

Taking Parnas' Principles to the Next Level – Declarative Language Design

Daniel E. Cooke
Computer Science Department
Texas Tech University – Abilene
302 Pine Street
Abilene, TX 79601
dcooke@coe.ttu.edu

J. Nelson Rushton
Computer Science Department
Texas Tech University
8th and Boston
Lubbock, TX 79409
nelson.rushton@gmail.com

Keywords:

D. Software; D.1 Programming Techniques; D.1.2 Automatic Programming; D.3 Programming Languages; D.3.2 Language Classifications: Applicative (functional) languages; D.3.3 Language Constructs and Features

ABSTRACT

SequenceL is a declarative language. Its abstraction is intended to enhance one's ability to declare data products in terms of form and content without having to specify how to disassemble or assemble non-scalars. It is a Turing Complete, domain-independent language that has been applied to NASA Guidance, Navigation, and Control Systems. In this paper we focus on the SequenceL abstraction and give evidence of our research goal to achieve information hiding i.e., to shield the specifier from the need to know how the language is implemented, and the fact that no single application domain is the object of the language's design.

Introduction

During the past seventeen years we have been experimenting with declarative approaches to computations on non-scalars. These efforts culminated in theoretical advances leading to the SequenceL language. The

language is Turing Complete, compact (12 grammar rules compared to JAVA's 150+), and its semantics are based solely on two simple computational laws: the Consume-Simplify-Produce and the Normalize-Transpose (CSP-NT), which are thoroughly described in [Cooke08]. During the seventeen-year period we created a dozen interpreters and four code generators used for compiling SequenceL solutions to C and C++. All of the example SequenceL code presented in this paper have been executed and verified on a SequenceL interpreter.

The most recently developed code generator runs Guidance, Navigation, and Control applications in the NASA simulator being used for Space Shuttle and Orion systems. The runtime performance of the code we generate from SequenceL is comparable to the performance of C code versions of the same applications hand written by NASA engineers. Best case results have shown that compiled SequenceL programs execute faster than their counterparts written in C by hand. In the worst case, SequenceL programs approached the performance of C within a factor of 2. [Cooke08]

In this paper we will focus on the SequenceL abstraction and the goal to achieve Parnas' information hiding in language design. Information hiding of software modules consists of two principles:

- One must provide the intended user with all the information needed to use the module correctly and nothing more.
- One must provide the implementer with all the information needed to implement the module correctly and nothing more. [Parnas]

Our intention is to demonstrate how these principles can be applied to language design as follows:

- One must provide the programmer with all the information needed to use the language correctly and nothing more.
- One must provide the language designer with all the information needed to implement the language correctly and nothing more.

We assert that part one is accomplished when the programmer does not need to know the operational semantics of the language in order to use it, and part two is accomplished when the language is not designed with a particular application domain in mind. We will provide evidence that our design of SequenceL, together with the development of principles for using the language, is quickly gaining ground in achieving these goals. To this end, we will focus on the SequenceL abstraction and not the semantic based refinements (i.e., the CSP-NT) leading to algorithms that solve a given problem.

When we discuss abstractions, it is worthwhile to review abstractions for other language approaches. The procedural abstraction is arguably best summarized in the title of Wirth's book, *Algorithms + Data Structures = Programs*. [Wirth] The Object Oriented approach extends the procedural abstraction with its additional features of Data Encapsulation, Inheritance, and Polymorphism. The Object Oriented approach is an attempt to satisfy Parnas's Principles of Information Hiding at the module level. [Parnas]

Functional programming is a style that emphasizes the evaluation of expressions, rather than the execution of commands. Expressions provided in functional languages are formed by using functions to combine basic values. [Hutton] In SequenceL we extend the functional abstraction to include the ability

to declare data products in terms of form and content without having to state the recursive or iterative algorithms to break nonscalars apart, or to reassemble or assemble nonscalars. SequenceL allows for recursion when needed since some problem solutions are best stated as recursive functions, e.g., Quicksort. The difference is that in SequenceL the need to deploy recursion is significantly diminished.

This paper will present motivations for improved abstractions, examples of transparency in problem solving, basic knowledge needed to use SequenceL, principles for using the language, and a glimpse of the range of domains to which the language can be applied. To use an analogy, we will present here how one “drives the car” (i.e., uses the language) and how one can be shielded from the knowledge of what goes on “under the hood” (i.e., how the language produces a procedural solution to the problem). To see the language semantics from a mathematical point of view one can read [Cooke08] in which we define the two computational laws which enable all of the features of SequenceL: the Consume-Simplify-Produce and the Normalize-Transpose (CSP-NT).

Motivation for Improved Abstractions

There is a modern legend about a truck being driven under an overpass. The overpass lacks the clearance necessary for the truck’s height. The truck becomes wedged under the bridge and its driver is unable to go forward or backward. Engineers and emergency personnel are called to the site and offer proposals to solve the problem, including several that involve using high-powered jacks to raise the bridge the few additional inches the truck needs for clearance. The story concludes with a child, who is watching the activities, voicing the ultimate solution: to deflate the front tires of the truck so that it can be backed out of its predicament. One can imagine many messages in the story. Among them are these:

1. The engineers saw the problem in terms of the tools that could be applied to the problem and these very tools limited their vision of the solution space; and
2. The child was not distracted or confined by the solution space that relevant tools brought to bear on the problem.

Lately, many resources have been dedicated to the development of tools to help programmers use modern programming languages. The languages are large and complex, and the tools are meant to assist programmers in managing these complexities. In the present context, it is worth recalling the view, first voiced decades ago, that programming languages provide abstractions to help programmers organize the complexity of problem solutions. [Dahl] If this is true, then the tools currently being developed are meant to help programmers manage the complexity of the languages they use to manage the complexity of problem solving. As a discipline we have become so accustomed to this view that many computer scientists now believe that one must have complicated tools to solve complicated problems. Simon Peyton Jones, one of the originators of Haskell, is quoted as saying “any language large enough to be useful must be dauntingly complex.” [Jones] When it comes to computer languages we tend to take C.A.R. Hoare’s view that “The price of reliability is the pursuit of utmost simplicity.” [Hoare]

The more complicated tools are, the more they tend to confine and distract us in problem solving. These complicated tools may even incline us towards more complicated problem solutions. We applaud the ongoing efforts to improve our ability to apply procedural and object-oriented languages to complicated

problems. However, we believe there are many opportunities for languages that are simpler to use. Languages are needed that improve our ability to view solution spaces while abstracting away the details of *how* solutions are executed. We feel such languages have the potential to draw one towards simpler and more reliable solutions.

Transparency.

Just like the engineers who were trying to free the wedged truck, software developers often immediately think about the procedural scaffolding of a problem solution. This tendency can distort their view of the problem itself. This is not meant to be an indictment or criticism of software developers in any respect. When we think of achieving any goal we typically first try to envision how known tools can help us. The fact that software developers often first consider the procedural aspects of problem solving is a natural byproduct of their training and experience, and is indicative of the fact that they tend to do their work the way engineers in other fields do theirs.

So what is this procedural “scaffolding” that affects our ability to address problems? Consider the following segment of code:

```
procedure myst(v1, v2 : integer; m1,m2 : tMatrix; var result : tMatrix );
var
  i, j, k, val: integer;
begin
  Dec(v1); Dec(v2);
  For i := 0 To v1 do
  begin
    For j := 0 To v2 do
    begin
      val := 0;
      For k := 0 To v2 do
      begin val := val + (m1[i, k] * m2[k, j]);end;
      result[i, j] := val;
    end;
  end;
end;
```

What is the desired computation specified by this procedure? Unfortunately, the procedural “scaffolding” used to describe how one achieves the goal obscures the abstract relationship between the procedure’s input and its output. This scaffolding is similar to the scaffolding surrounding a building under repair or being built. The difference between erecting buildings and writing programs is that architects are not compelled to focus on the scaffolding – instead, they create a direct representation of the desired outcome. Similarly a SequenceL programmer can focus exclusively on the desired outcome. An executable SequenceL specification equivalent to the procedural code above is as follows:

```
mysti,j (matrix m1,matrix m2) = Sum(m1(i,all) * m2(all,j))
```

Even with poor identifiers, this function is identifiable as matrix multiplication, because it *looks* like matrix multiplication. **Transparency** is what we call this ability to see at a glance what computation is being specified. Although one may argue that an experienced programmer might quickly recognize that the procedural code also implements matrix multiplication, the programmer may have to spend a bit of time visually inspecting the loops, and the respective subscripts they manage, to verify that the code is correct. The experienced programmer will decipher this information based upon knowledge of at least the operational semantics of the imperative statements comprising the code. Matrix Multiplication in the functional language Haskell is written as:

```
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]
```

We argue that this solution is less transparent than the SequenceL version, not just because it is longer, but also because one must know what the *zip* in the *dotProd* function does, and the fact that each column of *b* must be transposed.

We hypothesize that the simplification of the matrix-multiply function provided by SequenceL is characteristic of a wide range of programming problems, scaling up to applications of commercial interest and value. To test this hypothesis, we have been working on generalizing semantics to provide facility for the declaration of solutions for a wide range of iterative problems. Furthermore, we have been testing the language against serious and complex, real-world applications.

Fundamental Knowledge of SequenceL – Scalars and Nonscalars are Created and Treated Equally.

A scalar is an element of any atomic data type. For example, in SequenceL the integer *10*, the floating point number *2.4*, and the string *abc*, are all scalars. Nonscalars are ordered collections of scalars and/or nonscalars, e.g. vectors and matrices. The following are examples of nonscalars in SequenceL syntax:

```
[1.2, 2.43, 3.1]
[the, fox, jumped, the, creek]
[[1, 2, 3], [10, 20, 30], [100, 200, 300]]
[[1, 2, 3], 4.5, [10, 20, 30], [100, 200, 300]]
[1, [2, [3, 4]], [5]]
```

The SequenceL abstraction allows a problem solver to use operators and functions to combine:

```
scalars and scalars
scalars and nonscalars
nonscalars and nonscalars
```

The programmer can treat scalars and nonscalars equally in the following sense: in other languages most operations are defined on scalars. To process a nonscalar, the programmer must break it apart to find the scalars upon which operators can operate (as seen in the procedural matrix multiplication above). In SequenceL, the programmer is typically freed from this obligation. The operators are defined to work on nonscalars and scalars and do so in a predictable, consistent, and intuitive manner. Thus, scalars and nonscalars are treated equally.

Arithmetic - Conditionals.

One can apply any operator to any SequenceL term. A SequenceL *term* can be:

- a scalar;
- a nonscalar;
- a variable; or
- a built-in or user-defined operator applied to a SequenceL *term*

Let's first consider the SequenceL operators. They include the typical unary and binary mathematical and relational operators; structural operators like the *append*, subscripts, ellipses, and so-called *when-else* clauses, which are used to write user-defined function bodies.

Arithmetic expressions involving scalars are evaluated as they are in any language. For example (where $a \rightarrow b$ is read as *a evaluates to b in SequenceL*),

$$(4 + 3)^2 \quad \rightarrow \quad 49$$

Replacing *4* in the above left hand expression with a list $[1,2,3]$, we illustrate how SequenceL operations automatically operate on nonscalars:

$$([1,2,3] + 3)^2 \quad \rightarrow \quad [4,5,6]^2 \quad \rightarrow \quad [16,25,36]$$

Notice, many operations of vector arithmetic are naturally expressed in the language by virtue of the same semantics, without any user-defined or library operations::

$$\begin{array}{l}
 [1,2,3] * 2 \quad \rightarrow \quad [2,4,6] \\
 \left(\begin{array}{l} [1,2,3] \\ [4,5,6] \\ [7,8,9] \end{array} \right) * 4 \quad \rightarrow \quad \left(\begin{array}{l} [4, 8, 12] \\ [16, 20, 24] \\ [28, 32, 36] \end{array} \right)
 \end{array}$$

$$\begin{pmatrix} [1,2,3] \\ [4,5,6] \\ [7,8,9] \end{pmatrix} * [2,4,5] \rightarrow \begin{pmatrix} [2, 8, 15] \\ [8, 20, 30] \\ [14, 32, 45] \end{pmatrix}$$

Even arithmetic on complicated structures works in an analogous fashion and is consistent with all of the examples above:

$$[1, [2, [3, 4]], [5]] + [10, [20, [30, 40]], [50]] \rightarrow [11, [22, [33, 44]], 55]$$

Unary operators work on scalars and non-scalars:

$$\begin{aligned} \text{sqrt}(36) &\rightarrow 6 \\ \text{sqrt}([9, 16, 25]) &\rightarrow [3, 4, 5] \end{aligned}$$

Relational operators work on scalars and non-scalars:

$$\begin{aligned} 2 < 4 &\rightarrow \text{true} \\ [1, 2, 3] < 4 &\rightarrow [\text{true}, \text{true}, \text{true}] \\ \text{and}([1, 2, 3] < 4) &\rightarrow \text{true} \end{aligned}$$

Subscripts work on scalars and non-scalars:

$$\begin{aligned} \text{abcd}(2) &\rightarrow \text{b} \\ \begin{pmatrix} [1,2,3] \\ [4,5,6] \\ [7,8,9] \end{pmatrix} (2,2) &\rightarrow 5 \\ \begin{pmatrix} [1,2,3] \\ [4,5,6] \\ [7,8,9] \end{pmatrix} (3) &\rightarrow [7,8,9] \\ \begin{pmatrix} [1,2,3] \\ [4,5,6] \\ [7,8,9] \end{pmatrix} (\text{all},2) &\rightarrow [2, 5, 8] \\ \begin{pmatrix} [1,2,3] \\ [4,5,6] \\ [7,8,9] \end{pmatrix} ([2,3],[2,1]) &\rightarrow \begin{pmatrix} [5,4] \\ [8,7] \end{pmatrix} \end{aligned}$$

A *generative term* is one that fills in the values between lower and upper bounds: $[lower, \dots, upper]$.
 Generative constructs work on scalars and nonscalars:

$$\begin{array}{l} [2, \dots, 7] \qquad \qquad \qquad \rightarrow \quad [2, 3, 4, 5, 6, 7] \\ [[2, 4], \dots, [7, 8]] \qquad \rightarrow \quad [[2, \dots, 7], [4, \dots, 8]] \quad \rightarrow \quad [[2, 3, 4, 5, 6, 7], [4, 5, 6, 7, 8]] \end{array}$$

Conditionals work on scalars and nonscalars:

$$\begin{array}{l} 3 \textbf{ when } 3 < 4 \textbf{ else } 4 \qquad \qquad \qquad \rightarrow \quad 3 \\ [1, 2, 3, 4, 5] \textbf{ when } [1, 2, 3, 4, 5] < 4 \textbf{ else } 4 \quad \rightarrow \quad [1, 2, 3, 4, 4] \end{array}$$

In SequenceL, these terms can be nested and combined in an unrestricted fashion. It is worth noting that no special operators or recursive calls are needed to break apart or reassemble the nonscalars *and* that all of what we have seen so far, and will see later, is handled by a series of simplifications provided for by the Consume-Simplify-Produce and Normalize-Transpose semantics.

Using Principles of SequenceL to Write Functions

To achieve better transparency in problem solutions there is a need to address information hiding – similar to the principles David Parnas developed decades ago. [Parnas] Essentially, Parnas’ principles concern compartmentalization – that the user of an operation has no knowledge of its implementation *and* that the implementer of an operation has no knowledge of particular applications. [Parnas] In SequenceL, the CSP-NT are abstract semantics that enable all of the non-primitive SequenceL constructs. There are no specialized operators as there are in other languages – operators requiring specialized knowledge about the operators’ semantics. The only semantics the SequenceL user would need to know are the CSP-NT.

We are working toward developing principles for using SequenceL which encapsulate the semantics of CSP-NT into a set of high level principles. In this and the following sections we will demonstrate that these principles can be applied across a broad range of applications. Thus, we attempt to achieve information hiding at the language level itself.

A key issue to consider in the development of SequenceL functions is the level of nesting for which the operators are defined.

Principle 1 Base Cases: *For most problem solutions that require the disassembly and reassembling of nonscalar data structures one can state a very simple solution which amounts to stating the base case of the traditional recursive solution.* Consider the case where there is a list of *subscript/word* pairs, e.g., $[[1, \text{dog}], [2, \text{cat}], [3, \text{fox}], [4, \text{parrot}]]$, and a function is needed to provide the subscript of a target word if it appears in the list of tuples. In most languages the solution requires an iterative or recursive solution that breaks the list into its tuples and returns the subscript of the word that matches a target word. In SequenceL, one simply states the basic computation:

```
Search(scalar Target, tuple [Subscript, Word]) =
  Subscript when Word = Target
```

When a list of *subscript/word* tuples is applied to *search*, SequenceL disassembles the structure and returns the *appropriate* subscript if the word is in the list, otherwise it returns an empty list. No recursion or special operators are needed:

```
search(fox, [[1, dog], [2, cat], [3, fox], [4, parrot]]) → 3
search(rabbit, [[1, dog], [2, cat], [3, fox], [4, parrot]]) → []
```

This works because the basic computation is defined at the tuple level. When a list of tuples is applied to the function, the language itself breaks the list into the tuples the function expects. Consider another example in which one wants to extract the even numbers from a list of integers. Once again the problem solver states the basic computation and lets the language break the list of integers down and assemble the final result:

```
Evens(scalar Number) = Number when Number mod 2 = 0
```

Given a list of numbers the function returns only the even elements. Again, no recursion or special operators are needed:

```
evens( [1, 2, 3, 4, 5, 6, 7]) → [2, 4, 6]
```

Both numeric and non-numeric computations on nonscalars benefit from the SequenceL abstraction and constructs. SequenceL's abstraction can be applied to string replacement, which is another example of Principle 1. For example, when one is writing a language interpreter and wants to instantiate values for the variables of an expression, the required SequenceL function is simply:

```
ins(scalar Character, vector ID, vector Value) =
  Value when ID=Character else Character
```

Given the following inputs, the predicted results are obtained:

```
ins([(z * x) / (x + y)], [x], [8]) → [(z * 8) / (8 + y)]
ins([(z * x) / (x + y)], [x, y, z], [8, 7, 4]) → [(4 * 8) / (8 + 7)]
```

Note the SequenceL solution above is essentially a transcription of only the base case of a typical recursive solution to the variable instantiation problem.

Principle 2 – Equation Simplifications: *When there are patterns in equations, some which vary and some which are constant, the constant information does not need to be repeated.* Consider the following example to compute Euler's number:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

Notice that the numerator remains constant and the denominator varies. In SequenceL, the function is best described as:

$$e(\text{scalar } N) = \text{sum}(1/\text{fact}([0, \dots, N]))$$

where N is a limit for discrete computation of Euler's number

Consider further part of the definition of a quaternion from one of the NASA applications we have developed:

$$q_1 = \frac{1}{2} \sqrt{r_{11} + r_{22} + r_{33} + 1}$$

In SequenceL, the translation is straightforward and transparent:

$$q_1(\text{matrix } R) := \text{sqrt}(\text{sum}([R(1,1), R(2,2), R(3,3), 1])) / 2$$

The remaining equations are:

$$q_2 = \frac{r_{32} - r_{23}}{4q_1}$$

$$q_3 = \frac{r_{13} - r_{31}}{4q_1}$$

$$q_4 = \frac{r_{21} - r_{12}}{4q_1}$$

Notice that the numerators vary and the denominators are constant. In SequenceL the equation is simplified so it is not necessary to repeat constant information:

$$\text{quaternions}(\text{matrix } R) :=$$

$$[$$

$$[R(3,2) - R(2,3), R(1,3) - R(3,1), R(2,1) - R(1,2)]$$

$$/ (4 * q_1(R))$$

$$]$$

Principle 3 – Using Subscripts: *Subscripts are needed when (1) they appear as a condition to a computation, (2) they contain constants, and (3) when the left-hand side use of the subscripts varies from the right-hand side use.*

For case (1) consider the definition of the identity matrix:

```
identity_matrixI,J(matrix M) = 1 when I = J else 0
```

For case (2) consider the following example specification of Jacobi Iteration. The actual specification for one iteration is:

$$\mu_{j,k}' = (\mu_{j+1,k} + \mu_{j-1,k} + \mu_{j,k+1} + \mu_{j,k-1})/4 - ((\rho_{j,k} * \Delta^2)/4)$$

when neither j or k subscript the first or last row or the first or last column

where μ and ρ are input matrices for each iteration and μ' serves as μ in the next iteration. Considerable work and knowledge of semantics is required to see that the following Haskell Code implements the specification:

```
jacobi a delta =
  map (\i -> [jacobi_helper a delta i j | j <- [0..(length (a!!i))-1]]) [0..(length a)-1]

jacobi_helper a delta i j
  | i==0 || j==0 || (length a)-1 == i || (length (a!!i))-1 == j = a!!i!!j
  | otherwise =
    (a!!(i+1)!!j + a!!(i-1)!!j + a!!i!!(j+1) + a!!i!!(j-1))/4 - ((a!!i!!j) * delta^2)/4
```

Notice, that the *jacobi_helper* is close to the specified equation. However, a second order operator, the *map*, is used to distribute the *jacobi_helper* over the elements of the matrix. Therefore, one must know how this specialized operator works, and work through the *i,j* manipulations, to be certain that indeed the two functions implement the specified equation.

In *SequenceL*, one can see that the solution maps directly to the specifying equation with minimal effort:

```
jacobiJ,K(matrix a,b, scalar delta) =
  (a(j+1,k) + a(j-1,k) + a(j,k+1) + a(j,k-1)) / 4 - ((b(j,k)*delta^2)/4)
  when not((j =1 or size(a)=j) or (k=1 or size(a)=k)) else a(j,k)
```

Notice that all of the operators are intuitive and no specialized functions need to be known so that the *SequenceL* function aligns well with the specifying equation. Furthermore, a natural language definition tends to spring from the *SequenceL* definition: *The (j,k)th element of the resulting matrix is the average of its surrounding elements below, above, to the right, and to the left, when j is not the first or last row and k is not the first or last column.*

A matrix transpose function is an example of **Case 3**:

$$\text{transpose}_{i,j}(\text{matrix } M) = M(J, I)$$

as is the matrix multiplication from an earlier section:

$$\text{Mat_Mul}_{i,j}(\text{matrix } m1, \text{matrix } m2) = \text{Sum}(m1(i, \text{all}) * m2(\text{all}, j))$$

This definition declares that the i,j th element of the function is defined as the sum of the products of the respective members of the i th row of $m1$ and the j th column of $m2$.

Principle 4 – Using Recursion: *If the problem definition is most easily visualized as recursive, then recursion is typically required in the SequenceL definition.* The specification of Quicksort follows:

1. Choose a pivot from the unsorted list;
2. Create a list that contains, first, a sort of all items<pivot, followed by the pivot, which is followed by the sort of all items>pivot;
3. Actions 1 and 2 are repeated until a list of one item or a scalar is the argument of the quicksort.

In SequenceL quicksort is:

$$\text{Quick}(\text{any } X) = X \text{ when scalar}(X) \text{ else} \\ \text{appends}(\text{ quick}(X \text{ when } X < X(1)), X(1), \text{ quick}(X \text{ when } X > X(1)))$$

Principle 5 – Transcribing Equations: *In general any effective formal definition can be clearly and easily transcribed into an executable SequenceL definition.* Evidence for this principle also illustrates the fact that SequenceL was not defined with a particular application domain in mind, and this is the topic of the next section.

More on the Range of SequenceL Applications.

In the previous section, a range of problems was used to illustrate the basic principles of writing SequenceL code. Here we will broaden the range to include other areas of formal definitions and natural language requirements. Initial efforts to develop principles have focused on equation-based specifications like the Jacobi example above. We have explored equations beyond numerical computations to include ones from set-builder notation, denotational semantics, and others. In all cases explored so far, one can simply translate the equation directly into SequenceL using the principles. Examples from set builder notation follow:

$$\text{in}(\text{scalar } X, \text{ any } S) = \text{or}(X = S)$$

The *or* operator is acting as an existential quantifier on a series of finite equalities, while the *and* in the next example acts like a universal quantifier. Note that these follow the definitions of the universal and existential operators.¹

```
subset(sets S1,S2)           =      and(in(S1,S2))
set_equality(sets S1,S2)    =      subset(S1,S2) and subset(S2,S1)
```

Just as the definitions above are intuitively linked to set theory, the following look very similar to their set builder equivalents.

```
S1∩S2
intersection(sets S1,S2)    =      {x | x∈S1 & x∈S2}
                             =      {X when in(X,S1) and in(X,S2) }

S1∪S2
union(sets S1,S2)          =      {x | x∈S1 OR x∈S2}
                             =      {X when in(X,S1) or in(X,S2) }

S1-S2
difference(sets S1,S2)      =      {x | x∈S1 & x∉S2}
                             =      {X when in(X,S1) and not(in(X,S2)) }

S1×S2
cartesian_product(sets S1,S2) =      {<x,y> | x∈S1 & y∈S2}
                             =      {[X,Y] when in(X,S1) and in(Y,S2) }
```

Consider the following denotational semantic equation for the *while* loop:

$$\Sigma \llbracket \text{while } C \text{ do } S \text{ od} \rrbracket \sigma = \Sigma \llbracket \text{while } C \text{ do } S \text{ od} \rrbracket \cdot \Sigma \llbracket S \rrbracket \sigma \text{ when } \zeta \llbracket C \rrbracket \sigma \text{ else } \sigma$$

Here is the SequenceL version:

```
Sigma( any [while,C,do,S,od], Matrix State)      =
                                             Sigma([while,C,do,S,od],Sigma(S,State))
                                             when Gamma(C,State) else State
```

Notice in these cases, too, that a straightforward transcription from the specifying equations is possible in SequenceL. With simple transcription from semantic equations to SequenceL we have specified a Turing Complete *while* language. The SequenceL version is virtually identical to respective semantic equations and will execute programs written in the language, given an initial state. We have also transcribed definitions of Tuple Relational Calculus from database theory into SequenceL definitions *and* these SequenceL definitions are executable.

We have only recently initiated studies of the relationship between Natural Language specifications and SequenceL. We are focused on requirements for abort systems we are developing in collaboration with NASA Johnson Space Center's Guidance, Navigation, and Control engineers.

¹ For example $\forall X (p(X)) = p(x_1) \& p(x_2) \& \dots p(x_n)$ assuming a finite domain for X.

One aspect of this work is sorting Trans-Atlantic Landing Sites (SortTAL sites), which is one component of NASA’s Shuttle Abort Flight Management System. The function arranges the top four preferred abort landing sites according to pre-defined priorities. These are targeted landing options available in the event of an abort during the ascent (or launch) phase. As input, SortTAL sites takes four vectors, each representing a highly ranked abort landing site. Each vector has a subscript (1-4) identifying the site, a throttle value, a *valid* flag, an *available* flag, and a geographical location. What follows is a portion of the executable requirements in SequenceL. After the SequenceL requirement, we show the Natural Language requirements.

Sort_TAL – component 1

```
preferred(vector B, A, scalar Prime) =
  B when and ([B:subscript = 4, B:available, B:valid]) else
  [] when and ([A:subscript = 4, A:available, A:valid]) else
  B when and ([B:location = Prime, B:available, B:valid]) else
  [] when and ([A:location = Prime, A:available, A:valid]) else
  B when and ([B:throttle < A:throttle, B:available, B:valid]) else
  B when and ([B:throttle < A:throttle, B:available]) else
  B when A:subscript > B:subscript
```

Natural Language:

1. Site B is preferred to site A when B is the 4th site and is available and is valid. However Site A overrides B when it is the 4th site and is available and is valid; otherwise
2. B is preferred to A when B is the prime site and is available and is valid. However Site A overrides B when it is the prime site and is available and is valid; otherwise
3. B is preferred when B’s throttle value is less than A’s throttle value and B is available and is valid; otherwise
4. B is preferred when B’s throttle value is less than A’s throttle value and B is available; otherwise
5. B is preferred when A was entered after B in the list of sites.

We are in the process of creating Natural Language specifications from SequenceL definitions to identify patterns and a restricted grammar for producing SequenceL from Natural Language Specifications, and vice versa.

Summary and Conclusions.

Although SequenceL is effective on a wide range of applications, we have had our more industrial strength work done at NASA. From this NASA work we have consistent and strong anecdotal evidence of SequenceL’s transparency. SequenceL has been applied to three major NASA Guidance, Navigation, and Control applications. The abort system prototypes for the Space Shuttle and the new Orion Crew Exploration Vehicle have been developed in SequenceL, and most recently we have completed a system that predicts the near future trajectory of a space vehicle, as a function of its current position, velocity, control settings, and anticipated crew commands. This system is critical to abort determination in that it gives altitude and velocity information needed to determine feasible aborts, which can include abort to orbit, return to Kennedy Space Center, or to proceed to one of several landing sites on the east coast or

overseas. When we presented the SequenceL “predictor” code, the aerospace engineers in the audience, having no knowledge of how SequenceL works, volunteered that the code was clear and understandable at the requirements level – more understandable than their own requirements document. Additionally the SequenceL predictor requirements are executable – so NASA’s specification is now also a prototype. In running this prototype we found errors in our requirements that would normally go unnoticed in a static requirements document.

Our recent work on a proof-of-concept interpreter has shown that apart from semantics to perform scalar arithmetic and other primitive operations, we can implement *all other language constructs* elegantly in terms of the CSP-NT semantics. [Cooke08] In other words, once the primitive operations and the CSP-NT exist we can implement the instantiation of function arguments, subscripting of structures, evaluation of function bodies, and handling of function references. These semantics illustrate clear differences between SequenceL and other languages, including Haskell, FP, APL, NESL, and Gamma. [Hudak, Backus, Iversen, Blelloch, Banatre] These differences are reported in [Cooke96, Cooke00, Cooke08].

In this paper we presented the abstraction of SequenceL, a language designed to enhance one’s ability to declare data products in form and content without having to specify the recursive or iterative scaffolding to disassemble or reassemble nonscalars. The goals of our SequenceL effort are to achieve transparency in specification and automatic synthesis of correct procedural codes from those specifications.² Key to resolving some of the complexity of program synthesis is the fact that SequenceL’s semantics are simple and based on only two computational laws. Key to transparency is the development of a small set of principles to guide the specifier when writing specifications in formal and natural languages. As a result, the overarching goal of SequenceL is to simplify problem solving to achieve greater productivity and reliability by removing a significant amount of technical detail that otherwise obscures solution spaces. Syntax errors are statically handled by the SequenceL translators, while semantic errors are largely handled at runtime. Ongoing research is focused on a verification logic for SequenceL to provide formal checks of correctness.

In this paper, we also presented evidence that the principles of information hiding can help direct language design. SequenceL’s transparency and the definition of principles for using it are intended to satisfy the first principle of Information Hiding: that the programmer does not have to know the imperative semantics of the language to use it effectively. Domain independence of SequenceL satisfies the second principle.

This research was supported by NASA-NNG06GJ14G, NASA-CAN NNJ06HE94A, and Abilene Community/Sheldon Fund grant 146144C571

² To support, respectively, validation and verification.

References

- [Backus] John W. Backus: Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. *Commun. ACM* 21(8): 613-641 (1978)
- [Banatre] Jean-Pierre Banatre and Daniel Le Metayer, "Programming by Multiset Transformation, January, 1993, Vol. 36, No. 1. *Communications of the ACM*, pp. 98-111.
- [Blelloch] Guy Blelloch, "Programming Parallel Algorithms," March, 1996, Vol. 39, No. 3. *Communications of the ACM*, pp. 98-111.
- [Cooke96] Daniel E. Cooke, "An Introduction to SEQUENCCEL: A Language to Experiment with Nonscalar Constructs," *Software Practice and Experience*, Vol. 26(11), (November, 1996) 1205-1246.
- [Cooke00] Daniel E. Cooke and Per Andersen, "Automatic Parallel Control Structures in SequenceL," *Software Practice and Experience*, Volume 30, Issue 14, (November 2000), 1541-1570.
- [Cooke08] 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 (February 2008), 50 pages. DOI 10.1145/1330017.1330020 <http://doi.acm.org/10.1145/1330017.1330020>
- [Dahl] O.J Dahl, E. W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, London. 1972.
- [Hoare] Charles Antony Richard Hoare, The emperor's old clothes, *Communications of the ACM*, v.24 n.2, p.75-83, Feb. 1981 [doi>10.1145/358549.358561]
- [Hudak] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, John Peterson: Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. *SIGPLAN Notices* 27(5): R1-R164 (1992)
- [Hutton] Graham Hutton, <http://www.cs.nott.ac.uk/~gmh/faq.html#functional-languages>.
- [Iverson] Iverson, K. *A Programming Language*, Wiley, New York (1962).
- [Jones] Simon Peyton Jones, "Wearing the hair shirt A retrospective on Haskell," 2002, research.microsoft.com/~simonpj/papers/haskell-retrospective/HaskellRetrospective.pdf.
- [Parnas] David Parnas, "On Criteria To Be Used in Decomposing Systems Into Modules," *CACM*, vol. 14 no. 1, April, 1972, pp. 221-227.
- [Wirth] Niklaus Wirth, *Algorithms & Data Structures*. Prentice-Hall, 1986.