Numerical Analysis Using Nonprocedural Paradigms

STEPHEN J. SULLIVAN
Mathcom, Inc.
and
BENJAMIN G. ZORN
University of Colorado at Boulder

This article presents a survey on the innovative features of a handful of languages that offer new features that can be valuable in numerical analysis, and a survey of the pros and cons of the languages with regards to work in numerical analysis. Language features such as polymorphism, first-class functions, and object-oriented programming offer improved writability, readability, reliability, and maintenance of computer software. The article discusses language features and uses, and includes a comparison of current implementations. It is intended both as an introduction to nonprocedural language features for persons working in numerical mathematics and as an exploration of some of the language requirements of numerical mathematics for persons working in language development. The article discusses C++, Fortran 77, Fortran 90, Haskell, Lisp/CLOS, Modula-3, Sather, and SML with respect to a variety of numerical analysis tasks: interpolation, optimization, array access and update, iteration, recursion, random number generation, and Gaussian elimination on sparse matrices.

Categories and Subject Descriptors: G.1 [**Mathematics of Computing**]: Numerical Analysis

General Terms: Languages

Additional Key Words and Phrases: Benchmarks, experimental languages, Gaussian elimination, linear algebra, programming languages, sparse matrices

1. INTRODUCTION

Many new languages offer features that can provide significant benefits for developers of mathematical software. In most large software projects the costs of personnel are far larger than the costs of computer resources, and the fraction of cost devoted to personnel continues to grow. Languages that reduce programmer time and increase software reliability, even at some cost in computer resources, can increase software reliability and reduce the time and cost of software development.

Authors' addresses: S. J. Sullivan, Mathcom, Inc., 8555 Hollyhock Street, Lafayette, CO 80026; email: Sullivan@mathcom.com; B. G. Zorn, Campus Box 430, Department of Computer Science, University of Colorado, Boulder, CO 80303; email: zorn@cs.colorado.edu.

Many problems are better modeled by functional or object-oriented paradigms than by more-traditional procedural approaches. This fit can make the translation from concept to program easier and more reliable, and is sometimes called the appropriateness of the metaphor or model.

Often, nonprocedural languages enable code reuse by reducing duplication of program code. Code reuse leads to better reliability, quicker time to completion, and ease of maintenance by reducing the length and complexity of programs.

Object-oriented programming is integral to a developing set of standards for interoperability [Eckerson 1993; Horn 1993; Mobray and Brando 1993; Nicol et al. 1993; OMG 1991]. Interoperability is the ability of software components, such as user interfaces, mathematics libraries, and graphics packages, to be used together with no prior knowledge of each other.

1.1 Features

The language features that enable the above benefits are discussed at length in the body of this article and are briefly described below.

The *first-class* objects of a language are those that may be dynamically created, passed to, and returned from a function, may be used in an expression, and may be an element in a data structure. *First-class functions* [Davie 1992; Louden 1993; Sethi 1989] are a feature in some languages: functions themselves may be passed to and returned from other functions, may be contained in data structures, etc. In many cases, use of first-class functions enables code that is more elegant and reliable. First-class functions are one of the defining characteristics of functional programming languages.

Some languages support *lazy evaluation* [Davie 1992; Hudak and Fasel 1992], meaning that no expression is evaluated until its value is actually needed. Thus a data structure may have an essentially infinite number of members, such as a list of all primes or of all derivatives of a function, since only a finite number will ever actually be used. Lazy evaluation facilitates dealing with many problems involving conceptually infinite lists and structures.

One of the most-powerful facilities of modern languages is the ability to apply a given routine to different types of data. For example, a single sort routine might be used to sort integers, floats, character data, and records. This ability is loosely called *genericity* or *polymorphism* [Louden 1993; Sethi 1989]: a routine may take generic arguments; its arguments may take "many forms."

In *object-oriented programming* [Booch 1993; Louden 1993; Wegner 1990], objects are data structures that combine both data and program routines related to that data. Object-oriented programming allows one to define new kinds of objects and their properties easily. For example, one might define a class of rational functions, each instance of which is a ratio of polynomials. One could define operations such as addition, multiplication, etc. on them. Using that class, one could write a program to solve a system of linear equations whose coefficients are rational functions (see J. T. Holland's Inter-

net posting, 1993). In a procedural language such as C or Fortran 77, the solution of the equations involving rational function coefficients would be a complicated undertaking: a new Gaussian package would have to be written from scratch, and the programmer would have to worry about the manipulation of rational functions at each step of the way. But with a more-flexible language, it is possible that an *existing* Gaussian elimination package, if previously implemented with an eye toward polymorphism, could be used *unchanged* to solve both the traditional problem with floating-point coefficients and the problem with rational function coefficients.

All modern languages contain some sort of *type system* [Louden 1993; Sethi 1989]. The languages C++, Fortran 77, Fortran 90, Modula-3, SML, Haskell, and Sather all use static typing, meaning that type checking (finding errors such as adding floats to characters) is done at compile instead of run-time. *Strong* type checking means that *all* type errors are discovered; *weak* means that some may go undetected. Common Lisp and CLOS use dynamic typing, meaning that type errors are not caught until run-time. Many implementations of Fortran 77 do not catch *any* errors in types or numbers of parameters passed to subroutines or functions, not even at run-time. This lack of support makes such errors difficult to find, since they can be discovered only as run-time logic bugs.

Very large problems require decomposition of some kind. While procedures can be used for this purpose, additional levels of modularization are helpful for decomposing very large problems. As a result, modular decomposition (i.e., breaking a program up into separate modules) is supported in many languages. Modules allow separate pieces of a large program to be constructed independently—modules are only allowed to interact through their interfaces and thus encapsulate their contents. Modules also support data abstraction—functionality provided by a module is entirely specified by its interface, and the implementation of that functionality is hidden from the user. Such programming languages allow the programmer to control the external visibility of procedures, variables, and types defined in a module.

1.2 Languages Reviewed

The languages reviewed are listed below. For compiler versions and availability, see Section 5.3. C++ is reviewed here following two usage styles: a C-like simple imperative style without classes and an object-oriented style using classes. We chose to review only general-purpose languages, and omitted Matlab, Mathematica, and similar special-purpose languages. For a more-detailed discussion, see Section 6.

—C++ is a large procedural object-oriented language with weak type support. C++ is a superset of C, with added object-oriented features [Coplien 1992; Ellis and Stroustrup 1990; Lippman 1991].

—Fortran 77, the current lingua franca of numerical analysis, is a procedural language [ANSI 1978].

—Fortran 90 is a successor to Fortran 77, with added features for user-defined types (records), array handling, dynamic allocation, interface definitions, and modules [ISO 1991].
—Haskell is a huge, sophisticated, pure functional language with modules and strong static typing [Hudak 1992].
—Common Lisp is a huge dynamically typed language with a long history, based on list processing [Steele 1990].
—Common Lisp Object System (CLOS) is an object-oriented extension to Common Lisp [Keene 1989; Steele 1990].
—Modula-3 is a large, procedural, object-oriented language with modules and strong static typing [Cardelli 1992].
—Sather is a small object-oriented language based on Eiffel and C [Omohundro 1990].
—SML is a small, elegant, mostly functional language with strong static typing [Milner 1990].

1.3 Article Overview

Section 2 presents an overview of functional programming. Several powerful features of functional languages are examined. Many small examples and two sample problems—interpolation and optimization—are presented that illustrate some of the pros and cons of these languages for numerical analysis work. Section 3 presents an overview of object-oriented programming, again with small examples. The same two example problems—interpolation and optimization—are revisited from the object-oriented point of view. Section 4 presents a variety of other language issues, including array handling, random numbers, exception handling, and garbage collection. And Section 5 presents a review of the performance and features of implementations of the reviewed languages. The benchmark used is a typical numerical analysis task: Gaussian elimination on sparse matrices. The implementations are compared with respect to compilation and execution times, error messages, and documentation. Sources for the implementations are also noted. Sections 6 and 7 present summary and concluding discussions.

2. THE FUNCTIONAL PARADIGM

There are a handful of powerful features found commonly in functional languages that could be of particular value in numerical analysis. For more-complete descriptions, please consult Cardelli and Wegner [1985], Davie [1992], Louden [1993], Hudak [1989], Hudak and Fasel [1992], Paulson [1991], and Sethi [1989].

2.1 First-Class Functions

The *first-class* objects of a language are those that have the properties listed below [Davie 1992]. As an example of a first-class object, consider floating-point values in C or C++.

—First-class objects may be given a name: float x;
—First-class objects may be passed to and returned from a function: y = f(x); x = f(u);
—First-class objects may be used in an expression: a = b + c*x;
—First-class objects may be used in data structures: struct s {float x; float y;}; or class s {float x; float y;};

First-class functions [Davie 1992; Louden 1993; Sethi 1989] are a feature in some languages: functions themselves may be passed to and returned from other functions, may be contained in data structures, etc. In many cases, use of first-class functions enables code that is more elegant and reliable. First-class functions are one of the defining characteristics of functional programming languages.

In a language supporting first-class functions, functions can be used as any other value; functions can be passed to functions; functions can return functions; elements of arrays can be functions; etc. While C, C++, and Modula-3 support pointers to functions, languages like SML and Haskell that provide full first-class functions allow far more flexibility.

A *pure* functional language is one that supports first-class functions and does not provide any means of changing the value of a variable, once it has been set. This restriction facilitates some types of computational analysis. Haskell is a pure functional language. SML is a functional, but since it, like C++ and Modula-3, supports the alteration of variables (e.g., i := i + 1;), SML is not pure. This issue is discussed more in Section 2.3. Functional languages offer notation for *anonymous* functions, sometimes called *lambda* expressions. Anonymous functions are, in essence, functions without their names—for example, the function $f(x) = x^2$ has name f. In functional language jargon, the identifier f has as its value the anonymous function that maps its argument x to x^2. The anonymous function is $x \rightarrow x^2$.

Some functional languages also support *currying*, which is the partial application of a function. For example, let f be a function of two arguments $f(x, y) = x + y$, and let $g = f(2)$. The function f is applied with only it first argument known, returning a function as a result. Thus the variable g is bound to the function that maps $y \rightarrow 2 + y$, or in other words, $g(y) = 2 + y$.

Consider the following examples of first-class functions, which could be written in either SML or Haskell:

Example 2.1.1 *Function Composition in SML.* Define a function *compose* that returns the composition of its two arguments, each of which is a function. The notation fn x => expression denotes an anonymous function that, when invoked, maps its argument x into the value of expression.

```
fun compose f g = (fn x => f (g(x)));  (* Define function compose *)
fun plus1 x = x + 1;  (* Define function plus1 *)
fun mult2 x = 2*x;    (* Define function mult2 *)

val addmult = compose mult2 plus1;  (* addmult (x) is 2*(x + 1) *)
val x = addmult 5;  (* The result is that x is bound to 12. *)
```

```
Or alternately·
val addmult = compose (fn x => 2*x) (fn x => x + 1);
val x = addmult 5,   (* The result is that x is bound to 12. *)
```

Example 2.1.2 *"Map" in Haskell.* The function map takes two arguments: a function f and a list la. When invoked, map returns a list lb, where the ith element of lb is $f(i$th element of $la)$. Note that in Haskell, function application is written f x y instead of the more-familiar f(x, y). An example of use:

```
la = [1, 2, 3, 4, 5]   --define list la
f x = x*x              --define fcn f, which squares its argument
lb = map f la          --define lb = [1, 4, 9, 16, 25]
```

Example 2.1.3 *Interpolation.* Suppose that a person has a set of (x, y) pairs and wishes to create an interpolating function f, such that $f(x)$ estimates y. Let the matrix amat consist of one (x, y) pair on each row.

In Fortran 77, Fortran 90, C++, or Modula-3, the solution would be similar to the following Fortran 90 code:

```
      call make_interp (amat, cvec)  ! Calc coefficients vector cvec
c  Now use coefficients in cvec to predict y for a given x
      y = cvec(1) + cvec(2)*x + cvec(3)*x**2 + cvec(4)*x**3
      y = use_interp (cvec, x)   ! Alternative way to do interpolation
```

But in a functional language like SML or Haskell, one could write instead:

```
f = make_interp (amat);  (* Calc and return interpolating function f *)
y = f(x);                (* Use function f to predict y for a given x *)
```

This example illustrates that first-class functions in SML or Haskell can be used to provide information hiding similar to that provided by the module mechanisms of Fortran 90, Modula, or Ada. Specifically, in this example, first-class functions provide the following advantages:

—The main procedure does not have to know about or deal with the type of interpolating function.

—The method of interpolation can be changed without changing any other procedures or functions.

—The interpolating function can be complicated (for instance, using different interpolating coefficients for different ranges of x) without cluttering the main program.

—The code to implement the interpolation function does not have to be repeated everywhere interpolation is used.

Example 2.1.4 *Optimization.* Another common example in which first-class functions are useful is when passing information from a high-level routine through a middle one to a low-level routine.

For example, suppose one has a general optimization program genopt (f, a, b), which finds the value of real x that minimizes a function f(x) over some interval [a, b].

But suppose f is really a function not just of x, but of integers i and j. And finally, assume that one wishes to try a handful of discrete values of i and j,

and find the optimum value of x for each. How can the main program communicate the values of i and j to f, without rewriting genopt?

In a procedural approach, the solution might be to keep i and j in global variables. Consider the following solution in Fortran 77:

```
program main
common / ijvals / globi, globj
integer globi, globj
... globi = ...,globj = ...
x = genopt (tstfun, a, b)
...
double precision function tstfun (x)
common / ijvals / globi, globj
interger globi, globj
tstfun = globi* x** globj
...
```

This solution has all the disadvantages of global storage. Although having a unique name for the common block or module does provide some security, it is still:

—difficult to know what functions alter or use global values,
—difficult to trace errors,
—difficult to maintain the program, as dependencies are implicit, and
—sometimes difficult to avoid inadvertent name collisions with other similarly named objects.

Another way to solve this program would be to rewrite the general optimization program genopt to accept parameters i and j, and to pass them through to function f. While this solution makes dependencies explicit, it is an even-worse solution in that it means rewriting genopt for each new use. Using this solution means:

—multiple versions of genopt,
—little code reuse for the versions of genopt, and
—maintenance and version control problems for genopt.

Another way to solve this problem would be to use modules as provided by Fortran 90, Modula, or Ada. Such a solution in Fortran 90 could be expressed as follows:

```
module funmod
integer, private :: locali, localj
contains
  subroutine reinit (ii, jj)
  integer ii, jj
  locali = ii
  localj = jj;
  end subroutine reinit
```

```
  real function testfunc (x)
  real x
  testfunc = ...
  end function f
end module funmod

program main
  use funmod
  ...calc new i, j...
  reinit (i, j)
  x = genopt (testfunc, a, b)
end program main
```

In this solution, the module funmod exports only two routines, testfunc and reinit. The subroutine reinit is provided to control access to the variables locali and localj in the module. They can only be modified by calling this subroutine. In an alterative solution, variables locali and localj could be made public, essentially making them global to all routines that "USE" funmod. Making these variables part of the interface has the disadvantage that they might be unintentionally modified elsewhere in the program.

Finally, in a language with first-class functions, such as SML or Haskell, one might write instead:

```
... calc new i, j...
let  fun fnew (x) = f(i, j, x)        (* Create function fnew which *)
                                      (* encapsulates the current i, j values *)
     val x = genopt (fnew, a, b)      (* Find x minimizing fnew for i and j *)
in ...
```

These languages allow the creation of new functions "on-the-fly." All information of i and j is encapsulated in the definition of fnew. An encapsulation of a function and variable bindings is termed a *closure*. The cost of creating a closure is similar to allocating a record for its environment, which is small. The cost of calling a closure is only marginally slower than calling a statically defined function, and is the same as calling a function passed as an argument to a Fortran routine.

In summary, first-class functions provide a way to encapsulate values that might otherwise be passed via globals or, worse yet, via general-purpose routines that must be rewritten for each special case.

2.2 Functional Programming Idioms

The lack of ability to modify variables in pure functional languages leads to a definite programming style. This style is often used in nonpure functional languages, such as SML, as well.

2.2.1 *Recursion.* In a functional-style program, *recursion* is used to accomplish iteration. For example, in a language such as Fortran or C++, iteration is accomplished by updating a loop variable repeatedly:

```
prod = 1;                  // calculate n factorial
for (i = 2; i <= n; i+)    // variables i and prod are
                           // repeatedly updated
  prod = prod*i;
```

But in a functional languages such as SML and Haskell, recursion would be used. In SML one could write:

```
factorial (n) = if n = 0 then 1 else n* factorial (n − 1)
```

Recursive functions can be written in all the reviewed languages except Fortran 77. While recursion is an occasionally used feature in nonfunctional languages, in functional languages recursion is the preferred way to accomplish any iterative task. In general, recursion is more easily accomplished and more elegantly expressed in functional languages than in procedural ones. Some languages (such as Scheme) guarantee "tail-recursion elimination," which means that, when possible, recursion is converted into iteration. This optimization is common in all functional language implementations and can often be applied. If this optimization is not performed, there is substantial cost to using recursion over iteration.

2.2.2 *Reduction.* A list of values may be *reduced* by applying a reduction operator, which collapses the list into a single value. For example, the list [1, 2, 3, 4] may be reduced by sum ([1, 2, 3, 4]), resulting in the sum. In general, a reduction operator specifies a binary operation and an identity value. In sum, the binary operation is addition, and the identity value is 0. In the reduction operator product, the binary operation is multiplication, and the identity value is 1. Functional languages such as SML and Haskell provide methods of creating reduction operators. In Haskell, one could write: sum = foldr (+) 0 and product = foldr (*) 1. Here (+) and (*) are sections: they are functions of two arguments, created from the binary operators + and *. Fortran 90 has sum and product intrinsics for intrinsic types, but they lack the generality of reduction operators.

2.2.3 *Pattern Matching.* SML and Haskell also support *pattern matching*. The formal arguments to a function may be built out of constructors. In SML, the operator :: prepends an element to a list, like the Lisp cons function. For example, the expression a::[b, c, d] yields the list [a, b, c, d]. The argument to function f, below, uses a pattern to decompose the given argument into the head (first element) and tail (remainder) of the list:

```
fun f(hd::tl) = hd*10;
```

So the expression f([3, 4, 5]) would yield 30.

Also, SML and Haskell functions may be defined using cases. For example, the following SML code defines a function f that maps 3 to 33, 4 to 444, and 5 to 5555.

```
fun f(3) = 33
  | f(4) = 444
  | f(5) = 5555
```

Another common idiom in functional languages is use of an *auxiliary* function, which is discussed next.

2.3 Pure Functional Programming

Haskell is a *pure* (side-effect-free) functional language, meaning essentially that values may be defined, but never modified. Standard C constructs like i = i + 1; or x = 0; . . . ; x = 1; cannot be used.

Pure functional languages have *referential transparency*, meaning that a variable retains the same value throughout its scope, and may be replaced by its definition with no change to program meaning.

For example, contrast the following programs to sum the elements of a one-dimensional array in Fortran 90 and Haskell:

Fortran 90.

```
double precision function sumvec (veca)
double precision, dimension (:) :: veca
integer i
double precision sum
sum = 0.0;
do i = 1, size (veca)
  sum = sum + veca (i)
enddo
sumvec = sum
return
end
```

Note that two variables are updated: the loop index i and the accumulator sum.

Haskell. Array indices run from 1 to n. Define the function sumvec using an *auxiliary* function sumveca. An auxiliary function is a local function for use solely in the function defining it. In the following example, the auxiliary function sumveca takes parameters vec, n, and i, and sums the elements of vec from i to n using recursion. In Haskell, ! performs array subscripting, so vec!i is equivalent to the more familiar vec(i) or vec[i].

```
sumvec vec n = sumveca vec n 1   --Main fcn sumvec sums from 1 to n
  where sumveca vec n i = if i > n then 0   --Aux fcn sums from i to n
                          else (vec!i) + sumveca vec n (i + 1)
```

Note that in this version, variables are not updated. The following Haskell function illustrates a more-elegant way:

```
sumvec vec n = sum (map (vec!) [1..n])
```

This example illustrates several features common in functional languages. Here, [1..n] generates a list of indices 1 through n. The ! operator performs subscripting, so vec!3 returns the third element of vec. The construct (vec!) is a section, and represents the function that, when applied to k, returns the kth element of vec. So (vec!) k is the kth element of vec. The map function applies its first operand, here the function (vec!), to each element of its second operand, which here is the list [1..n]. The result of the map application in this case is a list of elements of vec. Finally, sum returns the sum of the elements of the list. While this explanation may seem tedious, the example shows that Haskell does offer numerous ways to create functions and use them, i.e., first-class functions lead to new programming idioms.

Pure functional languages present large problems in dealing with state (updatable variables) and I/O. Since there are no updates in the pure functional paradigm, any algorithms based on updating values or arrays (i.e., nearly all numerical algorithms) must be redeveloped to accommodate the pure functional style. This is not a matter of rewriting or translating an algorithm: in general, the basic algorithm must be redeveloped. Additionally, dealing with I/O in a pure functional program can be extraordinarily complicated. A program must be viewed as a function of all its input data that returns all its output data. Simple programs using just a few lines of input and output can become surprisingly convoluted. For more details on the complexity of Haskell I/O, see Hudak and Fasel [1992]. This is a topic of current research, and promising methods to alleviate these difficulties are being explored [Peyton Jones and Wadler 1993].

2.4 Lazy Evaluation

Lazy evaluation, sometimes called "nonstrict semantics," means that no expression is evaluated until absolutely necessary: if a result is never used, the program statements used to calculate it never get executed. Haskell uses lazy evaluation. All the other reviewed languages have strict evaluation, meaning that program statements are executed as determined by standard flow of control. To gain a feel for lazy evaluation, consider the following examples:

C++.

```
x = 1.0;      // set value x
y = x / 0.0;  // set y to x / 0.
              // Produces error at this point.
```

Haskell.

```
x = 1.0          --define value x
y = x / 0.0      --define y as x / 0. y is not evaluated yet.
f z = 3          --define function f, which always returns 3.
answer = f y     --evaluate f(y). f or y returns 3.
                 --not error, since y is never evaluated.
```

Lazy evaluation can be useful in defining infinite objects, as illustrated by the following Haskell example. In Haskell, list indexing is performed by !!, and the first element of a list is numbered 0.

```
f x = x*x          --define function f,
                   --which squares its argument
x = map f [1..]    --define value x as infinite list
                   --of squares [1, 4, 9, 16, ...]
answer = x !! 3    --get fourth element of x, 16.
```

As another example, define a function *fiter* that takes a function f and value x as arguments, and returns the infinite list $[f(x), f(f(x)), f(f(f(x))), f(f(f(f(x)))), \ldots]$. In Haskell, the single colon : is the cons operator—like the double colon :: of SML. In Haskell, if x is a value, and la is the

list [a, b, c], x:la is the list [x, a, b, c]. The double bang !! operator selects an element from a list: la!!0 returns the first element of list la.

```
fiter f x = x : fiter f (f x)   --define function fiter
  g x = x*x                     --define function g,
                                --which squares its arg
la = fiter g 2          --define la = [2, 4, 16, 256, ...]
answer = la !! 3        --get fourth element of la, 256.
```

Although lazy evaluation can be mimicked in nonlazy languages, it becomes convoluted and tends to obscure the meaning of the program. Typically, in a nonlazy language a lazy list is implemented by a function that returns the next list element on each succeeding call, or by one of several similar methods.

Some optimizing compilers for conventional languages such as Fortran 77 provide an effect similar to lazy evaluation. Such compilers use dead-code elimination, which simply elides sections of code whose results will not be used. While this is certainly beneficial, it lacks the generality of lazy evaluation in dealing with infinite data structures.

2.5 Polymorphism

In brief, *polymorphism* is the ability of a function or operator to accept arguments of more than one type. In *parametric* polymorphism a function or operator may be used with different types, but always performs the same actions on its arguments. For example, some languages have a "length" function that returns the number of items in a list. The same function works on lists of integers, lists of floats, lists of strings, etc. In ad hoc polymorphism or *overloading*, a function or operator performs different actions, depending on the types of its arguments. For example, the + operator can be used to add integers or to concatenate strings. There are many types of polymorphism, and a large literature on the subject exists. For an overview, see Louden [1993] and Cardelli and Wegner [1985].

As an example of parametric polymorphism, consider a function "head" that returns the first element of a list. The elements in the list can be of any type α, so that function argument is of type "list of α," and the returned value of type α.

As another example, consider writing a "sort" function. Ideally, it would sort integers, float, strings,... or any type for which the comparison operation < is defined. The ideal sort could handle arbitrary structures or records for which an operator < is defined.

Here is a comparison of the languages with respect to parametric polymorphism.

C++. *C++* provides parametric polymorphism through "templates." While templates provide the flexibility to solve problems like the sorting discussed above, they are clumsy and, in current implementations, can be space inefficient.

Fortran 77. Although Fortran 77 has a handful of generic intrinsic functions, there is no facility for user-defined functions.

Fortran 90. Fortran 90 allows user-defined, generic functions. However, this is much different than the parametric polymorphism provided by SML, for example. A sort function in SML could sort *any* type for which $<$ is defined. In Fortran 90, the programmer must replicate the sort routine for each type to be sorted. While not difficult for a small number of types, having many nearly identical copies of a routine can become a maintenance headache.

Modula-3. Modula-3 provides generics, based on its module and interface system. While workable, it is relatively clumsy. Defining a generic procedure involves four files: the generic procedure, its instantiation, the generic interface, and its instantiation. In Harbison's text on Modula-3 [Harbison 1992] he writes "The Modula-3 generic facility may seem somewhat cumbersome for simple generics."

SML and Haskell. SML and Haskell use *type variables* to provide parametric polymorphism. Haskell also provides a system of *type classes*. A type variable is a variable representing the type of an ordinary variable. For example, the SML type variable 'a would have the value float when representing a floating-point variable. Type classes are sets of types. For example, the Haskell type class Num contains the classes Integral and Floating. SML uses functors and structures to provide other types of abstractions. While the type classes of Haskell provide more flexibility than the type variables of SML, Haskell's module system lacks the flexibility of SML's. In short, some operations that are easy to perform in Haskell are difficult in SML, and vice versa.

SML. SML has a full built-in system of type variables. However, it has only two type classes, in the Haskell sense of type class. SML type classes are the class of all types, denoted 'a, where a may be any letter, and the class of types supporting an equality predicate (''a). The equality type variables (''a) can be instantiated only by types for which the equality operator (=) is defined. This distinction is useful in defining some functions—for example, to test a list to see if it contains a given element:

```
fun member (x:''a, [ ]) = false (* x cannot be in the empty list *)
    (* x is in list if it is the first element *),
    (* or if x is in remainder of list (ys) *)
  | member (x:''a, y::ys) = if x = y then true
                            else member (x, ys);
```

For more-general polymorphism, SML uses a module and signature system similar to that of Modula-3.

Haskell. Haskell has a fully built-in system of type variables. It also has classes of types that can be used with "contexts" to specify arbitrary constraints on the types to which a function applies. Haskell handles the constrained polymorphism needed to write a generic sort function in a straightforward and elegant manner. For example, to define a type "Pair of a" where "a" is any type, one would write "data Pair a = Pr a a." Further, the

Table I. Summary of Functional Features

	First class fcns	Recursion	Reduction	Pattern matching	Lazy evaluation	Parametric poly-morphism
C++		Yes				Yes
Fortran 77		No (Note 1)				
Fortran 90		Yes				
Haskell	Yes	Yes	Yes	Yes	Yes	Yes
Lisp	Yes	Yes	Yes			Yes
Lisp/CLOS	Yes	Yes	Yes			Yes
Modula-3		Yes				Yes
Sather		Yes				Yes
SML	Yes	Yes	Yes	Yes		Yes

Note 1. Some Fortran 77 implementations allow recursion in subroutines, although the standard forbids it.

"Eq" class is defined in the Haskell prelude, and contains those types for which equality (==) and not-equal (/=) operators are defined. The type "Pair of a" can be included in class Eq if "a" is in Eq and if we define the equality operator as, for example,

```
instance (Eq a) => Eq (Pair a) where
  (Pr a b) == (Pr c d) = (a == c) && (b == d).
```

The "/=" operator is defined automatically, using a default rule that specifies "/=" is not "==." In a similar vein, the "Ord" class is also predefined in the Haskell prelude, and contains those types that are in class Eq and additionally support operators <, <=, >, >=, min, and max. The type of "Pair of a" could be included in class Ord with a definition similar to the above, and a general sort function that takes any list of class Ord types would work on "Pair of a," assuming "a" is in Ord. Although the classes Eq and Ord are defined in the Haskell prelude, the mechanism of class definition is general. Had they not been predefined, a programmer might easily define them. For an example, see Hudak [1992].

2.6 Summary of Functional Features

See Table I.

3. THE OBJECT-ORIENTED PARADIGM

There are many widely varied programming systems that have been called "object-oriented," and nearly as many definitions of the object-oriented (OO) paradigm as there are authors of OO papers. For background material,

see Wegner [1990], Louden [1993], and Booch [1993]. Here we consider the salient features of an OO system to be:

Class Structure. Classes may be defined, usually at compile-time. Instances or members of a class, called objects, may be created at run-time. A class may define some or all of the following:

- Shared variables: a shared variable only occurs once in the class, and all objects of the class share the same instance of the variable.
- Instance variables, sometimes called slots or fields: an instance variable has a separate instantiation in each object within the class—each object has its own copy.
- Methods: methods are functions that are invoked via an object. Typically, a method is invoked by notation like objа.metha (parm1, parm2, ...), where objа is an object of a class classa, and metha is the name of a method associated with that class. In some systems, methods are called messages, and although the metaphor is based on messages and responses, the functionality is similar to that of methods.

Inheritance. A class Beta may be defined as a subclass of a class Alpha, in which case each Beta object "inherits" all items—shared variables, instance variables, and methods—of class Alpha and may access them as if they were defined in class Beta. Class Alpha is called the *parent* of Beta. Some OO systems, such as C++, allow class Alpha to restrict the items that Beta may inherit. In one common method of method handling, if an application invokes method M associated with an object of class Beta, the class Beta definition is searched for a method matching M's signature (parameter types). If found, it is invoked. If not found, a similar search is performed on the parent class (Alpha) of Beta, and so on up the class hierarchy until either finding a method matching M's signature or reaching the top of the class hierarchy, in which case an error is signaled. Many languages support the ability to inherit from more than one parent class, which is known as multiple inheritance.

Dynamic Dispatching. When a function associated with an object is invoked, the actual function invoked depends on the type of object and on its class hierarchy. For example, assume class Alpha, representing a small volume element, has subclasses Beta and Gamma. Assume that the method M is defined for all three classes, so that M for the subclasses overrides M for the parent class Alpha. Assume l is a list of objects of class Alpha. If an application program selects an object o from l and invokes method M on o, the version of M actually executed will be the method associated with Beta or Gamma if o is in Beta or Gamma, and will be the method of Alpha otherwise. Dynamic dispatching is critical to object-oriented programming, but can make static type checking difficult or impossible.

There are a great variety of additional features in various OO systems, but for the purpose of this article we will focus on the above three.

The set of classes for a system is usually represented as a tree, with the root at the top, and subclasses descending from their parents. This tree is sometimes called the class hierarchy. If class A is above class B in the

hierarchy, class A is the parent of B, and is sometimes called a *base class* for B. The transitive closure of parents of B are the *ancestors* of B; the transitive closure of subclasses of B are the *descendents* of B.

There are two main benefits to using the OO paradigm instead of standard procedural methods. First, OO programming allows more code reuse. In many applications, a person may need several versions of a routine, each slightly different from the preceding one, to accomplish a variety of similar tasks. When implemented in a procedural language, this requirement often means maintaining several similar copies of the routine, leading to ample opportunity for errors. In the OO paradigm, usually the routines can be implemented as subclasses of a base class, and the subclasses need only specify the differences from the base version. There are other ways in which OO programming facilitates code reuse, which are covered below.

The second benefit of the OO paradigm is that it often provides a more-natural model for the physical world. Often, physical and computational systems are approached in terms of a hierarchical model, which are represented far more easily using the OO paradigm than the procedural paradigm.

Of the reviewed languages, C++, Lisp/CLOS, Modula-3, and Sather qualify as object oriented.

There are two main approaches to setting up a class hierarchy:

Method A. Have the class structure represent our conceptual model of a situation. Classes higher in the tree represent more-general categories (say, vehicles), and subclasses represent more-specific categories (such as aircraft, automobiles, and under them trucks, cars, etc.).

Method B. Have the class structure facilitate program implementation. For example, an automobile dealer that handles a great many cars and only an occasional large truck or motorcycle might have a base class "car," containing fields for "manufacturer," "model," and "price." The "car" class might have a subclass "general_vehicle," containing the fields "num_wheels," "payload_capacity," "empty_weight," etc. Although this class system seems conceptually inverted, it eliminates the need to store irrelevant data for each car, and it still allows the subclass "general_vehicle" to make full use of all the methods of class "car" for determining vehicle price, manufacturer, etc.

The OO paradigm lends itself to groups of similar objects over which a common set of operations is defined, and hierarchies over such groups. Three areas that are common candidates for OO implementation are:

—Computational objects: objects internal to a program, such as representations for numbers.

—Graphical user interface (GUI) objects: windows, graphs, etc.

—External objects: objects in the problem domain, such as antennas, automobiles, aspects of a finite-element model, subatomic particles, etc.

3.1 Examples

Some examples of the OO paradigm for computational, GUI, and other objects are shown below. The class hierarchy is indicated by indentation.

Examples of Computational Objects

Numbers:
A base class, number, represents all numbers
  Subclass int represents integers
    Subclass stdint uses represents integers stored in 4-byte fields
    Subclass arbint uses arbitrary-precision integers
  Subclass complex represents complex floating-point values
    Subclass float represents floating-point values
      (Note: this is arranged by the conceptual organization method (Method A above), as complex values are more general than floats. However, a practical organization might use Method B: class complex would be a subclass of class float. Class float would contain the real part, and class complex would have an additional field for the imaginary part.)

Matrices:
A base class, matrix, represents all matrices
  Subclass dense_matrix represents matrices stored in standard dense format
    Subclass row_vector represents row vectors as dense matrices constrained to have only 1 column.
    Subclass column_vector represents column vectors as dense matrices constrained to have only 1 row.
  Subclass sparse_matrix represents matrices stored in sparse format
  Subclass diag_matrix represents diagonal matrices. Only the diagonal elements are actually stored.

Finite-element modeling:
A base class represents general elements.
  Subclasses represent elements having edge effects.

Algebra:
A base class, group_element, represents elements of an algebraic group.
  Subclass field_element represents elements of an associated algebraic field.

Examples of Graphical User Interface Objects [*Paxson et al. 1989*].

A base class, window, represents a general graphics window.
  Subclass basic_graph represents a graph using pixel coordinates.
    Subclass world_graph represents a graph using world coordinates (floating-point value between 0 and 1).
      Subclass labeled_graph represents a world_graph with scales and labels.

Examples of External Objects.

A base class, widget, represents some equipment type.
  Subclass rack_mounted_widget represents those requiring rack mounting.
  Subclass enclosed_widget represents those requiring a separate enclosure
  Subclass rs232_widget represents those using an RS232 serial interface
  Subclass GPIB_widget represents those using a GPIB interface
    Subclass rs232_enclosed_widget is formed by inheriting from both the rs232 and enclosed classes.
    Other subclasses are formed similarly.

3.2 Object-Oriented Idioms

The OO paradigm has spawned several programming idioms. Two of the more-common ones are described below.

3.2.1 *Wrappers.* One of the great benefits of OO technology is the reuse of code, by allowing incremental changes to procedures. A base class may define a method M, and a subclass may modify the method M by defining its own method M, thereby overriding the base class definition. The subclass M is often structured as:

—Perform subclass-specific initialization.
—Invoke the base class method M to do the bulk of the work.
—Perform subclass-specific postprocessing.

Thus the subclass method M "wraps around" the base class method M, and contains only the desired differences from the base method M instead of an entire copy of it. In the GUI example above, the subclass methods wrap around their parent class methods.

3.2.2 *Abstract and Concrete Classes.* An abstract base class has no instance variables or shared variables, although methods may be fully defined. Abstract classes are sometimes called *stateless* classes. Any variables must be declared in subclasses. A *pure* abstract base class has the additional restriction that while a method may be declared in it by specifying only the method's signature, the full method definition may not be specified in the abstract base class. In this case, the method must be redeclared in a subclass, where the full definition of the method is specified.

An abstract base class is useful for specifying a uniform interface for all subclasses below it. The abstract/concrete idiom is useful in separating the interface specification (the abstract class) from its various implementations (the subclasses). It is sometimes known as the "handle/body" idiom [Coplien 1992].

For example, in the matrix class in Section 3.1, the matrix could be implemented as a pure abstract base class, with no variables. The subclasses, such as sparse_matrix, would contain both the variables and the implementations of the functions first declared in matrix.

3.3 Interpolation and Optimization Examples Revisited

As an example of class usage, consider again the problem of finding an interpolating function from Section 2.1. An OO solution might define a class "interpolator," and within that class define methods to calculate and use interpolation coefficients. Each object of the class represents one interpolation function (one set of interpolation coefficients). In C++, the interpolation solution is:

```
// Each object in class "interpolator" represents one interpolation
// function: that is, one set of interpolation coeffs.
class interpolator {
private:
  vsmcolvecf cvec;                    // vector of coefficients
```

```
public:
  void make_interp (vsmmatf amat) {      // method to calc coeffs
    //...compute cvec based on amat      ...
    }
  float use_interp (float x) {           // method to use the coeffs
    //...compute y = cvec[0] + cvec[1]*x + cvec[2]*x*x + ...
    return y;
  } // end of use_interp
}; // end of class interpolator

// To use the interpolator, one could then program:
  interpolator interpa;                  // create an object interpa
  //...fill amat...
  interpa.make_interp (amat);            // calculate interpolation coeffs cvec
  y = interpa.use_interp (1.234);        // use coeffs to perform interpolation
```

This implementation of the interpolation function shows essentially the same structure as the functional solution, although the OO solution is less elegant than the functional. The OO solution offers the same advantages: separation and encapsulation of interpolation specifics apart from the main program, thereby enhancing maintainability and reliability. Modula-3 and CLOS implementations follow similar lines, but are more verbose.

As a second example, consider again the optimization example from Section 2.1. In that example the optimization routine genopt (f, a, b) took as parameters the objective function f, and the lower and upper search limits. In the OO paradigm, we use an optimization function genopt (obja, a, b) that takes as parameter the object obja instead of the function f. The class of obja defines a method for the function f, as well as variables for any additional information (such as additional parameters i and j) needed by f. Since different applications would require different functions f, the class definition of the object passed to genopt would change from application to application. Therefore the function genopt must have parametric polymorphism. In C++, parametric polymorphism is implemented via templates.

In the OO implementation shown here, the genopt procedure only uses the f method associated with object obja internally; any other slots or methods associated with obja may be used to communicate between genopt's caller and f. In C++ the solution would look like:

```
template <class TP> double genopt (   // genopt, the general opt proc
  TP optobj,   // TP is the class of the passed object
  double a,    // low search limit
  double b)    // high search limit
  {
    //...xmin = ...;   // calc and return value of x that minimizes f return xmin;
  } // end of genopt func

// To use the optimization for a specific application, one could write:
class optclass {              // define class with method f for objective func
public:
  int savei, savej;           // values of i and j for use by function f
  double f (double x) {       // objective function f
    ...resval = some objective function...
    return resval;
  }  // end of function f
```

```
}; // end of optclass class
int main (int argc, char *argv[ ]) {
  optclass optobj;      // alloc object to be passed to genopt
  //...calc new i, j...
  optobj.savei = i;     // set i and j in the object
  optobj.savej = j;
  // find x minimizing f on the interval [a, b], for the given i and j.
  xmin = genopt (optobj, a, b);
}  // end main
```

Sather handles the problem in a similar fashion but more elegantly, primarily because parametric functions in Sather are far easier to use than the clumsy template mechanisms of C++.

The main disadvantage of the OO approach, when compared with the functional approach, is that a separate class must be defined to serve as the parameter carrier for each type of invocation of genopt. In C++ and Sather, this class has global visibility, cluttering the global name space. In functional languages, the function passed to genopt can have either a local name or no name (anonymous). None of the reviewed languages allow the definition of anonymous classes.

The syntax of templates in C++ is particularly convoluted; this simple example does not begin to show the size of the problems that come with larger programs. The GNU C++ compiler does not yet implement templates fully. Some commercial C++ compilers do implement templates, but often C++ templates take so long to compile and link that many programmers avoid using them (personal communication, P. Jensen, 1993).

The CLOS implementation of the optimization example follows the same lines as the C++ and Sather versions: define a class whose objects enclose the parameters to be passed from genopt's caller to the objective function f.

4. MISCELLANEOUS LANGUAGE FEATURES

4.1 Array Handling in General

The languages offer different facilities for creating, accessing, and updating arrays. In some cases, the array origin (index number of the first element) is fixed at 0 or 1, in other cases it is flexible. See Table II.

4.2 Mathematical Functions

All the reviewed languages have intrinsic functions for exp, log, cos, sin, and sqrt. All except SML additionally have tan, acos, asin, cosh, sinh, tanh. Haskell, Lisp, and Modula-3 support in addition the inverse hyperbolic trig functions. Why so few mathematical functions in SML? All the functions available in C++ are easy to define in terms of these six. Perhaps the authors of SML did not provide a more-complete set because, like Niklaus Wirth [Wirth 1988], they value minimality. Unfortunately, this means everyone using the functions has to define them personally. The only languages with intrinsic support for complex values are Fortran 77, Fortran 90, Haskell, and

Table II. Array-Related Features

	Array Origin	Supports arrays of arbitrary types	Subscript checking	Supports array updates	Supports looping	Supports recursion
C++ as C	0	Yes	No	Yes	Yes	Yes
C++ with classes	user-specified	No	Yes	Yes	Yes	Yes
Fortran 77	user-specified (Note 3)	No	optional	Yes	Yes	No
Fortran 90	user-specified	Yes	optional	Yes	Yes	Yes
Haskell	user-specified	Yes	Yes	No (Note 1)	No	Yes
Lisp	0	Yes	Yes	Yes	Yes	Yes
Lisp/CLOS	user-specified	Yes	Yes	Yes	Yes	Yes
Modula-3	0 (Note 2)	Yes	Yes	Yes	Yes	Yes
Sather	0	Yes	optional	Yes	Yes	Yes
SML	0	Yes	Yes	Yes (Note 1)	Yes, but awkward	Yes

Note 1. *Pure* functional languages such as Haskell do not support updatable (mutable) objects, such as updatable arrays. Haskell does provide a facility to create an updated copy of an (unchanged) existing array. SML provides both immutable (pure functional) and mutable (updatable) arrays. However, the immutable arrays are only partially implemented in the version of SML used here (see the implementations information in Section 5.3), so all SML array work in this article relates only to SML's mutable arrays.

Note 2. In Modula-3, arrays lose all index information when they are passed to a subprocedure. For example, an array dimensioned [1:10] becomes [0:9] in a called procedure. This is a lack of referential transparency in the language: avec[1] means different things, depending on which procedure it appears in. If a caller passed an array and an index value of an element to a subprocedure, the index value would be invalid in the subprocedure.

Fortunately, most numerical analysis problems are best written with dynamically allocated arrays (whose dimensions are determined at run-time), and in Modula-3 such arrays must start at origin 0. However, when a programmer uses a statically allocated array (whose dimensions are specified at compile-time, such as [1..256]), the change of origin on entering a subprocedure can cause run-time bugs that are difficult to resolve.

Although Modula-3 does have FIRST and LAST functions that give the first and last indices of an array, they seem more like afterthoughts than features. Even if the above subprocedure were coded as

```
v = avec[FIRST(avec) - 1 + k]
```

the subprocedure must still know the origin of vector avec (here hardcoded as "1"). And in any event, avec[FIRST(avec) − 1 + k] is far less clear to the reader than avec[k].

In Modula-3, arrays may be indexed by arbitrary enumerated sets. The question arises: when passing an array to a subprocedure, how does the compiler inform the

Table II—*Continued*

subprocedure of the array's index set? This question is particularly meaningful when the main and subprocedures are compiled separately. Some possible solutions to the problem are:

—Decide that all arrays must be indexed by integers, and must have fixed origin. This is the solution used in C and SML.

—Decide that all arrays must be indexed by integers, but may have arbitrary user-specified origin. The origin information is carried with the array when it is passed to a subprocedure. This is a solution available in Fortran 90, and can be implemented in C++ matrix classes.

—Decide that arrays may be indexed by arbitrary enumeration sets, but that no enumeration set information is passed to subprocedures. Programmers must maintain a coding style carefully that avoids possible confusion on array indexing. This is the solution used in Modula-3. In the authors' view, this choice compromises seriously the safety and utility of the language in handling arrays. In defense of Modula-3, it may be said that in most cases the origin defaults to 0, and if the programmer always uses 0-based indexing, the result is much like coding in C. This is similar to the solution used in Fortran 77: see Note 3.

—Decide that arrays may be indexed by arbitrary enumeration sets, and that the enumeration set must be defined before the function using it. This is the solution used in Haskell.

Note 3. In Fortran 77 arrays lose all index information when they are passed to a subprocedure. For example, an array dimensioned (3, 5) becomes (1, 3) in a called procedure. The programmer may get around this difficulty by passing explicitly the array's lower and upper bounds as procedure parameters, and in the subprocedure dimensioning the array as arrayname (lobound, hibound).

Lisp. However, it is a straightforward matter to define complex operations in the remaining languages.

4.3 Random Numbers

Many languages come with random number generators (RNGs), and they nearly all fail at generating good random numbers [Marsaglia 1993; Marsaglia and Zaman 1991; Park and Miller 1988; Press et al. 1988; Sullivan 1993]. C++ provides no built-in RNG.

Modula-3 provides a set of procedures based on the linear congruential method. We did not test it for randomness. SML version 0.93 provides an RNG, but unfortunately it is based on the Park and Miller RNG, which fails some tests for randomness [Marsaglia 1993; Sullivan 1993]. The next release of SML should contain Marsaglia's more-robust subtract-with-borrow generator [Marsaglia and Zaman 1991]. Sather provides several RNGs, but unfortunately the default is the flawed Park and Miller RNG.

The Haskell library includes a linear congruential method that, while not credited, appears identical unfortunately to the Park and Miller RNG. In pure functional languages such as Haskell, the use of *any* type of RNG is problematic. In pure functional languages, one must make the RNG a function of the old seed, since no state information can be retained, and must specify the old seed on each call. This means that if a sub-subfunction in a

large program uses a random number generator, the seed must be passed up to all its ancestors and back down again on the next call, even if none of the ancestors deal with random numbers. This can be a burden on the programmer.

This requirement that the RNG seed information be passed up and down the call chain implies more conceptual load for the programmer using functional languages. In effect, RNGs cannot be encapsulated, since every ancestor must deal with their state. Random number generators fit naturally in an object-oriented paradigm; in a pure functional paradigm, RNGs are problematic.

The difficulty is in the metaphor: functional languages have referential transparency, implying no side-effects, and by definition RNGs, like input/output operations, involve side-effects.

4.4 Language Syntax

The syntax used in mathematics varies from language to language. In some it may appear a bit unnatural. Consider the following comparison of statements to update an element of an array y:

```
C++:      y[i] = a + b*x[i] + c*x[i]*x[i];
Fortran:  y(i) = a + b*x(i) + c*x(i)**2
Haskell:  Not possible to update variables or array elements.
Lisp:       (setf (aref y i) (+ a(* b (aref x i)) (* c (aref x i) (aref x i))))
Modula-3: y[i] := a + b*x[i] + c*x[i]*x[i];
Sather:   y[i] := a + b*x[i] + c*x[i]*x[i];
SML:      Array.update(y, !i, !a + !b*Array.sub(x, !i)
                          +!c*Array.sub(x, !i)*Array.sub(x, !i));
```

4.5 Modularity

Supporting modularity involves defining mechanisms for creating modules and allowing programmers to control the visibility of the contents of the module. Classes in object-oriented programming languages serve many of the same functions as modules in modular programming languages, although there are also subtle differences. Of the languages reviewed, Fortran 77 supports modularity only via procedures, and C++ and Sather provide only classes. Fortran 90, Haskell, and SML provide modules only. Lisp/CLOS and Modula-3 support both modules and object classes.

4.6 Exceptions

Often a programmer wishes to signal an exceptional condition instead of aborting the entire program, and wishes to have control return immediately to some distant ancestor in the call chain. C++, Lisp, Modula-3, Sather, and SML provide exception-handling capabilities. Fortran 77, Fortran 90, and Haskell do not.

4.7 Dynamic Allocation

Dynamic allocation is the allocation of objects at run-time instead of compile-time. This is useful in allocating arrays whose dimensions are not known

Table III. Summary of Miscellaneous Features

	Language syntax readability	Modules	Classes	Exception handling	Dynamic allocation	Garbage collection
C++	good		Yes	Yes	Yes	
Fortran 77	good					
Fortran 90	good	Yes			Yes	
Haskell	good	Yes			Yes	Yes
Lisp/ CLOS	fair	Yes	Yes	Yes	Yes	Yes
Modula-3	good	Yes	Yes	Yes	Yes	Yes
Sather	good		Yes	Yes	Yes	Yes
SML	fair	Yes		Yes	Yes	Yes

until the program is run, and in creating dynamic data structures such as lists and trees. All reviewed languages except Fortran 77 support dynamic allocation.

4.8 Garbage Collection

Garbage collection is the automatic reclamation of dynamically allocated objects that are no longer in use. With garbage collection, once a dynamically allocated item (an array or object or similar item) is no longer used, automatically the system reclaims the space for use by future allocated items. Without garbage collection, programs that allocate memory without reclaiming it will exhaust all available memory eventually in the heap and fail. Even before that happens, these programs may suffer significant performance penalties due to poor reference locality. The definitions of Haskell, Lisp, Modula-3, Sather, and SML require all implementations of these languages to provide garbage collection. Fortran 77, because it does not support dynamic storage allocation, does not need garbage collection. C++ and Fortran 90 do not provide garbage collection. However, with a good deal of difficulty one can implement garbage collection in C++ by making pointers a class of their own, and defining destructors for pointers that delete the object they point to. This method is sometimes known as "smart pointers," and is described in Coplien [1992]. There are also current research efforts on "conservative garbage collection," to add garbage collection facilities on top of existing C and C++ compilers [Zorn 1993].

4.9 Summary of Miscellaneous Features

See Table III.

5. BENCHMARKS

5.1 Gaussian Elimination on Sparse Matrices

Gaussian elimination is a method for solving for the vector x in the matrix equation $Ax = b$, where A is an n-by-n matrix, and x and b are n-element

column vectors. Here, $n = 1000$. All values used herein are double-precision floating point.

In this test, the A matrices are initialized to random values with roughly two nonzero elements per row. The b vectors were set to all nonzero random values.

In a sparse matrix, only elements having nonzero value are stored. In these tests, there are two versions of implementing sparse matrices: with dynamic allocation or with static allocation.

In the dynamic version a sparse matrix is implemented as a vector having elements 0 through $n - 1$. Each element of the vector represents one row of the sparse matrix. Each such vector element heads a list of pairs, and each pair contains the column number and value of a nonzero sparse matrix element in that row. Element (i, j) is found by accessing the list at vector element i, and traversing the list until finding an $index \geq i$, or the end of the list. If we find $index = i$, we return the associated value; otherwise we return 0. Updates must insert nonzero elements into the lists, and must delete near-zero elements of the lists when, for example, one row is added to another and when some of the sums are near zero.

In the static version, sparse matrices are implemented using four statically (at compile-time) allocated vectors. Each element of vector headvec represents the head of a matrix row. Elements in a row are represented by an index number i.

—column(i) contains the column number associated with index number i
—value(i) contains the double-precision value associated with index number i
—next(i) contains the index number of the next nonzero element on the same matrix row as i, or zero if it is the last on the row.

The dynamic version is simpler and more flexible than the static, since array bounds may be set at run-time instead of compile-time. However since Fortran 77 does not have dynamic allocation, we tested static versions in Fortran 77, Fortran 90, and C++ for comparison purposes.

5.2 Compilation and Execution Times

Timings were performed on a Sun Sparc-2 with 16MB memory that was not on a network and was otherwise idle. Main-storage requirements of all execution time tests were small enough that no memory paging was required; however the compilers did cause paging. For each test situation, three compile times and ten execution times were measured and the means reported. The standard deviation of the measurements was less than 5% of the reported mean in nearly all cases. See Table IV.

Three versions of the Gaussian elimination program were used:

—Version A used dynamic allocation of the sparse memory elements, and included code for subscript checking on all array references.
—Version B used static array allocation, as discussed above, and included subscript checking.

Table IV Compilation and Execution Times

Language	Compiler	Test program version	Compile time not optimized/ optimized (secs)	Exec time, optimized (secs)
C++	CenterLine 2.04	A: dynamic, subchk	14.9 / 16.1	3.23
C++	CenterLine 2.04	B: static, subchk	18.9 / 22.9	7.47
C++	CenterLine 2.04	C: static, no checks	11.4 / 13.0	1.83
C++	Sun 2.0.1	C: static, no checks	9.1 / 17.0	2.82
C++	GNU gcc 2.4.5	C: static, no checks	5.4 / 8.0	2.59
Fortran 77	Sun 2.0.1	C: static, no checks	8.9 / 19.6	2.65
Fortran 90	NAG 2.0a(264)	A: dynamic, subchk	6.7 / 13.6	8.0
Fortran 90	NAG 2.0a(264)	C: static, no checks	6.3 / 12.8	4.1
Haskell	Chalmers 999.5	(Note 1)		
Lisp	CMU 17c	A: dynamic, subchk	0.3 / 12.9 (Note 2)	54.0
Lisp/CLOS	CMU 17c	A: dynamic, subchk	0.43 / 12.1 (Note 2)	54.6
Modula-3	DEC 2.11	A: dynamic, subchk	27.1 / 27.4	3.34
Sather	ICSI 0.5.6	A: dynamic, subchk	17.1 / 16.5	3.99
SML	SML-NJ 1.02	A: dynamic, subchk	19.4 / 20.4	9.25

Note 1. We did not write a Haskell version of the Gaussian elimination benchmark, because of the difficulty in representing state in Haskell. See Section 2.3. In benchmarks on the Chalmers and Yale implementations, not shown here, we found the Haskell implementations to be 50 to 10,000 times slower than C++ or Fortran in array manipulations. Additionally, current Haskell implementations tend to die when handling arrays larger than 20 by 20.

Note 2. Lisp offers both a compiler and an interpreter. The time to read all the source into the interpreter is approximately 1.0 second, making it very fast for debugging work.

—Version C used static array allocation and had no subscript checking.

All versions used 1000-by-1000 arrays with approximately 2 nonzero elements per row.

Discussion. The static version with no subscript checking, while awkward to write and maintain, offered the fastest execution times. The dynamic version, while offering more flexibility, safety, and maintainability, was somewhat slower. The C++, Fortran 77, Modula-3, and Sather implementations had the fastest execution speeds. The Fortran 90 and SML implementations were slightly slower. This may be because they are relatively new

compilers, and work on optimization is progressing. The Lisp compiler produced results significantly slower—most likely because of the run-time type checking in Lisp.

The choice between the faster static version of the program and the more-elegant dynamic version reflects the old trade-off: cost of machine time versus cost of personnel.

5.3 Implementations

See Table V.

5.4 Error Messages

We translated a short program containing a one-character typographical error into each of the review languages. The error caused a type mismatch between the formal and actual parameters to a function. Fortran 77 and Fortran 90 without interface definitions do not catch type errors at all, and Lisp does not catch type errors until run-time. The Lisp message produced at run-time was good. Fortran 90 using interface definitions, C++, Modula-3, and Sather all produced concise and accurate error messages. The message produced by SML was slightly more complex, but still manageable. In the case of Yale Haskell, the error message was even more complicated, and with Chalmers Haskell the error message was so abstruse as to be difficult to decipher.

6. SUMMARY

C++ (CenterLine 2.04, gcc 2.4.5). C++ is based on the imperative and object-oriented paradigms. The C++ language has a large, complex, and poorly integrated set of features, partly as a result of its long heritage. It has fair type checking. It is difficult to extend in some directions, easy in others. C++ offers operator overloading, which can facilitate some applications. The language works well in the object-oriented and imperative paradigms but is poor at list handling. It offers no built-in garbage collection. Existing C++ implementations are mature in many respects and offer among the fastest execution time on all operations not requiring memory allocation. The gcc compiler does not yet have a full implementation of templates; the CenterLine compiler does.

Fortran 77 (Sun Microsystems 1.4). Fortran 77 is an imperative language with a long history, and is the lingua franca for numerical analysis. Implementations of Fortran 77, often by default, have poor type checking. Fortran 77 has none of the common features of new languages: dynamic allocation, user-defined types (records), object-oriented features, first-class functions, arbitrary array origins, exceptions, and modules. However, it does present a clear and easily learned model. Because of the simple model and long history, implementations are mature and offer excellent execution speeds. To the best of the authors' knowledge, there are no freely available compilers for either

Table V. Implementations

	Version	Available from	Ease of use	Inter preter	Docum entation	Error messages (See next section)
C++	2.04	CenterLine Software Cambridge, MA USA	good	no	good	good
C++	gcc 2.4.5	ftp: gatekeeper.dec.com pub/GNU/gcc-2.4.5	good	no	good	good
Fortran 77	2.0.1	Sun Microsystems, Mountain View, CA USA	good	no	good	good (Note 3)
Fortran 90	2.0a(264)	Numerical Algorithms Group, Oxford, U.K.	good	no	good	good (Note 3)
Haskell: Chalmers	999.5	ftp: animal.cs.chalmers.se pub/haskell	fair	yes (Note 1)	poor (Note 2)	poor
Haskell: Yale	"new-cmu"	ftp: nebula.cs.yale.edu pub/haskell/yale	fair	no (Note 1)	poor (Note 2)	poor
Lisp/ CLOS	CMU 17c	ftp: lisp-rt1.slisp.cs.cmu.edu project/clisp-1/release	good	yes	good	good (Note 3)
Modula-3	2.11	ftp: gatekeeper.dec.com pub/DEC/Modula-3/release	good	no	good	good
Sather	0.5.6	ftp: icsi-ftp.berkeley.edu pub/sather	good	no	good	fair
SML	SML-NJ 1.02	ftp: research.att.com dist/ml	good	no (Note 1)	good	fair

Note 1. Although SML and the Yale Haskell do not have separate interpreters they do have incremental compilation, which amounts to nearly the same thing. Chalmers Haskell does have an interpreter, but the syntax it accepts is different from that accepted by the compiler. This certainly makes the Chalmers interpreter less useful than it could be.

Note 2. While the Haskell Report is voluminous, many features are documented sparingly if at all.

Note 3. Since Lisp uses dynamic type checking, many type errors that might be caught at compile-time in a statically typed language are not caught until *run-time* in Lisp. Given that, the error messages in the reviewed Lisp implementation are reasonably good. The Fortran 77 standard requires checking of type and number of arguments; however some implementations do not support this checking. This lack of default checking may result, in part, from the need to compile large legacy Fortran programs that contain such parameter errors. Many Fortran implementations provide compiler options to direct the amount of compile and run-time checking that is performed (e.g., a compiler flag to declare "implicit none" everywhere in the program). Our experience has been that often, however, such checking is not performed by default. Fortran 90 supports more-robust interface declarations and checking by means of the interface block, in which the external interface of a procedure can be explicitly declared. Reported experience with current Fortran 90 implementations indicates that interface block declarations are carefully type checked.

Fortran 77 or Fortran 90. However, f2c, a utility to convert Fortran 77 to C, is available from netlib at Internet ftp address netlib.att.com.

Fortran 90 (*NAG* 2.0*a*(264)). Fortran 90, an extension of Fortran 77, offers additional features including: dynamic allocation, user-defined types (records), modules, optional interface definition, and a host of features to ease dealing with dynamically allocated arrays. As with Fortran 77, implementations are based on a long history and offer good execution speeds and good documentation.

Haskell (*both Chalmers* 998.5 *and Yale versions*). The Haskell language offers strong type checking, is purely functional, and is often elegant. It offers a huge variety of features, including lazy evaluation and arbitrary type classes, and excels in the functional paradigm. Unfortunately, the pure functional paradigm is difficult to use in applications doing input/output and may be inappropriate for algorithms that make updates to large data structures such as arrays. Current implementations of Haskell are immature and can be slow. The immaturity of the implementations also results in weak documentation and poor error messages. Haskell is a large, experimental language with many sophisticated features and may require a significant investment of programmer time and effort in order to become proficient. Whereas many of the features of C++ seem to be afterthoughts and patches to C, the features of Haskell appear better through out and coordinated.

Lisp/CLOS. Like C++, Lisp has a long history and a large variety of features, some poorly integrated with others. Lisp syntax is based on a list notation. While it is sometimes difficult to read, the simple syntax is helpful in many symbolic applications, such as symbolic algebra systems.

Since nearly all binding in Lisp is done at run-time instead of compile-time, it has tremendous flexibility. For example, the language allows creation and evaluation of its own source code at run-time. The downside of run-time binding is that type errors are caught at run-time instead of compile-time. The reviewed Lisp/CLOS system provides a mix of execution speeds: some Lisp functions are nearly as fast as those of C++ and Sather; others are far slower.

Modula-3 (2.07). Modula-3 offers a wide array of fairly well integrated features and strong type checking. Its modularization structure can be useful in large complex systems, but can be a clumsy overhead otherwise. Modula-3 is younger than C++, Fortran, and Lisp, relieving Modula-3 designers of having to accomodate a load of historical baggage. While Haskell's large size comes from inclusion of many experimental features of theoretical interest, Modula-3's size comes from many practical features of interest to "real-world" programmers.

Sather (0.5.6). Sather is a small object-oriented language offering a handful of powerful features and fast execution speed. Many of the algorithms used in Sather's library are almost toy-like in their lack of sophistication, which can cause large execution times in some cases. Sather was designed for

use on small programming projects, and consequently does not scale well to large projects.

SML (*SML-NJ* 1.02). SML is a small, elegant, mostly functional language that offers strong type checking. SML is especially good in the functional paradigm: it supports first-class functions and is powerful and flexible in handling functions and lists. SML's syntax for ordinary imperative programming (updatable variables) is clumsy and difficult to use. The SML-NJ implementation is, in general, easy to use and has good documentation. The SML of New Jersey implementation is relatively mature and offers fairly good execution speed.

7. DISCUSSION

When a person, whether a professional programmer or scientist or general computer user, wishes to use a computer to solve a problem, often he or she has a mental conception, or model, of the problem at hand.

To implement a solution, the person must translate the mental model into a programming language. If the language closely reflects their mental model, the translation is easy. If the language paradigm is different from paradigms used in the mental model, the translation can be difficult. Furthermore, all the tools of formal proofs and verification of correctness apply only *after* completing the translation from mental model to language paradigm. The translation process itself is not verified.

We submit that numerical analysts and programmers in general often become habituated to translating problems into a familiar paradigm—such as the imperative model of Fortran, C, and Pascal. While this is useful for many applications, other paradigms often provide a more-natural and elegant representation of the problem domain.

The various languages reviewed here offer new paradigms and language features that are useful in many situations. Some features associated with the functional paradigm include first-class functions, several types of polymorphism, lazy evaluation, and infinite sequences. Some features associated with the object-oriented paradigm, with its many variants, are new ways to encapsulate data and procedure, code reuse through inheritance, and dynamic dispatching determined by object type.

The benefits of using these languages are better program reliability, greater ease of maintenance, and fewer errors because of:

—Better fit between the language paradigm and the problem domain.
—Better reuse of code.
—Better encapsulation through first-class functions and objects, resulting in less use of the global name space.

The bottom line is, in many cases, faster software development and more-reliable programs, saving costs in both the development and maintenance of programs.

Some of the languages reviewed here are fairly new, giving them both pros and cons. They often have an elegance unencumbered by historical baggage. And their implementations are in some cases immature, showing cryptic error messages, difficult-to-use interfaces, and slow execution times.

The source code shown in the examples in this article may give others an easier start in using some of the reviewed languages. The source for the examples, as well as the programs used to produce the timing statistics, is available by Internet ftp from site ftp.mathcom.com in directory Mathcom/numex.

In summary, the paradigms and languages reviewed offer new and useful ways for approaching tasks in numerical analysis.

ACKNOWLEDGMENTS

John Reppy of the AT&T SML Project and Mark Jones of the Yale Haskell Project gave many helpful suggestions and comments on SML and Haskell. John Reid, associate editor of *ACM TOMS*, and the two anonymous reviewers also provided many helpful comments on the article and sparse matrix benchmarks. Elizabeth Jessup and Dirk Grunwald of the University of Colorado gave helpful suggestions on the presentation. The authors thank Numerical Algorithms Group of Oxford, U.K., and Sun Microsystems of Mountain View, Calif., for their support in providing compilers.

REFERENCES

ANSI. 1978. *American National Standard Programming Language Fortran ANSI X3.9-1978*. ANSI, New York.

BOOCH, G. 1993. *Object Oriented Design with Applications*. 2nd ed. Benjamin Cummings, Redwood City, Calif.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec.), 471–522.

CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. 1992. Modula-3 language definition. *SIGPLAN Not. 27*, 8 (Aug.), 15–43.

COPLIEN, J. O. 1992. *Advanced C++*. Addison-Wesley, Reading, Mass.

DAVIE, A. J. T. 1992. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, Cambridge, Mass.

ECKERSON, W. 1993. Smack dab in the middle. *Netw. World* (June 21), 43.

ELLIS, M. A. AND STROUSTRUP, B. 1990. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass.

HARBISON, S. P. 1992. *Modula-3*. Prentice-Hall, Englewood Cliffs, N.J.

HORN, C. 1993. Standardizing your object interface. *Obj. Mag. 3*, 3 (Sept.), 56–64.

HUDAK, P. 1989. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv. 21*, 3 (Sept.), 359–411.

HUDAK, P., Ed. 1992. Report on the programming language Haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Not. 27*, 5 (May), R1–R164.

HUDAK, P. AND FASEL, J. 1992. A gentle introduction to Haskell. *SIGPLAN Not. 27*, 5 (May), T1–T53.

ISO. 1990. *Fortran 90 Standard ISO / IEC 1539:1991(E)*. ISO, New York.

KEENE, S. E. 1989. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, Mass.

LIPPMAN, S. B. 1991. *C++ Primer*. Addison-Wesley, Reading, Mass.

LOUDEN, K. C. 1993. *Programming Languages Principles and Practice*. PWS-Kent, Boston, Mass.

MARSAGLIA, G. 1993. Remarks on choosing and implementing random number generators. *Commun. ACM 36*, 7 (July), 105–108.

MARSAGLIA, G. AND ZAMAN, A. 1991. A new class of random number generators. *Ann. Appl. Prob. 1*, 3, 462–480.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Mass.

MOBRAY, T J. AND BRANDO, T. 1993. Interoperability and CORBA-based open systems. *Obj. Mag. 3*, 3 (Sept), 50–54.

NICOL, J R., WILKES, C. T., AND MANOLA, F. A. 1993. Object orientation in heterogeneous distributed computing systems. *IEEE Comput. 25*, 6 (June), 57–67.

OMG. 1991 Common object request broker: Architecture and specification Document 91.12.1, Object Management Group, Framingham, Mass.

OMOHUNDRO, S. M. 1990. *The Sather Language*. International Computer Science Inst., Berkeley, Calif.

PARK, S. K. AND MILLER, K. W. 1988. Random number generators: Good ones are hard to find. *Commun. ACM 31*, 10 (Oct.), 1192.

PAULSON, L. C. 1991. *ML for the Working Programmer*. Cambridge University Press, Cambridge, Mass.

PAXSON, V., ARAGON, C., PEGGS, S., SALTMARSH, C., AND SCHACHINGER, L. 1989. A unified approach to building accelerator simulation software for the SSC. In *Proceedings of the 1989 IEEE Particle Accelerator Conference*. IEEE, New York.

PEYTON JONES, S. L. AND WADLER, P. 1993. Imperative functional programming. In the *ACM 20th Principles of Programming Languages*. ACM, New York.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., AND VETTERLING, W. T. 1988. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, Mass

SETHI, R. 1989. *Programming Languages Concepts and Constructs*. Addison-Wesley, Reading, Mass

STEELE, G. L. 1990. *Common Lisp the Language*. 2nd ed. Digital Press, Bedford, Mass.

SULLIVAN, S. J. 1993. Another test for randomness. *Commun. ACM 36*, 7 (July), 108.

WEGNER, P. 1990. Concepts and paradigms of object-oriented programming. *OOPS Mess. 1*, (Aug.), 7–87.

WIRTH, N. 1988. From Modula to Oberon. *Softw. Pract. Exp. 18*, 7 (July), 661–670.

ZORN, B. 1993. The measured cost of conservative garbage collection. *Softw. Pract. Exp. 23*, 7 (July), 733–756

Received May 1993; revised September 1993 and March, May, June 1994; accepted June 1994