Reusable Ada Libraries Supporting Infinite Data Structures

Arthur G. Duncan
Rensselaer Polytechnic Institute
Troy, NY 12180
1-518-377-8797
duncan@cs.rpi.edu

1. ABSTRACT

In this paper we describe a method, based on lazy evaluation, for creating infinite data structures in Ada. We illustrate some potential applications of infinite data structures and describe three different implementation approaches.

1.1 Keywords

Ada, functional programming, lazy evaluation, infinite data structures.

2. INTRODUCTION

In this paper, we show how one can employ some elegant methods of functional programming within the Ada programming language. In particular we show how to create libraries of infinite data structures and higher order functions. We illustrate how these data structures might be used in a variety of applications and discuss three different approaches to their implementation.

The remainder of the paper is organized as follows:

In Section 3 we introduce lazy evaluation, using some familiar examples to illustrate its usefulness.

In Section 4 we describe some of the difficulties of defining lazy evaluation in a language like Ada, then show how to overcome them with a "one size fits all" definition of infinite data structures in Ada.

In Section 5 we illustrate the use of infinite sequences in such diverse areas as Numerical Analysis, Statistics, and Distributed Interactive Simulation.

In Section 6 we discuss two alternative approaches to achieving infinite data structures that still preserve the spirit of Ada's strong typing.

In Section 7 we discuss some limitations Ada places on functional programming and suggest some ways to work around them.

In Section 8 we mention some related work, such as the Ada Generic Library, the C++ Standard Template Library, and the Kawa and Pizza systems.

Finally, in Section 9 we offer some conclusions and suggest some future lines of investigation.

3. AN OVERVIEW OF LAZY EVALUATION AND INFINITE DATA STRUCTURES

Lazy evaluation, which has been used in the functional programming community for the past two decades [5, 6], provides elegant solutions to a number of problems, such as the creation and manipulation of infinite data structures. Several functional languages, including Haskell [7] and Miranda [16], even use lazy evaluation as the usual mode of operation.

3.1 Lazy Expressions

Lazy evaluation consists of delaying the actual evaluation of an expression until the expression is actually needed in another computation. The general technique is to replace the actual computation of an expression with a "promise" to compute the expression when necessary. This promise consists of a function (actually, a closure) and the appropriate arguments necessary for performing the given computation.

Virtually all computer languages contain some lazy constructs. In Ada, for example, the *and then* and *or else* constructs are lazy in their first argument.

To see this, consider the expression

```
x /= 0 and then y/x < 5
```

This expression is lazy in that *Y/X* is not evaluated unless we know that *X* is non-zero. The laziness of the expression guards against divide-by-zero errors.

3.2 Lazy Functions

In the functional programming community, the term *lazy* generally applies to functions, meaning that a given function does not evaluate its arguments.

To illustrate, consider the *cons* function found in Lisp [14] and in virtually all functional languages. The function call

```
(cons e L)
```

adjoins the element *e* to the front of the linked list *L*. The conventional evaluation order would be to evaluate *e* and *L*

first, then apply *cons* to the results. However *cons* need not, and perhaps should not, evaluate its arguments [5]. It could merely create a new list cell containing a pointer to the element *e* and a pointer to the list *L*, without having to know what *e* and *L* evaluate to.

Again, we can think of a lazy expressions as a promise to compute a given value, if we need it.

3.3 The Need for Eager Expressions

An "eager" expression is one that is evaluated at once. An eager function, therefore, is one that evaluates its arguments before it is applied.

Even lazy languages must, of necessity, contain some eager constructs, otherwise no work would get done. (If every function were lazy, we would be left with a set of empty promises.)

Some constructs are eager in some arguments and lazy in others. For example, an *and then* or an *or else* construct will evaluate its first argument eagerly, only evaluating its second argument if it is needed.

In a lazy functional language, the general rule is that every construct that is not required to be eager is considered lazy. Expressions that are necessarily eager include:

- the condition in an *if* or *while* expression,
- the first expression in an *and* expression or an *or* expression (for most functional languages *and* and *or* correspond to Ada's *and then* and *or else* constructs),
- the control value of a *case* expression,
- all arithmetic operations, and
- all component selectors, such as the head and tail of a list.

3.4 Infinite Data Structures

One important use of lazy evaluation has been to create "infinite" data structures, such as the sequence of all natural numbers or the sequence of all prime numbers.

Using a notation based on the language Miranda, we might define the sequence of all natural numbers as

```
from n = n : from (n+1)
naturals = from 0
```

In our Miranda-like notation, the symbol ":" indicates the operation of *cons*-ing an element onto the head of a list.

Since we humans are finite creatures, any real application of infinite data structures will only involve looking at some finite number of actual values within any particular data structure.

For example, we may want to compute

- the sequence of all prime numbers less than 1000,
- the sequence of approximations to the square root of 2, terminating when the relative error is less than some prescribed quantity ε, or
- a statistical sample corresponding to a particular time interval within a sequence of sensor input readings from a potentially infinite forward time span.

The trick is never to deal with the entire data structure at once, but only those parts that we need. To the user, however, the particular object should be indistinguishable from an actual infinite data structure.

Consider a simple example — extracting the third natural number. We can define the function third as

```
third L = hd (tl (tl L))
```

where *hd* returns the first element of a sequence and *tl* returns the sequence of all elements excluding the first. The third natural number would then be given by

```
third naturals
```

Tracing the computation of *third naturals* gives us

```
1. third naturals => third (from 0) -- by definition
2. => hd (tl (tl (from 0))) -- again, by definition
3. => hd (tl (tl (0 : from 1) -- since tl is eager
4. => hd (tl (from 1))
5. => hd (tl (1 : from 2)) -- again since tl is eager
6. => hd (from 2)
7. => hd (2 : from 3) -- since hd is eager
8. => 2
```

We never explicitly computed the rest of the sequence, represented by *from 3*, although it remained available in case we needed it.

A useful auxiliary function is *take n L*, which returns the first *n* elements of a list *L*.

```
take n L = if n = 0 then [ ] else hd L : take (n - 1) (tl L)
(* here [ ] denotes the empty list *)
```

Thus *take 5* naturals would yield the list *[0, 1, 2, 3, 4]*.

Many functional languages allow us to define functions in terms of constructor patterns on their arguments, which often provides a more readable alternative to using *case* statements or *if-elsif-else* statements. Using pattern notation, *take* can be defined as

```
take 0 L = [ ]
take n L = hd L : take (n-1) L
```

A more interesting example is the set of all prime numbers, which we can define using the well-known Sieve of Eratosthenes. In a functional language, the definition is quite elegant.

```
filterOut p (n : L) = if n mod p = 0 then filterOut p L
                      else n : filterOut p L
sieve p : L = p : sieve (filterOut p L)
primes = sieve (from 2)
```

Notice how *sieve* describes an infinite sequence in terms of a countably infinite set of infinite sequences.

3.5 Functions as First Class Citizens

The distinguishing feature of functional languages is that functions are first class citizens, which means we can treat a

function as an ordinary data object. We can pass functions as parameters and return functions as results, which enables us to define higher order functions, that is, functions that take functions as arguments and return functions as values.

One useful higher order function is *map*, which applies a function across an entire list of arguments. For example, if we define *square* by

```
square n = n * n
```

the expression

```
take 5 (map square naturals)
```

would yield the list *[0, 1, 4, 9, 16]*.

We can define *map* by

```
map f [ ] = [ ]
map f (x : L) = f x : (map f L)
```

In his excellent paper, "Why Functional Programming Matters," John Hughes [8] shows how functional programming provides a powerful mechanism for decomposing programs into modules and gluing them together in easily understood ways. In the above examples, we have constructed programs from infinite data structures, using lazy evaluation and higher order functions as the glue.

One can describe many problems, from such diverse areas as Numerical Analysis, Artificial Intelligence, and Distributed Simulations, effectively in terms of operations on infinite sequences. We can then use higher order functions to tie these sequences together. Lazy evaluation provides the key to producing transparent descriptions of solutions to such problems while still achieving reasonably efficient implementations.

3.6 Avoiding Needless Computation

Most implementations of lazy evaluation are "lazy" in two senses — (1) they will not compute any result before it is needed and (2) they will only compute any particular result once, saving the result for future use, if necessary.

This second sense of laziness is usually implemented by defining a promise as a structure with two components:

- a function to compute the result, and
- a holder in which to keep a reference to the computed result.

4. DEFINING LAZY EVALUATION IN ADA

Whenever we wish to introduce a new paradigm into an existing language, we should ask three questions:

- Can we do it at all?
- Can we do it in a way that preserves the advantages and general flavor of the original language?
- Can we do it efficiently?

With regard to introducing lazy evaluation and infinite data structures into the Ada language, we will answer the first question in this section and address the second and third questions in Section 6.

4.1 Implementing Lazy Evaluation in an Eager Language

To implement lazy evaluation in a language where each function evaluates its arguments eagerly, we need a mechanism for delaying the evaluation of a function (or its arguments) until the result is needed.

This question has been addressed at length in the functional programming community. Eager functional languages, such as Scheme [9] and ML [13], simulate lazy evaluation by replacing a function call with a "promise" (i.e., the function code, its list of arguments, and its definition environment), which is then evaluated when necessary.

Scheme, for example, provides the functions *delay*, to create a promise and *force*, to evaluate a promise. In Scheme terminology, the function *delay* is more or less equivalent to the definition

```
(define (delay expr) (lambda () expr))
```

while *force* is more or less equivalent to the definition

```
(define (force delayed-expr) (delayed-expr))
```

For those unfamiliar with Lisp-like notation, the expression

```
(lambda () expr)
```

represents an anonymous function of zero variables that evaluates to the value of *expr*, while

```
(delayed-expr)
```

represents the process of applying the function represented by *delayed-expression* to an empty argument list. While we have called the *lambda* expression above a *function*, it is actually a *closure*, i.e., a function definition together with the variable bindings that were present at the time the function was defined.

We will now show how to achieve delayed evaluation in Ada,

4.2 *Sexpr*: A Universal Type

For our first stab at creating an infinite sequence package, we will adopt a "one size fits all" approach, similar to that found in a weakly typed language like Lisp or Scheme. We will create the type *Sexpr* (or "symbolic expression"), which will encompass all the data types we will use — all scalar types, sequences, and function types. This allows us, for example, to represent a function of any number of variables as a *Sexpr* and to represent the list of actual parameters to the function as another *Sexpr*.

4.2.1 Elementary Values

We can create *Sexpr*(s) corresponding to elementary values, such as booleans, integers, and floats. For example,

```
function New_Int(I : Integer) return Sexpr;
```

returns a *Sexpr* corresponding to an integer value. We also have functions to extract an elementary value from a *Sexpr*. For example,

```
function Int(S : Sexpr) return Integer;
```

extracts the integer value contained in a *Sexpr* created with *New_Int*.

4.2.2 Building Finite Sequences

Sequences can be either finite or infinite. To build finite sequences, we start with the empty sequence and create larger sequences by tacking *Sexpr*(s) onto the fronts of existing sequences. Thus, our basic sequence builders are

```
function Nil return Sexpr;
function Cons(S1, S2 : Sexpr) return Sexpr;
```

Nil simply returns a constant empty sequence, while *Cons* adds *S1* to the front of *S2*. In all of the examples in this paper, we will assume *S2* to be a (finite or infinite) sequence.

As an example, the list *[1, 2, 3]* would be represented as

```
Cons(New_Int(1), Cons(New_Int(2), Cons(New_Int(3), Nil)))
```

We can access sequences by using the functions

```
function Head(S : Sexpr) return Sexpr;
function Tail(S : Sexpr) return Sexpr;
procedure Set_Head(S : Sexpr; To_Be : Sexpr);
procedure Set_Tail(S : Sexpr; To_Be : Sexpr)
```

Head returns the front of a sequence, while *Tail* returns the sequence consisting of the remaining elements in order. *Set_Head* replaces the first element, while *Set_Tail* replaces the remainder sequence. In the case of the sequence *Nil*, the function *Tail* returns *Nil*, while the other functions raise an exception. All the above routines will return an exception if *S* is not a sequence.

We also note that the parameter *S* to *Set_Head* and *Set_Tail* is an *in* parameter rather than an *in out* parameter. We accomplish side effects by altering values accessed via pointers. We avoid *in out* parameters whenever possible, so that we can use any expression in an appropriate context, no matter how that expressions was obtained.

4.2.3 A Notational Convenience

To simplify our notation, we have overloaded *Cons* and the common arithmetic operators to work on *Sexpr*(s) and on mixed arguments of elementary values and *Sexpr*(s). Thus, we adjoin the integer *3* to the front of a list *L* with the call

```
Cons(3, L)
```

rather than the more lengthy

```
Cons(New_Int(3), L)
```

Also, if *X* and *Y* represent two *Sexpr*(s) corresponding to integer values, we can form the *Sexpr* corresponding to their some by writing the expression

```
X + Y
```

rather than the more tedious

```
New_Int(Int(X) + Int(Y))
```

Finally, we have introduced implicit type coercions into our arithmetic operators, so that an arithmetic operation on a mixture of integer and real *Sexpr*(s) returns a real *Sexpr*.

While implicit type coercion goes against the Ada strong typing philosophy, it fits in well with the weakly typed Lisp and Scheme approach. Our more Ada-like versions of lazy evaluation, discussed in Section 6, will not allow implicit type conversions.

4.2.4 Making Promises (or Functions as Sexprs)

Just as we use Nil as the starting point for finite sequences, we build infinite sequences using a *promise* (called an *application* in our system) as a starting point. We define the type *Sexpr_Function* to be an access to a function from *Sexpr*(s) to *Sexpr*(s). We then define the constructor

```
function New_Function(F : Sexpr_Function) return Sexpr;
```

which allows us to treat functions as data objects. Finally, we define a promise as

```
function New_Application(F : Sexpr;
                         S : Sexpr_Function) return Sexpr;
```

When we build an application object, the second argument will always be a sequence of elements, corresponding to the list of actual parameters.

As a complement to the application objects, we create the function *Forced*, specified as

```
function Forced(S : Sexpr) return Sexpr;
```

which can be used to force evaluation of a function application. For any *Sexpr* that is not an application, *Forced* will return the *Sexpr* itself.

Eager functions, like *Head*, *Tail*, and the arithmetic functions, will call *Forced* repeatedly on their arguments until they receive an answer that is not an application.

4.2.4.1 Defining Auxiliary Functions

It is often convenient to define a function on elementary values, then implement it using an auxiliary *Sexpr* function. For example, we could define the sequence of integers *n*, *n+1*, *n+2*, etc. by defining the following two functions:

```
function From_N(N : Integer) return Sexpr;
function From_N_Aux(S : Sexpr) return Sexpr;
```

The definition of *From_N_Aux* can be made private, if we wish to hide the implementation details of *From_N*.

4.2.4.2 Implementing an Infinite Sequence

The implementations of *From_N* and *From_N_Aux* illustrate a general pattern of infinite sequence creation; therefore we will look at their implementations in detail.

We can define the function *From_N* as

```
function From_N(N : Integer) return Sexpr is
begin
    return From_N_Aux(New_Int(N));
end From_N;
```

and *From_N_Aux* as

```
function From_N_Aux(S : Sexpr) return Sexpr is
    H : Sexpr := Head(S);
begin
```

```
        return Cons(H, New_Application(From_N_Aux'Access,
                                       Cons(H + 1, Nil)));
    end From_N_Aux;
```

The *New_Application* function delays the evaluation of the rest of the sequence. We could not simply write

```
function From_N_Aux(S : Sexpr) return Sexpr is
    H : Sexpr := Head(S);
begin
    return Cons(H, From_Aux(Cons(H + 1, Nil)));
end From_N_Aux;
```

since this would result in an infinite recursion.

4.3 Operations on Infinite Sequences

Infinite sequences constitute building blocks that allow us to perform interesting operations on sequences. We will look at two particular kinds of operations — (1) transformations on the sequences themselves, such as filtering out particular elements of a sequence, and (2) higher order functions on sequences, such as mapping a function across all elements of a sequence.

4.3.1 Transformations on Sequences

Consider the sequence *2, 3, ...*, given by

```
From_2 : Sexpr := From_N(2); -- the sequence 2, 3, ...
```

We could define the set of all prime numbers as:

```
Primes : Sexpr := Sieve(From_2);
```

Where Sieve applies the Sieve of Eratosthees to the sequence *2, 3, ...* .

If the function *Take(N, S)* computes the finite sequence consisting of the first N items in the sequence *S*, then we can represent the first 100 prime numbers, in order, by the expression:

```
First_100_Primes := Take(100, Primes);
```

The *Sieve* itself can be defined as

```
function Sieve(S : Sexpr) return Sexpr is
    L : Sexpr := Head(S);
    H : Sexpr := Head(L);
    F : Sexpr := New_Application(Filter_Out'Access,
                                 Cons(H, Cons(Tail(HT), Nil)));
begin
    return Cons(H, New_Application(Sieve'Access,
                                   Cons(F, Nil)));
end Sieve;
```

and *Filter_Out* is defined by

```
function Filter_Out(S : Sexpr) return Sexpr is
    H : Sexpr   := Head(S);
    N : Integer := Int(H);
    S2 : Sexpr   := Head(Tail(S));
    H2 : Sexpr   := Head(S2);
begin
    if Int(H2) mod Int(H) = 0 then
        return New_Application(Filter_Out'Access,
                               Cons(H, Cons(Tail(S2), Nil)));
    else
        return Cons(H2, New_Application(
                                Filter_Out'Access,
                                Cons(H, Cons(Tail(S2), Nil))));
    end if;
end Filter_Out;
```

The code is somewhat messy, due to the need to pass the parameters as a list. We could clean up the code a bit by defining constructor functions to build the parameter lists.

4.3.2 Higher Order Functions

Higher order functions can be implemented in much the same way as other functions on sequences. For example, we can define *Map* function, discussed earlier, as

```
function Map(S : Sexpr) return Sexpr is
    H : Sexpr := Head(S);
    L : Sexpr := Head(Tail(S));
begin
    if Is_Nil(L) then
        return Nil;
    else
        return Cons(Apply(H, Head(L)),
                    New_Application(Map'Access
                                    Cons(H, Cons(Tail(L), Nil))));
    end if;
end Map;
```

4.4 The Underlying *Sexpr* Data Structure

Sexpr is defined as a record type with a single component, namely, a pointer to a variant record type that encompasses all the varieties of *Sexpr* objects.

The record variants consist of

- a null variant, corresponding to the *Nil* object,
- several *elementary* variants, each of which has a single field containing an elementary value,
- a *function* variant that contains a pointer to a function on *Sexpr*(s),
- a *pair* variant that contains a pointer to the head of a sequence and a pointer to the rest of the sequence (Pairs are the result of *Cons* operations), and
- an *application* variant that contains a function pointer and a list of arguments to the function.

Pairs and applications can both represent sequences.

One could trade some space for time by including an additional field in an *application* variant. This field could hold the computed value, once the application had been *forced*. If we had several pointers to the same application structure active at the same time, this would allow a computed value to be shared by all references to that value.

The Appendix presents some of the implementation code for the *Sexpr* data type.

4.5 The Role of Ada 95

Our implementation of lazy evaluation is based upon the function access types available in Ada 95.

Since Ada 83 did not have function accesses, we could not easily define this kind of mechanism in Ada 83. While we could resort to some subterfuge, such as passing a function's

address as an integer to as machine language routine that called the function, this approach would be highly implementation dependent.

Another approach might be to try to simulate function objects by task objects; however, since all task objects of a given type must perform the same operation, we would need to decide in advance what functions were to be allowed in a given application, then build a task type specifically for that application.

The author encountered a similar problem when defining the pretty printer for ENCORE, a software reengineering system developed at the G.E. Research and Development Center [2]. In that system we had to resort to a large *case* statement in order to invoke the appropriate function at any given point in the pretty printer operation.

5. SOME USES OF INFINITE DATA STRUCTURES

Sequences arise naturally in many applications. These include applications include problems in numerical analysis, statistics, and simulation. In numerical analysis, we often deal with sequences of approximations to a given solution; in statistical problems we deal with sequences of observations; and in simulations we deal with sequences of system states.

5.1 Successive Approximations

One common approach in solving numerical problems is to define a sequence of approximations to a particular value. A simple example is Newton's method for finding the positive square root of a real number. (A good discussion of successive approximations can be found in [8].)

5.1.1 Newton's Method

Newton's method for finding the square root of a number x can be described by the sequence

$$y_0, y_1, y_2, y_3, \ldots$$

where y_0 is some initial approximation to the square root of x and each subsequent y_n can be defined in terms of its predecessor in the sequence:

$$y_n = (y_{n-1} + x/y_{n-1}) / 2.$$

We can use our *Seq* package to implement this sequence, by defining a function that computes the rest of the sequence, given *x* and an initial value for the sequence. We assume here that *S* consists of the list *[x, y]*, containing *x* and the latest approximation *y* to the square root. We can then define our *Sexpr* function as

```
function Approximations(S :Sexpr) return Sexpr is
    X : Sexpr := Head(S);
    Y : Sexpr := Head(Tail(S)));
    New_Y : Sexpr := (X + Y/X)/2.0;
begin
    return Cons(Y,
                New_Application(Approximations'Access,
                                Cons(X, Cons(New_Y,
                                             Nil))));
end Approximations;
```

Using this function, and *2.0* as our initial approximation, we could define the square root of *2* as

```
Sqrt_2 : Sexpr := Approximations(Cons(2.0, Cons(2.0, Nil)));
```

If we compute the first six approximations, using the expression:

```
Take(6, Sqrt_2)
```

We will get the sequence

2.00000, 1.50000, 1.41667, 1.41422, 1.41421, 1.41421

5.1.2 Maclaurin's Series

Every calculus student learns to use Taylor's series to approximate the value of an analytic function near some point. The the special case of Taylor's series, where the known point is zero, is called Maclaurin's series.

Perhaps the simplest example of Maclaurin's series is the expansion of e^x, which can be written as

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \ldots$$

We can define e^x by the following declarations:

```
X : Float := ...; -- Some floating point value
Exp_X_Sequence : Sexpr := Pointwise_Div(Powers(X), Facs);
Exp_X_Series : Sexpr := Sums(Exp_X_Sequence);
```

We can define *Facs* by

```
Facs : Sexpr := Prods(From(1.0));
```

where *From(X : Float)* is defined similarly to the *From* function for integers.

We can define *Sums* as

```
function Sums(S : Sexpr) return Sexpr is
begin
    return Sums_Aux(Cons(Cons(0.0, S), Nil);
end Sums;
```

Where *Sums_Aux* is defined as

```
function Sums_Aux(S : Sexpr) return Sexpr is
    H : Sexpr := Head(S);
begin
    if Is_Nil(Tail(H)) then
        return Head(H);
    else
        return New_Application(Sums_Aux'Access,
                               Cons(Cons(Head(H) + Head(Tail(H)),
                                         Tail(Tail(H)), Nil));
    end if;
end Sums_Aux;
```

The function *Prods* can be defined in a similar way, with addition replaced by multiplication.

Finally, we can define *Pointwise_Div* by

```
function Pointwise_Div(S1, S2 : Sexpr) return Sexpr is
begin
    return Pointwise_Div_Aux(Cons(S1, Cons(S2, Nil)));
end Pointwise_Div_Aux;
```

where *Pointwise_Div_Aux* is defined by

```
function Pointwise_Div_Aux(S : Sexpr) return Sexpr is
    S1 : Sexpr := Head(S);
    X1 : Sexpr := Head(S1);
    T1 : Sexpr := Tail(S1);
    S2 : Sexpr := Head(Tail(S));
    X2 : Sexpr := Head(S2);
    T2 : Sexpr := Tail(S2);
begin
    return Cons(X1/X2,
            New_Application(Pointwise_Div_Aux'Access,
                            Cons(T1, Cons(T2, Nil))));
end Pointwise_Div_Aux;
```

If we want an approximation of e^x within some error ε, we need to define a stopping rule for selecting terms of the series. This might look like

```
Exp_X := Within_Relative_Error(Epsilon, Exp_X_Series);
```

To avoid the possibility of an infinite recursion, in case a given series diverges, we should also have another *Within_Relative_Error* function — one that is guaranteed to terminate after examining the first N elements of a sequence. A typical call to this rule would look like

```
Exp_X := Within_Relative_Error(Epsilon, N, Exp_X_Series);
```

5.1.3 A General Successive Approximations Package

Most successive approximations consist of applying a given incremental approximation function repeatedly to an initial value to obtain a sequence of approximations. We can capture the successive approximation pattern as a higher order function, which we could apply to particular incremental functions and initial values.

In other words, we could define a function

```
function Successive_Approximations(S : Sexpr) return Sexpr;
```

To define this higher order function, we need to pack all the information we need into the argument S. The result should give us the next term in the sequence as well as the information necessary to compute the rest of the sequence. Thus, the argument S will consist of

- a function F, and
- the information to compute our desired results.

The results should consist of

- the next term in the sequence, and a list consisting of
- the function F again, and
- the information necessary to carry out the next iteration.

The easiest way to represent the parameter and the result would be as a list. The size of the lists will vary, depending on the particular application. In the case of Newton's method, the argument list will look like

```
[F, X, Y]
```

where *F* is the function to go from one step to the next in the sequence, *X* is the number whose square root we wish to compute, and *Y* is the current approximation. The result of *F* will look like

```
[Y, F, X, Y1]
```

where *Y1* corresponds to the result *(Y + X/Y) / 2.0*.

The definition of the general *Successive_Approximations* function is straightforward, so we will omit it here.

If we let *Newton_Step* denote the Sexpr function that computes *(Y + X/Y)/2.0*, we can define Newton's method as

```
Successive_Approximations(Cons(Newton_Step,
                               Cons(X, Cons(2.0, Nil))));
```

Each new successive approximations application simply involves defining a new function *F* to go from one approximation to the next.

5.1.4 Adding a Stopping Rule

We are usually not interested in an entire sequence, but simply in the first element that satisfies some criteria. This brings us to the notion of a stopping rule.

Our relative error rule, mentioned previously, can be defined by the following function:

```
function Within_Relative_Error(E : Float; S : Sexpr)
                                                return Float is
    H, T, HT : Sexpr;
    X, Y : Float;
begin
    if Is_Nil(S) then
        raise Constraint_Error;
    else
        H := Head(S);
        X := Real(H);
        T := Tail(S);
        if Is_Nil(T) then
            return X;
        else
            HT := Head(T);
            Y := Real(HT);
            if Y = 0.0 and then abs(X) < E then
                return Y;
            elsif abs((X - Y)/Y) < E then
                return Y;
            else
                return Within_Relative_Error(E, T);
            end if;
        end if;
    end if;
end Within_Relative_Error;
```

As with the successive approximations process, we could define a higher order function called *Stopping_Rule* and supply individual functions to obtain particular stopping rules.

5.1.5 More Operations on Approximation Sequences

Some approximation algorithms converge slowly. This not only wastes computing resources but also can lead to a loss of accuracy (since roundoff errors tend to accumulate with the number of operations performed). Consequently, a number of sequence transformations have been developed in order to speed up the convergence of successive

approximation algorithms. These transformations can be viewed simply as functions from one sequence to another and, thus, can be defined using our lazy evaluation mechanism.

5.2 A Note on Using Generics

Certain patterns arise repeatedly in our applications. For example, in converting a floating point function *F* to a *Sexpr* function *F_S*, we often encounter code like:

```
function F_S(S : Sexpr) return Sexpr is
begin
    return New_Real(F(Real(Head(S))));
end F_S;
```

We can save some keystrokes (and also improve readability and reliability) by defining generic functions that embody a number of these patterns.

The approximation steps and the stopping rules could also benefit from this sort of treatment.

5.3 Statistical Applications

Statistical applications involve gathering sample data and analyzing the data in terms of some probabilistic model that the data are expected to satisfy. The samples themselves can appear in many forms, such as

- quality measurements on a randomly selected sample of machine parts from an assembly line,
- measurements of the effects of a new drug versus a placebo on patients with a particular illness, or
- an ongoing collection over time of closing share prices of a list of selected stocks on the New York Stock Exchange.

We can represent all the above data as sets of values, which we can represent, in turn, as sequences.

While the samples themselves may come from diverse sources, we can handle them in similar ways. Routines to analyze samples in terms of means, variances, correlation coefficients, etc., should not be concerned with whether the sample was typed in at a terminal, read from a file, or gathered by some set of sensors placed at strategic points in some area. All samples should appear to the statistical routines as instances of the same data type.

The use of infinite data structures allows us to view each sample as a (potentially infinite) sequence.

5.3.1 Computing the Mean and Standard Deviation

As our focus here is on dealing with different kinds of samples, rather than computing particular statistics, we will simply show how to compute the mean and standard deviation using our potentially infinite data structures.

We can compute the mean and standard deviation of a sample by keeping a running sum of data values and a running sum of the squares of the values. At any point in the process we could compute the mean and standard deviation (up to that point).

Our sample and running sums might look like:

```
Sample : Sexpr := ... definition of the sample sequence ...
Sample_Arg : Sexpr := Cons(Sample, Nil);
Sample_Sums : Sexpr := Sums(Sample_Arg);
Sample_Sums_Of_Squares :=
                              Sums_Of_Squares(Sample_Arg);
```

Our running means and standard deviations might be defined by

```
Running_Mean : Sexpr := Means(Sample_Arg);
Running_Variance : Sexpr := Variances(Sample_Arg);
Running_Standard_Deviation : Sexpr
    := Map(Cons(Sqrt_Fcn, Cons(Running_Variance, Nil)));
```

Each individual function listed above is fairly easy to define. The running standard deviation is also straightforward, once the running variance has been defined. The object *Sqrt_Fcn* is a *Sexpr*-ized version of the *Sqrt* function.

5.3.2 Gathering Samples by Hand

The usual way to gather a statistical sample is to have one or more researchers gather the relevant data and then enter the data into files. We then classify, filter, and merge these files to form a sample.

In order to describe such a sample as a sequence, we would use our lazy evaluation mechanism, where the function to create the rest of a sequence would perform the tasks of reading from one or more files then classifying, filtering, and merging the results.

As a simple example, suppose the data values consist of real numbers held in a number of files and that all the values will be used to form our sample. We could simply gather the file names into a sequence, then define a function that reads successively from each file, until all the files have been read. Let us make the following assumptions regarding the sample values:

- The set of files can be represented as a *Sexpr*, called *Files*, which consists of a sequence of symbols.
- The *Sexpr* function *Next_Element* reads the next element from the variable *Files* and prepares the variable *Files* to read the next element to be read.
- There is a function *End_of_Files* that indicates that the end of file has been reached for each of the files in the variable *Files*.

We can then define the sample as

```
Sample : Sexpr := Remaining_Elements(Cons(Files, Nil));
```

where the function *Remaining_Elements* is defined as

```
function Remaining_Elements(S : Sexpr) return Sexpr is
    X : Sexpr;
    Files : Sexpr := Head(S);
begin
    if End_Of_FIles(Files) then
        return Nil;
    else
        X : Sexpr := Next_Element(Files);
        return Cons(X, New_Application(
```

```
                    Remaining_Elements'Access,
                    Cons(Files, Nil)));
        end if;
    end Remaining_Elements;
```

We employ the local variable *X* to guarantee that the variable *Files* is properly adjusted before the recursive call to *Remaining_Elements*.

5.3.3 Using a Random Sample Generator

In devising experiments and in testing statistical procedures, one often employs random samples based upon a particular probability distribution, such as a uniform distribution, a Gaussian distribution, or a Poisson distribution. This kind of sample is fairly easy to describe as a lazy sequence. Suppose, for example, that

```
function Uniform(A, B : Float) return Float;
```

defines a pseudo-random number generator representing a uniform distribution on the interval from *A to B.* We could then define an infinite sequence of values from this distribution by

```
function Uniform(S : Sexpr) return Sexpr is
    A : Float := Real(Head(S));
    B : Float := Real(Head(Tail(S));
begin
    return Cons(New_Real(Uniform(A, B)),
                New_Application(Uniform'Access, S));
end Uniform;
```

An infinite sample from the interval 0 to 5, say, would be represented as

```
Sample_0_5 : Sexpr := Uniform(Cons(0.0, Cons(5.0, Nil)));
```

while a sample of, say, 1000 such numbers could be represented as

```
Sample_1000_From_0_5 : Sexpr := Take(1000, Sample_0_5);
```

5.3.4 Collecting Samples from External Sources

We can employ the same technique in analyzing samples gathered automatically from sampling devices, such as temperature sensors, flow meters, and electrical measuring devices. This is similar to reading files, except that the data are continuously fed into the system.

5.4 Control Applications and Simulations

Control applications are similar to statistical applications in that we gather data then perform some actions based on the data. The data gathering task is similar to collecting a continuous sample from an external source. Some differences are that

- rather than merely gathering statistics, we use the data to decide what actions to perform next, and
- the system will involve feedback, i.e., our actions will affect future inputs.

We can view the data values themselves as state information and the actions as state transformations.

We can define simulation by employing a higher order function called *State_Transformation*. A particular simulation would look like:

```
State_Transformation(Transform, State_Seq, Input_Seq);
```

Here, the *Transform* function and *Input_Seq* would be given, while the *State_Seq* would be built from an initial state using the Transform function and the *Input_Seq*.

5.4.1 Simulating a Factory Floor

In factory automation applications, we can choose to simulate an entire factory floor or to simulate particular aspects, such as the operation of a particular assembly line robot. In either case the data will appear as an infinite sequence of states.

Since we may view the data from the actual operation of a factory floor as an infinite sequence of states, we could extend a factory floor simulation in a straightforward way to obtain a system that actually controls the factory floor.

5.4.2 Prediction/Evasion Problems

Prediction/evasion problems occur in warfare. A good example is in anti-submarine warfare. Given the submarine's position at a given point in time, the destroyer will want to predict the submarine's position after some time interval, say, the time between launching an anti-submarine missile and the time the missile arrives at the target. The submarine, of course, will try to maneuver in such a way as to make this kind of prediction difficult.

In a prediction/evasion problem, we assume we know the past positions of the evader at particular uniform discrete time points in the past:

$$..., x_{-n}, ..., x_{-2}, x_{-1}, x_0$$

Our predictions will be represented by the sequence

$$x_1, x_2, ..., x_n, ...$$

corresponding to discrete time points in the future.

We can represent the first sequence, the observations, as an infinite sequence similar to that formed by gathering a sample from external sources. The second sequence, the predictions, will be generated by the prediction algorithm.

Note that the prediction sequence will change with each new observation. However, we will only need to examine a few points in any given prediction sequence.

Again, the object of the game, from the destroyer's point of view, is to predict the position of the submarine at some particular future time point N within sufficient accuracy to guarantee that the missile will sink the submarine.

We can still apply our general simulation pattern to this problem. We need to modify our general simulation pattern to cover sequences that extend infinitely in two directions, starting from some point in the middle.

5.4.3 Distributed Interactive Simulation

Finally, Distributed Interactive Simulation (DIS) has become a popular topic in strategic and tactical planning. In this type of simulation our inputs can come from a number

of diverse sources — from actual units on the ground, in the air, or at sea; from flight simulators and other similar devices; or from planners in a war games room.

We can unify our view of the inputs by viewing each source of inputs as a sequence of values and merging them to obtain a sequence of states. The simulation can then be considered as a sequence of transformations on the state sequence.

Similarly, we could present specific views of the simulation by applying projection functions to the state sequence.

6. ALTERNATIVE IMPLEMENTATION STRATEGIES

The definition of infinite sequences used so far provides a straightforward weakly typed approach to defining infinite sequences. This approach, patterned on languages like Lisp and Scheme, is particularly useful for prototyping.

We have glossed over a number of issues. First of all, we haven't addressed the problem of reclaiming storage and preventing memory leaks. This was done intentionally to avoid complicating the definitions. One could easily make *Sexpr* a controlled type and add a reference counting mechanism in order to avoid excessive memory leaks. Two references for this technique are Kempe's Tri-Ada 94 paper [10] and English's excellent introductory book on Ada [3].

A more serious matter is that we have effectively thrown away Ada's strong typing mechanism, at least as far as sequences are concerned. Since strong typing is a key feature of Ada, we would like to find a way to obtain infinite data structures without losing the advantages of strong typing.

In other words, can we provide an infinite sequence package for Ada that still has the "look and feel" of Ada?

Two feasible approaches are

- to define a generic lazy evaluation package that can be instantiated with any element type, or
- to define a lazy evaluation package that works with an abstract tagged type, then use Ada's inheritance mechanism to apply the package to any particular element type.

6.1 A Generic Lazy Evaluation Package

A generic lazy evaluation package builds sequences of a generic type *Element*. The actual functions are similar to those of the original *Seq* package; thus, the generic lazy evaluation package contain the functions *Head*, *Tail*, *Cons*, etc.

6.1.1 Specifying the Generic Sequence Package

A brief sketch of the package specification would look like:

```
generic
    type Element is private;
package Generic_Seqs is
    type Seq is private;

    -- General Operations
    function Nil      return Seq;
    function Cons(E : Element; S : Seq) return Seq;
    function Head(S : Seq) return Element;
    function Tail(S : Seq) return Seq;
    ...

    -- The Generate, Map, and Reduce Operators
    type Op0 is access function return Element;
    function Generate(Op : Op0) return Seq;
    type Op1 is access function(E : Element) return Element;
    function Map   (Op : Op1; Arg : Seq) return Seq;
    type Op2 is access function(E1, E2 : Element)
                                          return Element;
    function Reduce(Op : Op2; Init : Element; Arg : Seq)
                                          return Element;

    -- Applications
    type Seq_Op0 is access function return Seq;
    type Seq_Op1 is access function(S : Seq) return Seq;
    type Seq_Op2 is access function(S1, S2 : Seq) return Seq;
    type Seq_Element_Op1 is
        access function(S : Seq; E : Element) return Seq;
    type Seq_Element_Op2 is
        access function(S : Seq; E1, E2 : Element) return Seq;

    function New_Application(Op : Seq_Op0) return Seq;
    function New_Application(Op : Seq_Op1; S : Seq) return Seq;
    function New_Application(Op : Seq_Op2;
                                  S1, S2 : Seq) return Seq;
    function New_Application(Op : Seq_Element_Op1; S : Seq;
                                  E : Element) return Seq;
    function New_Application(Op : Seq_Element_Op2; S : Seq;
                                  E1, E2 : Element) return Seq;
    function Delayed(S : Seq) return Seq;
    function Forced(S : Seq) return Seq;
    ...
private
    ...
end Generic_Seqs;
```

6.1.2 Applying the Generic Sequence Package

This new version of the sequence package allows us to take advantage of Ada's strong typing. For example, if we want to define the sequence of integers 0, 1, 2, ..., etc., we can instantiate an integer sequence package by the declaration

```
package Integer_Seqs is new Generic_Seqs(Integer);
```

We can then declare the sequence of non-negative integers by a declaration like:

```
Natural_Numbers : Integer_Seqs.Seq := From_N(0);
```

where From_N is defined in some package of sequence builders.

We could also define new sequences, such as the sequence of all squares of natural numbers in terms of our original sequence.

```
Natural_Squares : Integer_Seqs.Seq
    := Map(Square'Access, Natural_Numbers);
```

6.1.3 Applications and Higher Order Functions

As we no longer have a *Sexpr* type that encompasses all

possible data types under consideration, the applications and higher order functions will be quite different from those in the weakly-typed *Seq* package. We can no longer make do with a single kind of function that maps from *Sexpr*(s) to *Sexpr*(s), rather, we must specify explicitly the number and types of arguments used in each application.

In this implementation, we have chosen to deal with the most common kinds of functions, namely,

- nullary, unary, and binary functions on elements,
- nullary, unary, and binary functions on sequences, and
- functions involving a sequence and either one or two elements.

6.1.4 Advantages and Disadvantages of the Generic Seq Package

The main advantages of the generic sequence approach are that we can maintain strong typing and that we can implement the package efficiently (at least in the case where all sequences are finite). This approach is similar to that taken in the Ada Generic Library [11], with lazy evaluation thrown in. (In fairness, we should point out that the AGL provides more low-level implementation options than we do and is more concerned with efficiency that we are. One could, however, implement the Generic Sequence Package by using the AGL approach.)

One disadvantage of the generic approach is that we are limited to just the sequence data type. We cannot, for example, build up trees or graphs containing the element type. For this, we would need to define a generic tree package or a generic graph package. In the generic case we are trading flexibility for type security.

6.2 Lazy Evaluation Using Inheritance

Another possibility is to define the sequence types in terms of tagged types and inheritance. Here we define an abstract type called *Sexpr*, then derive our sequence and application types from the abstract *Sexpr* type.

Ideally, we should retain the advantages of Ada strong typing while still enjoying the ability to create more complex data structures than just sequences of elements.

6.2.1 An Overview of the Inheritance Hierarchy

A brief sketch of the type hierarchy is

```
Sexpr <abstract> [predicates, forced, delayed]
    Atom <abstract> [elementary ops]
        ... individual element types ...
    Fcn <abstract> [apply]
        Nullary_Fcn
        Unary_Fcn
        Binary_Fcn
        Seq_Fcn
    Seq <abstract> [cons, head, tail, set_head, set_tail, is_null]
        Pair
        App
```

The hierarchy is indicated by indentation, with abstract types noted. The square brackets contain the names of inheritable functions associated with each type in the hierarchy.

The hierarchy can be presented as a single package or as a number of different packages, one for each type. This is pretty much a matter of taste, on which the author prefers to maintain neutrality.

Clearly, to do any real work requires that we have some types with actual information in them, such as numerical values, strings, or application-specific records.

6.2.2 Classwide Types and Pointers

For each type in the hierarchy, we will actually define two types — a tagged record type and an associated access type. We will illustrate the general pattern of type definition by giving the definitions for two types

- the abstract type *Sexpr_Impl*, which forms the root of the hierarchy, and
- the concrete type *Pair_Impl*, which corresponds to ordinary *cons* cells in our earlier implementation.

We can define the root type, *Sexpr* with the declarations

```
type Sexpr_Impl is abstract tagged private;
type Sexpr is access all Sexpr_Impl'Class;

function Is_Atom_Impl(S : Sexpr_Impl) return Boolean;
function Is_Seq_Impl (S : Sexpr_Impl) return Boolean;
... etc. ...
function Forced_Impl(S : Sexpr_Impl) return Sexpr;

function Is_Atom(S : Sexpr_Impl'Class) return Boolean;
function Is_Seq (S : Sexpr_Impl'Class) return Boolean;
... etc. ...
function Forced (S : Sexpr_Impl’Class) return Sexpr;

function Is_Nil(S : Sexpr) return Boolean;
function Is_Atom(S : Sexpr) return Boolean;
function Is_Seq (S : Sexpr) return Boolean;
... etc. ...
function Forced (S : Sexpr) return Sexpr;
```

The functions with *_Impl* in their names are dispatching functions called by the corresponding class-wide functions. The functions on the access types will simply call the corresponding class-wide functions on the object to which their arguments point.

Assume that the functions

```
function Head(S : Seq) return Sexpr;
function Tail(S : Seq) return Seq;
function Cons(X : Sexpr; S : Seq) return Seq;
```

have been defined with the *Seq* data type and that they are dispatched similarly to the functions on the *Sexpr* data type.

We can define the pair type with the declarations

```
type Pair_Impl is new Seq_Impl with private;
type Pair is access all Pair_Impl'Class;

function Is_Pair_Impl(S : Pair_Impl) return Boolean;
function Head_Impl(S : Pair_Impl; S1 : Seq) return Sexpr;
```

```
function Tail_Impl(S : Pair_Impl; S1 : Seq) return Seq;

function Cons_Impl(X : Sexpr;
                   S : Pair_Impl; S1 : Seq) return Seq;
```

with the private declaration

```
type Pair_Impl is new Seq_Impl with record
    Car : Sexpr;
    Cdr : Seq;
end record;
```

One rather peculiar feature of these definitions is that the *_Impl* functions have an extra parameter corresponding to the *Seq* data type. Both *S* and *S1* are required since

- we need *S* in order to do the dispatching, and
- we need *S1* in order to handle the low level implementation details.

In some cases, *S* will be the object to which S1 points, while, in other cases, *S* will be merely a dummy object that we use to control the dispatching.

6.2.3 A Question of Notation

One minor issue: that of notation. The usual Ada style is to define a record type and its associated access type with declarations like

```
type T is ... ; -- the record type
type T_Ptr is access T;
```

In dealing with graph-like structures, however, it is usually more convenient to deal with the access types than with the record types themselves. Accordingly, the author favors declarations like

```
type T_Impl is ...; -- the record type
type T is access all T_Impl;
```

6.2.4 Individual Atom Types

We can put atomic information, such as integers, floating point numbers, and strings, into the system by defining individual atom types.

For example, we might define an integer atom type, called *Int*, by deriving an *Int_Impl* type from *Atom_Impl*. We would add a field to the *Int_Impl* record to hold the integer value.

We could also add a creation function *New_Int*, which would convert an integer into an *Int*, as well as an extraction function *As_Integer*, which would convert an *Int* back into an integer.

We would also like to define a predicate to test a given Sexpr to see if it is an *Int*. Since the type *Sexpr_Impl* is already frozen at this point, we need to employ a mechanism other than inheritance to define the predicate. As it turns out, we can handle the predicates easily by using the class membership function. Thus, we could define *Is_Int* as

```
function Is_Int(S : Sexpr) return Boolean is
begin
    return not Is_Null(S) and then S.all in Int_Impl'Class;
end Is_Int;
```

6.2.5 Limitations of the Inheritance-Based Model

While inheritance allows us to retain strong typing, along with some of the flexibility of the original weakly typed model, the inheritance-based model requires a fairly tedious form of programming.

Although the *_Impl* types form a hierarchy, we must still perform explicit conversions of the corresponding access types, in order to use them effectively. For example, if we want to cons an *Int* onto a *Seq*, we must first convert the *Int* into a *Sexpr*. Thus, a typical call might look like:

```
Cons(Sexpr(New_Int(3)), Nil);
```

Another problem arises when we must deal with more than one hierarchy. Suppose, for example, that we have a hierarchy of bank accounts, given by

```
Bank_Account_Impl
    Savings_Account_Impl
    Checking_Account_Impl
```

Recall that we already have a hierarchy of sequence types given by

```
Seq_Impl
    Pair_Impl
    App_Impl
```

augmented by any specific types derived from *Pair_Impl* or *App_Impl*.

To define a hierarchy of sequence types involving specific kinds of bank accounts, we could either attach the bank account hierarchy to each type in the sequence hierarchy

```
Seq_Impl
    Bank_Account_Seq_Impl
        Savings_Account_Seq_Impl
        Checking_Account_Seq_Impl
    Pair_Impl
        Bank_Account_Pair_Impl
            Savings_Account_Pair_Impl
            Checking_Account_Pair_Impl
    App_Impl
        Bank_Account_App_Impl
            Savings_Account_App_Impl
            Checking_Account_App_Impl
```

Alternatively, we could attach the sequence sub-hierarchy to each member of the bank account hierarchy, giving

```
Seq_Impl
    Bank_Account_Seq_Impl
        Bank_Account_Pair_Impl
        Bank_Account_App_Impl
        Savings_Account_Seq_Impl
            Savings_Account_Pair_Impl
            Savings_Account_App_Impl
        Checking_Account_Seq_Impl
            Checking_Account_Pair_Impl
            Checking_Account_App_Impl
    Pair_Impl
    App_Impl
```

With the first choice, *Savings_Account_Pair_Impl* does not

derive from *Bank_Account_Pair_Impl*. With the second choice, *Savings_Account_Pair_Impl* does not derive from *Pair_Impl*.

Either choice results in some counterintuitive conclusions. (It is only fair to point out, however, that this is not a problem with Ada but, rather, a problem with most inheritance mechanisms.)

7. SOME LIMITATIONS IMPOSED BY ADA

The examples given above show how we can get around Ada's eager evaluation and lack of first class citizenship for functions.

There are still a number of difficulties in achieving the power of a truly functional language.

7.1 Lack of Closure Objects

The first problem is the lack of closures as objects. This is a consequence of Ada's stack-based memory allocation. Once we leave a scope, all definitions within that scope cease to exist. By contrast a functional language like Scheme or ML allows the creation of closures, enabling us to define a function such as

```
(define (make-counter)
  (let ((count 0))
    (lambda () (set! count (+ 1 count)) count)))
```

If we write

```
(define my-counter (make-counter))
```

then three successive calls to my-counter will produce the results 1, 2, and 3. We can also maintain several independent counters, each with its own internal state.

7.1.1 Simulating Closures via Augmented Parameters

Recall the example of Newton's method. In Scheme, it might look like

```
(define (sqrt x)
    (define (next y) (cons y (delay (next (/ (+ y (/ y x)) 2.0)))))
    (next 2.0))
```

The definition of *next*, nested within the definition of *sqrt*, takes the value of *x* from its enclosing environment.

In our version of Newton's method, the value of *x* was *cons*ed onto the front of the parameter list with each call. This represented perhaps the simplest way of adding closure-like objects to the system.

7.1.2 Defining a Closure Data Type

Another way to define a closure would be simply to cons a function object onto a list of name/value pairs.

We could even define a special kind of *Sexpr* called a closure, and allow *Apply* to work on a closure and list of arguments.

7.2 Lack of Function Builders

Virtually every functional language provides a mechanism for creating anonymous functions. For example, Lisp and Scheme provide the *lambda* construct for building functions. Thus, the expression

```
((lambda (x) (+ x 3)) 5)
```

in Scheme produces the answer *8*.

Since Ada does not provide this facility, functions are not, strictly speaking, first class citizens in Ada. This more of an annoyance than an actual problem, since we can always define functions statically and include pointers to them in our data structures. Adding an anonymous function facility to Ada would amount to building a *Sexpr* interpreter within Ada, which would probably not be worth the effort.

8. SOME RELATED WORK

Other efforts at realizing the benefits of higher order functions within a more conventional language include the Ada Generic Library [11], the C++ Standard Template Library [15], Kawa [1], and Pizza [12].

The first two come under the heading of generics-based approaches, while the latter two can be roughly categorized as language extensions.

8.1 Generics-Based Approaches to Higher Order Functions

The Ada Generic Library provided both singly linked and doubly linked lists within Ada 83. The library offered the programmer a number of underlying implementation choices upon which one could build higher order list operations. Higher order functions, such as *Map* and *Reduce*, were defined as generic functions.

The Ada Generic Library was notable in a number of ways:

- It used generics nested to deeper levels than most compiler vendors at the time had imagined would occur in use. Consequently, it broke a number of the leading Ada compilers at that time.
- It was designed from the beginning with efficiency in mind; thus, once compiled, the code was as efficient as hand written code.

The Standard Template Library (STL) of C++ was designed with the same goal of efficiency as the Ada Generic Library. Moreover, it was meant to unify a number seemingly different data structures, such as arrays and streams. STL provides a number of containers, iterators, and algorithms, all of which are handled in a uniform manner, regardless of the actual underlying data structure involved. In particular, applications involving user-defined data types will look similar to the corresponding applications involving built-in data types.

There has also been some recent work on providing an Ada analogue of the STL [4].

The system described in this paper differs from the generics- based approach in three ways:

- We are primarily interested in creating infinite data structures.
- We deal with function objects directly rather than

through generics or templates.

- We are not as concerned with efficiency as in the generics-based approaches.

8.2 Language Extension Approaches

Language extension involves either writing a preprocessor to the original language, or creating a new language that is a superset of the original.

Kawa and Pizza are two approaches based on extending the Java programming language. Kawa actually provides a Scheme interpreter that interprets Scheme by first compiling down to the Java Virtual Machine, then running the JVM code. Pizza works in a similar way, except that it uses the Java language as a base, adding functional language extensions to the Java language.

Our approach, again, is different in that we stay within the confines of the Ada programming language.

9. CONCLUSIONS AND FUTURE DIRECTIONS

Lazy evaluation offers a useful approach to defining sequences and other potentially infinite data structures; moreover, it lets us define a number of applications in terms of higher order functions.

Infinite data structures provide a unified view of a number of seemingly different objects, such as lists, arrays, files, random number generators, and external sensors.

Future directions for this work include

- specifying specific applications in terms of infinite data structures,
- combining lazy evaluation with distributed processing, such as a CORBA system, in order to treat a distributed or client/server application as a collection of infinite data types,
- extending the three models to handle general communicating sequential processes,
- building tools to handle the routine work of generating the appropriate dispatching functions for new classes in the inheritance-based model,
- providing tools to converting an application from the weakly typed model to the generic model and to the inheritance-based model, and
- fine tuning the system to increase efficiency.

10. ACKNOWLEDGEMENTS

The author is indebted to the referees for their valuable feedback on the submitted abstract. Their comments have greatly influenced the final version of the paper.

The author especially wishes to thank Norman Cohen for a number of excellent suggestions, particularly as regards the alternative implementations of infinite data structures.

11. REFERENCES

[1] Bothner, Per, The Kawa Scheme System, May 1997, available on-line as *http://www.cygnus.com/~bothner/kawa.html*.

[2] Duncan, A. G., Experience and Suggestions on Using Ada to Implement Internal Program Representations, in *Proc. of the Ada-Europe International Conference*., Dublin, Ireland, 1990, Cambridge University Press, 1990.

[3] English, J., *Ada 95: The Craft of Object-Oriented Programming*, London, Prentice Hall, 1997.

[4] Erlingsson, Ú. and A. Konstantinou, Implementing the C++ Standard Template Library in Ada 95, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, January 31, 1996.

[5] Friedman, D. and D. Wise, CONS Should Not Evaluate Its Arguments, in S. Michaelson and R. Milner (Eds.), *Automata, Languages and Programming*, Edinburgh, Edinburgh University Press, 1976.

[6] Henderson, P. and J. M. Morris, A Lazy Evaluator, in *Proc. of 3rd POPL Symposium*, Atlanta, GA, 1976.

[7] Hudak, P. and S. Peyton Jones, and P. Wadler, Report on the Programming Language Haskell, version 1.2, *ACM SIGPlan Notices*, *27*(5), 1992.

[8] Hughes, J., Why Functional Programming Matters, in D. A. Turner (Ed.), *Research Topics in Functional Programming*, Reading, MA, Addison-Wesley, 1990.

[9] Kelsey, R., W. Clinger, and J. Rees (Eds.), *Revised^5 Report on the Algorithmic Language Scheme*, February 1998.

[10] Kempe, M., Abstract Data Types Are Under Full Control with Ada 9X, *Proc. of Tri-Ada 94*, Baltimore, MD, 1994.

[11] Musser, D. R. and A. A. Stepanov, *The Ada Generic Library*, Springer-Verlag, 1989.

[12] Odersky, M. and P. Wadler, Pizza into Java: Translating Theory into Practice, revised from a paper presented at *24th POPL Symposium*, Paris, France, 1997.

[13] Paulson, L. C., *ML for the Working Programmer*, Cambridge, Cambridge University Press, 1996.

[14] Steele, G. L., Jr., *Common Lisp: The Language*, 2nd Edition, Digital Press, 1990.

[15] Stepanov, A. A. and M. Lee, The Standard Template Library, Tech. Rep. HPL-04-34, Hewlett-Packard, April 1994, revised July 7, 1995.

[16] Turner, D. A., An Overview of Miranda, in D. A. Turner (Ed.), *Research Topics in Functional Programming*, Reading, MA, Addison-Wesley, 1990.

12. APPENDIX: SOURCE CODE FOR THE *SEQ* PACKAGE

Here we present portions of the specification and body for the *Seq* package. We have omitted some features, such as *Sexpr* arrays and overloaded versions of the *Cons* function.

```
package Seq is
-- Basic Types
    type Symbol is ...;
    type Sexpr is private;
    type Sexpr_Function is access function(S : Sexpr) return
Sexpr;

    -- Builders
    function As_Symbol(Str : String) return Symbol;
    function Nil return Sexpr;
    ...
    function New_Int   (I : Integer) return Sexpr;
    ...
    function Cons(S1, S2 : Sexpr) return Sexpr;
    ...
    function New_Function   (Fcn : Sexpr_Function) return Sexpr;
    function New_Application(Fcn : Sexpr_Function;
                             S : Sexpr) return Sexpr;
    -- Predicates
    function Is_Int    (S : Sexpr) return Boolean;
    ...
    -- Values
    function Head(S : Sexpr) return Sexpr;
    function Tail(S : Sexpr) return Sexpr;
    function Int (S : Sexpr) return Integer;
    ...
    -- Side Effects
    procedure Set_Head(S : Sexpr; To_Be : Sexpr);
    procedure Set_Tail(S : Sexpr; To_Be : Sexpr);
    -- Function Application
    function Apply(F : Sexpr; Args : Sexpr) return Sexpr;
    -- Delaying and Forcing
    function Delayed(S : Sexpr) return Sexpr;
    function Forced (S : Sexpr) return Sexpr;
    ...
private
    ...
    type Int_Ptr    is access Integer;
    ...
    type Sexpr_Kind is (..., K_Int, ..., K_Fcn, K_App);

    type Sexpr_Record(Kind : Sexpr_Kind) is record
        case Kind is
            ...
            when K_Int => Int_Value : Integer;
            when K_Real => Real_Value : Float;
            when K_Pair => Car : Sexpr; Cdr : Sexpr;
            when K_Fcn => Fcn_Value : Sexpr_Function;
            when K_App => App_Op : Sexpr; App_Args : Sexpr;
        end case;
    end record;

    type Sexpr_Ptr is access Sexpr_Record;
    type Sexpr_Holder is record
        Sexpr_Value : Sexpr_Ptr;
    end record;

    type Sexpr_Ref is access Sexpr_Holder;
    type Sexpr is record
        Holder : Sexpr_Ref;
    end record;
    ...
end Seq;

package body Seq is
    ...
    function New_Int(I : Integer) return Sexpr is
        P : Sexpr_Ptr := new Sexpr_Record'(K_Int, I);
    begin
        return (Holder =>
                        new Sexpr_Holder'(Sexpr_Value => P));
    end New_Int;
    ...
    function Cons(S1, S2 : Sexpr) return Sexpr is
        P : Sexpr_Ptr := new Sexpr_Record'(Kind => K_Pair,
                                           Car => S1, Cdr => S2);
    begin
        return (Holder =>new Sexpr_Holder'(Sexpr_Value => P));
    end Cons;
    ...
    function New_Application(Fcn : Sexpr_Function;
                             S : Sexpr) return Sexpr is
        P : Sexpr_Ptr := new Sexpr_Record'(Kind => K_App,
            App_Op   => New_Function(Fcn),  App_Args => S);
    begin
        return (Holder =>new Sexpr_Holder'(Sexpr_Value => P));
    end New_Application;
    ...
    function Head(S : Sexpr) return Sexpr is
    begin
        case Kind(S) is
            when K_Pair =>
                return Forced(S.Holder.Sexpr_Value.Car);
            when K_App => return Forced(Head(Forced(S)));
            when others => raise CONSTRAINT_ERROR;
        end case;
    end Head;
    ...
    function Apply(F : Sexpr; Args : Sexpr) return Sexpr is
    begin
        return Fcn(F).all(Args);
    end Apply;
    ...
    function Forced (S : Sexpr) return Sexpr is
        Result : Sexpr;
    begin
        if Is_Application(S) then
            return Forced(Apply(Op(S), Args(S)));
        else
            return S;
        end if;
    end Forced;
    ...
end Seq;
```