Taking Parnas’s Principles to the Next Level: Declarative Language Design

Daniel E. Cooke and J. Nelson Rushton, Texas Tech University

SequenceL is a general-purpose functional programming language that exploits information hiding in language design, shielding the programmer from the need to know how the language is implemented.

During the past 17 years, we’ve been experimenting with declarative approaches to computations on nonscalars. These efforts culminated in theoretical advances leading to the SequenceL language. The language is Turing complete and compact—12 grammar rules compared to Java’s more than 150—and its semantics are based on two simple computational laws: consume-simplify-produce and normalize-transpose (CSP-NT).

During this period, we created a dozen interpreters and four code generators for compiling SequenceL solutions to C and C++, and we executed and verified all of the example SequenceL code appearing in this article on a SequenceL interpreter.

One of our most recent code generators runs guidance, navigation, and control applications in the NASA simulator for the space shuttle and the Orion crew exploration vehicle. The SequenceL-generated code’s runtime performance is comparable to the performance of C code versions of the same applications that NASA engineers handwrote. Best-case results show that compiled SequenceL programs execute faster than their counterparts handwritten in C; worst-case results show that SequenceL programs approached C’s execution time within a factor of 2.

We’ve recently revised one of the interpreters to generate multithreaded C++ code for the multicore processors.

We’re currently involved in an experiment with a third-party software company to rewrite an application for a major process controls company in SequenceL. The experiment aims to demonstrate productivity enhancements due to SequenceL’s transparency and performance improvements due to the parallelisms. All of the example SequenceL code in this article results in parallel execution.

For this discussion, we focus on the SequenceL abstraction and the goal of achieving David Parnas’s information hiding in language design. Information hiding of software modules is made up of two principles: providing the intended user with all the information needed to use the module correctly and nothing more, and providing the implementer with all the information needed to implement the module correctly and nothing more.

We’ll demonstrate how these principles apply to language design by providing the programmer with all the information needed to use the language correctly and nothing more, and providing the language designer with all the information needed to implement the language correctly and nothing more. We claim success with the first goal if the programmer doesn’t need to know the language’s operational semantics to use it, and with the second if the language isn’t designed with a particular application domain in mind. To this end, we focus on the SequenceL
abstraction and not semantic-based refinements, such as CSP-NT, which lead to algorithms for solving a given problem.

Our SequenceL design, coupled with developing principles for using the language, is quickly gaining ground in achieving these goals. To use an analogy, in this article we’ll discuss how to drive the car (use the language) without finding out what’s going on under the hood (how the language produces a procedural solution to the problem).

COMPARING ABstractions

When discussing abstractions, it’s worthwhile to review abstractions for other language approaches. The title of Niklaus Wirth’s book, *Algorithms + Data Structures = Programs* (Prentice Hall, 1976), best summarizes the procedural abstraction. The object-oriented approach extends the procedural abstraction with its additional features of data encapsulation, inheritance, and polymorphism. This approach attempts to satisfy Parnas’s information-hiding principles at the module level.

Functional programming is a style that emphasizes expression evaluation rather than command execution. Expressions provided in functional languages are formed by using functions to combine basic values. 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 necessary because some problem solutions are best stated as recursive functions, such as Quicksort. The difference between SequenceL and functional programming is that SequenceL significantly diminishes the need to deploy recursion. (In earlier work, we presented the language semantics from a mathematical viewpoint and defined CSP-NT.)

MOTIVATION FOR IMPROVED ABstractions

A modern legend tells of a truck getting wedged under an overpass because the overpass lacked the clearance necessary for the truck’s height. The driver can’t move the truck forward or backward. Engineers and emergency personnel offer solutions, including several involving high-powered jacks for raising the bridge the few additional inches the truck needs for clearance. The story concludes with a child voicing the ultimate solution: deflate the front tires of the truck so the driver can back it out of its predicament.

This story can have many messages. For example, the engineers saw the problem in terms of what tools to apply to the problem, and these very tools limited their vision of the solution space; or, the child wasn’t distracted or confined by the solution space that relevant tools could bring to bear on the problem.

Many resources are dedicated to developing tools to help programmers use modern programming languages. The languages are large and complex, and the tools aim to assist programmers in managing these complexities. Recall the view Ole-Johan Dahl and his colleagues first voiced decades ago that programming languages provide abstractions to help programmers organize the complexity of problem solutions. If this is true, then the tools many researchers are currently developing are meant to help programmers manage the complexity of the languages they use to manage problem-solving complexity.

The computing discipline has become so accustomed to this view that many computer scientists believe that complicated tools are necessary to solve complicated problems. As Simon Peyton Jones, one of Haskell’s originators, said “any language large enough to be useful must be dauntingly complex.”

When it comes to computer languages, we take C.A.R. Hoare’s view that “the price of reliability is the pursuit of utmost simplicity.” The more complicated the tools, the more they tend to confine and distract problem solving. These complicated tools might even steer programmers toward more complicated problem solutions. The ongoing efforts to improve programmers’ ability to apply procedural and object-oriented languages to complicated problems are laudable. However, many opportunities exist for easy-to-use languages. Programmers need languages that improve the ability to view solution spaces, while abstracting away the details concerning how to solve a problem. Such languages have the potential to draw programmers toward simpler and more reliable solutions.

TRANSPARENCY

Like the engineers who were trying to free the wedged truck, software developers often immediately think about a problem solution’s procedural “scaffolding.” This tendency can distort their view of the problem itself. This observation isn’t meant to indict or criticize software developers in any respect. When people think of achieving any goal, they typically first try to envision how known tools can
help them. That software developers often first consider problem solving’s procedural aspects is a by-product of their training and experience, indicating that they tend to work the way engineers in other fields do.

So, what is this procedural scaffolding that affects the ability to address problems? Consider the following code segment:

```haskell
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]
```

This solution is less transparent than the SequenceL version not only because it’s longer, but also because the programmer must know what the `zip` function does and that each column of `b` must be transposed.

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

### TREATING SCALARS AND NONSCALARS EQUALLY

A scalar is an instance of any atomic data type. For example, in SequenceL, the integer 10 and the floating-point number 2.4 and the string abc are scalars. Nonscalars are ordered collections of scalars, nonscalars, or both, such as 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]]`

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

- scalars and scalars,
- scalars and nonscalars, and
- 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 non-scalar, the programmer must break it apart to find the scalars on which operators can operate (as in the pre-
treated equally. Thus, scalars and nonscalars are typically freed the programmer from this obligation. The language defines operators to work on nonscalars and scalars, and they do so in a predictable, consistent, and intuitive manner. Thus, scalars and nonscalars are treated equally.

ARITHMETIC—CONDITIONALS

Any operator is applicable 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. 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 programmers use to write the bodies of user-defined functions.

SequenceL evaluates arithmetic expressions involving only scalars as they’re evaluated in any language—for example (where $a \rightarrow b$ is read as $a$ evaluates to $b$ in SequenceL): $(4 + 3) \rightarrow 2 \rightarrow 49$. Replacing $4$ in this expression with a list $[1, 2, 3]$, we illustrate how SequenceL operations automatically operate on nonscalars:

$$([1, 2, 3] + 3) \rightarrow [4, 5, 6] \rightarrow [2, 5, 8]$$

Notice that the language naturally expresses many vector arithmetic operations by virtue of the same semantics without any user-defined or library operations:

$$[1, 2, 3] \times 2 \rightarrow [2, 4, 6]$$

Even arithmetic on complicated structures works analogously and is consistent with all of the previous examples:

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

Unary operators work on scalars and nonscalars:

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

Relational operators work on scalars and nonscalars:

- $2 < 4 \rightarrow \text{true}$
- $[1, 2, 3] < 4 \rightarrow \text{true, true, true}$
- and $([1, 2, 3] < 4) \rightarrow \text{true}$

Subscripts work on scalars and nonscalars:

$$abcd(2) \rightarrow b$$

$$\begin{array}{c}
[1, 2, 3] \\
[4, 5, 6] \\
[7, 8, 9]
\end{array} \rightarrow
\begin{array}{c}
(2, 2) \\
5 \\
\end{array}$$

$$\begin{array}{c}
[1, 2, 3] \\
[4, 5, 6] \\
[7, 8, 9]
\end{array} \rightarrow
\begin{array}{c}
(3) \\
[7, 8, 9] \\
\end{array}$$

$$\begin{array}{c}
[1, 2, 3] \\
[4, 5, 6] \\
[7, 8, 9]
\end{array} \rightarrow
\begin{array}{c}
(\text{all,} 2) \\
[2, 5, 8] \\
\end{array}$$

$$\begin{array}{c}
[1, 2, 3] \\
[4, 5, 6] \\
[7, 8, 9]
\end{array} \rightarrow
\begin{array}{c}
([2, 3], [2, 1]) \rightarrow
\begin{array}{c}
[5, 4] \\
[8, 7]
\end{array}
\end{array}$$

A generative term fills in the values between lower and upper bounds: $[\text{lower, \ldots, upper}]$. Generative constructs work on scalars and nonscalars:

- $[2, \ldots, 7] \rightarrow [2, 3, 4, 5, 6, 7]$
- $[[2, 4], \ldots, [7, 8]] \rightarrow [[2, \ldots, 7], [4, \ldots, 8]]$

Conditionals work on scalars and nonscalars:

- $3 \text{ when } 3 < 4 \text{ else } 4 \rightarrow 3$
- $[1, 2, 3, 4, 5] \text{ when } [1, 2, 3, 4, 5] < 4 \text{ else } 4 \rightarrow [1, 2, 3, 4, 4]$

In SequenceL, the programmer can nest and combine these terms without restrictions. The language requires no special operators or recursive calls to break apart or reassemble the nonscalars. A series of simplifications that the CSP-NT semantics provide handled all of the examples in this article.

WRITING FUNCTIONS IN SEQUENCE L

Better transparency in problem solutions requires addressing information hiding similar to the principles Parnas developed decades ago. Essentially, Parnas’s principles concern compartmentalization: that an operation’s user has no knowledge of the operation’s implementation and that the operation’s implementer has no knowledge of particular applications. In SequenceL, the CSP-NT are abstract semantics that enable all the nonprimitive SequenceL constructs. SequenceL has no specialized operators as other languages do—that is, operators requiring specialized knowledge about the operators’ semantics. The only semantics the SequenceL user would need to know are the CSP-NT.

We’re working toward developing principles for using SequenceL that encapsulate CSP-NT’s semantics into a set of high-level principles. These principles can apply to a broad range of applications. We attempt to achieve information hiding at the language level itself. A key issue to
consider in developing SequenceL functions is the operators’ defined nesting level.

**Principle 1: Base cases**

For most problem solutions that require disassembling and reassembling nonscalar data structures, a programmer can provide a simple solution that amounts to stating the traditional recursive solution’s base case. Consider a list of subscript/word pairs, such as

\[[1, \text{dog}], [2, \text{cat}], [3, \text{fox}], [4, \text{parrot}]\]

and a function is needed to provide the subscript of a target word if the target word appears in the tuple list. In most languages, the problem 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. SequenceL simply states the basic computation:

\[
\text{Search(scalar Target, tuple [Subscript, Word]) = } \text{Subscript when Word = Target}
\]

When the programmer applies a list of subscript/word tuples 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 necessary:

\[
\text{search(fox,[[1, dog],[2, cat],[3, fox],[4, parrot]]) → 3}
\]

This process works because the function defines the basic computation at the tuple level. When the programmer applies a list of tuples to the function, the language itself breaks the list into the tuples the function expects. Consider another example in which the programmer 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:

\[
\text{Evens(scalar Number) = Number when Number mod 2 = 0}
\]

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

\[
\text{Evens([1,2,3,4,5,6,7]) → [2,4,6]}
\]

Both numeric and nonnumeric computations on nonscalars benefit from the SequenceL abstraction and constructs. SequenceL’s abstraction can apply to string replacement, which is another example of Principle 1. For example, when a programmer is writing a language interpreter and wants to instantiate values for an expression’s variables, the required SequenceL function is simply

\[
\text{ins(scalar Character, vector ID, vector Value) = Value when ID = Character else Character}
\]

The following inputs produce the predicted results:

\[
\text{ins([z * x) / (x + y)], [x], [8]) → [(z * 8) / (8 + y)]}
\]

\[
\text{ins([z * x) / (x + y)], [x, y, z], [8, 7, 4]) → [(4 * 8) / (8 + 7)]}
\]

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

**Principle 2: Equation simplifications**

When equations have patterns, some varying and some constant, the programmer needn’t repeat the constant information. Consider the following example to compute Euler’s number:

\[
e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots
\]

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

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

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

Consider further the definition of a quaternion from a NASA application we developed:

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

In SequenceL, the translation is straightforward and transparent:

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

The remaining equations are

\[
qu_2 = r_{32} - r_{23} / 4q_4, \quad q_3 = r_{13} - r_{31} / 4q_4, \quad q_4 = r_{21} - r_{12} / 4q_4,
\]

Notice that the numerators vary and the denominators are constant. SequenceL simplifies the equation so it’s not necessary to repeat constant information:

\[
\text{quaternions(matrix R)} : = \text{[R(3,2)-R(2,3),R(1,3)-R(3,1),R(2,1)-R(1,2)]/(4*q1(R))}
\]

**Principle 3: Using subscripts**

Subscripts are necessary in three cases: (1) when they appear as a condition to a computation, (2) when they contain constants, and (3) when the left-side use of the
subscripts varies from the right-side use.

For Case 1 consider the identity matrix’s definition:

identity_matrix(i,j)(matrix m) = 1 when i = j else 0

For Case 2 consider the following example specification of Jacobi iteration. The specification for one iteration is

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

where neither \( j \) nor \( k \) subscript the first or last row or the first or last column, and where \( \mu \) and \( \rho \) are input matrices for each iteration and \( \mu' \) serves as \( \mu \) in the next iteration. Understanding that the following Haskell code implements the specification requires considerable work and semantics knowledge:

\[
jacobi a \delta = \map (\forall i \rightarrow \mu j \rightarrow [jacobi Helper a \delta i j | j < -[0..(length (a!!i)))-1]]) [0..(length a)-1]
\]

\[
jacobi Helper a \delta i j | i == 0 || j == 0 || (length a)-1 == i || (length (a!!i)))-1 == j = a!!i!!j
\]

\[
| otherwise = (a!!(i+1)!!j + a!!(i-1)!!j + a!!i!!(j+1) + a!!i!!(j-1))/4 - (a!!i!!j) * \delta^2)/4
\]

Notice that the \( jacobi \) helper is close to the specified equation. However, a second order operator, the map, distributes the \( jacobi \) helper over the matrix elements. Therefore, the programmer 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, the solution maps directly to the specifying equation with minimal effort:

\[
jacobiJ,K (matrix a,b, scalar \delta) = (a(j+1,k)+a(j-1,k)+a(j,k+1)+a(j,k1))/4((b(j,k) \* \delta^2)/4) when not((j=1 or size(a)=j) or(k=1 or size(a)=k)) else a(j,k)
\]

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. Further, 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 \( j \) isn’t the first or last row and \( k \) isn’t the first or last column.

A matrix transpose function is an example of Case 3:

\[
transpose(i,j)(matrix m) = m(j,i)
\]
as is the matrix multiplication from an earlier section:

\[
mat_mul(i,j) (matrix m1,matrix m2) = Sum(m1(i,all) * m2(all,j))
\]

This definition declares that the function’s \( i,j \)th element is 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 the SequenceL definition typically requires recursion. The Quicksort specification is as follows:

1. choose a pivot from the unsorted list;
2. create a list that contains, first, a sort of all items < pivot, then the pivot, and, finally, the sort of all items > pivot; and
3. repeat actions 1 and 2 until a list of one item or a scalar is the Quicksort argument.

In SequenceL Quicksort is:

\[
Quick(any X) = X when scalar(X) else appends(quick( X when X < X(1)), X(1), quick(X when X > X(1)))
\]

**Principle 5: Transcribing equations**

In general, a programmer can clearly and easily transcribe any effective formal definition into an executable SequenceL definition. Evidence for this principle also illustrates that SequenceL wasn’t defined with a particular application domain in mind.

**SEQUENCEL APPLICATIONS**

Initial efforts to develop formal definitions and natural-language requirements have focused on equation-based specifications like the Jacobi example. We’ve explored beyond numerical computations to include equations from set-builder notation, denotational semantics, and others. In all cases we’ve explored, a programmer can simply translate the equation directly into SequenceL using the principles. Examples from set-builder notation include the following:

\[
in(scalar X, any S) = or(X = S)
\]
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. The any in the function signature indicates that there are no restrictions on nesting for parameter S. These operators follow the definitions of the universal and existential operators. For example, \( \forall X p(X) \equiv p(x_1) \& p(x_2) \& \ldots p(x_n) \) assuming a finite domain for X:

\[
\text{subset}(sets S1,S2) = \\
\text{and}(\text{in}(S1,S2))
\]

\[
\text{set_equality}(sets S1,S2) = \\
\text{subset}(S1,S2) \& \text{subset}(S2,S1)
\]

Just as the definitions above intuitively link to set theory, the following look similar to their set-builder equivalents.

\[
S1 \cap S2 = \{ x \mid x \in S1 \& x \in S2 \}
\]

\[
\text{intersection}(sets S1,S2) = \\
\{ x \mid x \in S1 \& x \in S2 \}
\]

\[
S1 \cup S2 = \{ x \mid x \in S1 \text{ OR } x \in S2 \}
\]

\[
\text{union}(sets S1,S2) = \\
\{ x \mid x \in S1 \text{ or } x \in S2 \}
\]

\[
S1 - S2 = \{ x \mid x \in S1 \text{ & x} \notin S2 \}
\]

\[
\text{difference}(sets S1,S2) = \\
\{ x \mid x \in S1 \text{ and not } (x \in S2) \}
\]

\[
S1 \times S2 = \{ \langle x,y \rangle \mid x \in S1 \& y \in S2 \}
\]

\[
\text{cartesian_product}(sets S1,S2) = \\
\{ [X,Y] \mid \text{in}(X,S1) \& \text{in}(Y,S2) \}
\]

Consider this denotational semantic equation for the while loop:

\[
\Sigma \langle \text{while C do S od} \rangle \sigma = \\
\Sigma \langle \text{while C do S od} \rangle \cdot \Sigma \langle S \rangle \sigma \text{ when } \zeta \langle C \rangle \sigma \sigma \text{ else } \sigma
\]

Here's the SequenceL version:

\[
\text{Sigma(any [while,C,do,S,od], Matrix State) = } \\
\text{Sigma([while,C,do,S,od], Sigma(S,State)) when Gamma(C,State) else State}
\]

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 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 also transcribed definitions of tuple relational calculus from database theory into SequenceL definitions, and these SequenceL definitions are executable.

We only recently initiated studies of the relationship between natural-language specifications and SequenceL. We focus on requirements for abort systems we’re developing in collaboration with NASA Johnson Space Center’s guidance, navigation, and control engineers.

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 predefined priorities. These sites are targeted landing options available in the event of an abort during the ascent, or launch, phase. As input, SortTAL sites takes four subscripted vectors, each representing a highly ranked abort landing site. Each vector identifies 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, followed by the natural-language requirements:

\[
\text{SORT_TAL—component 1}
\]

\[
\text{preferred(vector B, A, scalar Prime)} = \\
B \text{ when } \& (\text{B:subscript} = 4, B:available, B:valid) \text{ else } \\
[] \text{ when } \& (A:subscript = 4, A:available, A:valid) \text{ else } \\
B \text{ when } \& (\text{B:location} = \text{Prime}, B:available, B:valid) \text{ else } \\
[] \text{ when } \& (A:location = \text{Prime}, A:available, A:valid) \text{ else } \\
B \text{ when } \& (B:location < \text{Prime}, B:available, B:valid) \text{ else } \\
A \text{ when } \& (B:throttle < A:throttle, B:valid) \text{ else } \\
B \text{ when } \& (B:throttle < A:throttle, B:available) \text{ else } \\
B \text{ when } A:subscript > B:subscript
\]

Nature language. Preference between landing sites A and B is determined using the first rule that applies on the following list:

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; 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're creating natural-language specifications from SequenceL definitions to identify patterns as well as a restricted grammar for producing SequenceL from natural-language specifications, and vice versa.

Although SequenceL is effective on a wide range of applications, we've done our more industrial-strength work at NASA. From this NASA work, we have consistent and strong anecdotal evidence of SequenceL's transparency. We've applied SequenceL to three major NASA guidance, navigation, and control applications. The abort-system prototypes for the space shuttle and the new Orion systems are developed in SequenceL. Most recently, we completed a system that predicts a space vehicle's near-future trajectory 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 East Coast or overseas landing sites.

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. In other words, once the primitive operations and the CSP-NT exist, we can implement function argument instantiation, structure subscripting, function body evaluation, and function reference handling. These semantics illustrate clear differences between SequenceL and other languages, including Haskell, Function Programming, A Programming Language, Nested Data-Parallel Language, and Gamma. Earlier work reported these differences.

References


Daniel E. Cooke is the Paul Whitfield Horn Professor of the Computer Science Department at Texas Tech University. He is currently located at Texas Tech’s graduate site in Abilene. Cooke received a PhD from the University of Texas at Arlington. He is a member of the IEEE Computer Society and a senior member of the IEEE. Contact him at dcooke@coe.ttu.edu.

J. Nelson Rushton is an associate professor in the Computer Science Department at Texas Tech University. His research interests include functional programming and knowledge representation. Rushton received a PhD in mathematics from the University of Georgia. Contact him at nelson.rushton@gmail.com.