiply: \*(I,...,4) = \*(I,2,3,4) = 24.

We are working on a shorthand notation for more general cases. In the *factorial* example, quite a bit of default information is implied in (1,...,n). For example, the initial and exit conditions of an implied loop are given by the lower and upper bounds, I and n, respectively. Furthermore, a transition function that fills in the intervening values (i.e., from I to n) is implied to be an increment of I. For more general cases, initial states, formulae for transition functions, exit conditions, and the return value need to be specified. But note, even the simple generating capability that currently exists in SequenceL is quite powerful, especially when combined with other operations that result in NT's:

```
(0, ..., 3) *2+1, (1, ..., 4) ^2 =
((0,1,2,3), (****), (2,2,2,2) (+,+,+,+), (1,1,1,1)), ((1,2,3,4), (^,^,^,), (2,2,2,2)) =
((0*2+1, 1*2+1, 2*2+1, 3*2+1), (1^2, 2^2, 3^2, 4^2,)) =
((1,3,5,7), (1,4,9,16))
```

Returning to Quicksort, not only is there a build-up of references to the Quicksort function, which (like in the Fibonacci example) can be evaluated in parallel, the NT's result in further parallelisms as they apply to the helper functions *great* and *less*. The SequenceL version of Quicksort seen in section 2 (and the NESL version below) assumes no duplicates in the input.

Quicksort is representative of an important class of parallel problems. Because it is a pivot-based problem, even when the data is presented to the function and the number of elements is known, one cannot predict apriori the schedule of parallel behavior. SequenceL finds the parallelisms even in a dynamic scheduling problem, with no annotation for parallel execution or, as in pH (NA01) a need to break apart nonscalar structures to crack open opportunities for parallel execution.

The parallelisms discovered by SequenceL are equivalent to those found in NESL (B96) given the NESL solution:

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

SequenceL has also been applied to other classes of parallel problems such as Matrix Multiply, Forward Processing in Gaussian Elimination, Discrete Wavelet Transforms, Jacobi Iteration, Fast Fourier Transform, and problems involving convergence such as Newton-Raphson approximation of roots of an equation. These problems are all solved in SequenceL by simple codes, with no annotations for breaking down data structures or indicating parallel or iterative structures. All parallelisms are discovered as a result of the interaction of the tableau and the NT rule. Many of these other examples can be seen in (CRAx), which also gives the formal definitions of SequenceL.

## 4.0 Summary

SequenceL possesses a simple, yet very powerful semantic that derives its power from the interaction of the Normalize-Transpose semantic rule and the iterative replacement of SequenceL terms with their simplifications in a shared memory abstraction called a tableau. This combined with the natural form of recursion (natural that is to the SequenceL abstraction), which incurs no stack overhead for activation records, results in a very small, yet powerful language from which most iterative parallel algorithms are discoverable and therefore, need not be specified by the programmer.

### References

- (B96) Guy Blelloch, "Programming Parallel Algorithms," March, 1996, Vol. 39, No. 3. *Communications of the ACM*, pp. 98-111.
- (BL93) Jean-Pierre Banater and Daniel Le Metayer, "Programming by Multiset Transformation, January, 1993, Vol. 36, No. 1. *Communications of the ACM*, pp. 98-111.
- (CG91) Daniel E. Cooke and A. Gates, "On the Development of a Method to Synthesize Programs from Requirement Specifications," *International Journal on Software Engineering and Knowledge Engineering*, Vol. 1 No. 1 (March, 1991), pp. 21-38.
- (C96) Daniel E. Cooke, "An Introduction to SEQUENCEL: A Language to Experiment with Nonscalar Constructs," *Software Practice and Experience*, Vol. 26 (11) (November, 1996), pp. 1205-1246.
- (C98) D. Cooke, "SequenceL Provides a Different way to View Programming," *Computer Languages* 24 (1998) 1-32.
- (CA00) Daniel E. Cooke and Per Andersen, "Automatic Parallel Control Structures in SequenceL," *Software Practice and Experience*, Volume 30, Issue 14 (November 2000), pp. 1541-1570.
- (CRAx) Daniel E. Cooke, J. Nelson Rushton, Per Andersen, Changming Ma, and Adem Ozyavas "Normalize, Transpose, and Distribute: A Basis for the Decomposition and Parallel Evaluation of Nonscalars," under review at *ACM Transactions on Programming Languages and Systems*.
- (LPJ03) Ralf Lämmel and Simon Peyton-Jones, "Scrap your boilerplate: a practical design pattern for generic programming," appeared in Proceedings of TLDI 2003, ACM Press.
- (LPJ04) Ralf Lämmel and Simon Peyton-Jones, "Scrap more boilerplate: reflection, zips, and generalised casts," to appear in Proceedings of ICFP 2004, ACM Press.
- (MFP91) Meijer, E. and Fokkinga, M.M. and Paterson, R., "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire" *FPCA* (Springer-Verlag, 1991) LNCS Series Vol. 523, pp. 124—144
- (NA01) R.S. Nikhil and Arvind, *Implicit Parallel Programming in pH*, Morgan Kaufmann, San Francisco, 2001.
- (SW00) Christopher Strachey and Christopher P. Wadsworth, "Continuations: A Mathematical Semantics for Handling Full Jumps," Higher-order and Symbolic Computation, Vol(13), pp.135-152, 2000

### Appendix: SequenceL Grammar

```
Pref
       ::= sin| cos | sqrt| | abs | sum | product| transpose
       ::=+|-|*|/|^| mod
Inf
Rel
        ::= < | > | <= | >= | == | <>
       ::= Term Rel Term | not Cond | Cond or Cond | Cond and Cond
Cond
        ::=,Term |,Term L
L
       ::= () | Id Term* | const | (Term L) | Pref Term | Term Inf Term |
Term
                Term when Cond else Term
S
       ::= Id : S^* \rightarrow S \ Id(Id^*) ::= Term
Func
       ::= Func*
Prog
```