Classes and Methods in S
II: Future Directions

John M. Chambers[1]

[1]AT&T Bell Laboratories, Murray Hill, New Jersey 07974 USA

Summary

This is the second of two papers based on talks given at the Second International S Workshop in Toulouse, France. The general theme of the papers is the use of the *class/method* mechanism in the S system. This part describes planned directions for future work on S. In discussing future plans, the emphasis is on those aspects relating to the use of S as a programming environment for statistical computing in the broadest sense; that is, as computations supporting the process of preparing, presenting and analyzing data. Besides a new approach to classes and methods, this will involve plans for compiling S functions, for new data structures and memory management, and for changes in the base language used.

Key words and phrases: Programming, Languages, OOPS.

1 Future Directions for S.

This paper outlines some of the ideas being considered for the next major revision of S, here called Version 4. Our focus is on the use of S as a programming environment and in particular on the theme of classes and methods. The ideas presented here are at an early stage. There is no guarantee that any specific feature will appear as described here, and every reason to believe that at least some will change substantially. These are important and interesting areas for discussion, however, well worth presenting as far in advance as possible. The planned approach is both a substantial change from the present version of S and also distinctive in several features, compared with other languages that support versions of the class/method mechanism.

The programming environment in Version 4 of S will see changes in four main areas:

1. a new view of classes and methods;

2. the translation/compilation of S functions;

3. changes in the base language underlying S;

4. re-design of internal object structure and memory management.

Here we are emphasizing the first of these topics, though all are closely interrelated.

The changes in classes and methods will provide a more precise, flexible, and detailed ability to specify how methods are selected to evaluate the call to an S function. This is desirable for better and more efficient programming in the current interactive evaluation. In addition, however, we will support compilation of functions (specifically, of methods), via translation into the base language.

The assertion that a method can be compiled into code in the base language has implications for the method specification, as will be shown, and also for the semantics of S evaluation and for the choice of base language itself. One goal, often the main goal, of compilation will be to produce an efficient version of the function. To do this, the semantic definition of S evaluation needs to match that of the base language suitably. This will require some revision in argument evaluation, organization of evaluation frames, and basic memory management if the compiled code is to approach the efficiency of hand-coding in the base language. For example, lazy evaluation of arguments in full generality and the explicit construction of evaluation frames for each function call represent substantial overhead. Compilation will, at least optionally, provide for a function-call semantics closer to a language like C. For many situations, the loss of full S semantics will make no difference. Many of the *uses* of lazy evaluation and frames can be accommodated in the compiled code. When that is not sufficient, we anticipate an option to compile with full current semantics. Choosing this option will produce somewhat less efficient compiled code.

S has always been based on support code written in a "base language", originally Fortran and later C. The syntax and semantics of S are not restricted directly by those of the base language — this is the main distinction implied by saying that C code *underlies* current S, rather than that S is implemented *in* C. Implementing S requires choosing a base language and then making it conform to S semantics via data structures, memory management, and other tools. For the next version of S, the tools and possibly the language itself will change.

There are a number of possible strategies, some quite radical. The two most likely scenarios are to continue with C, but with a much revised set of tools, or to adopt C++ in order to use its class/method features. The potential advantages of using an object-oriented base language are enhanced simplicity and efficiency if the base language is able to implement features more effectively than we could manage by writing our own versions. An analogy in previous work was the conversion, between Versions 2 and 3, from our own data structure mechanisms built largely on Fortran to a relatively direct use of C language structures. Overall, the change made for both simpler and more efficient code.

The main disadvantage of adopting such a language is that the view of classes and methods in S does not exactly correspond, for a variety of reasons, to any of the main candidate languages (see section 7). A second, practical problem is that adopting a new base language is likely to increase the amount of rewriting needed for the new version. (A counter argument that proponents of such languages would make is that the rewriting should be made easier and less error-prone by the features of the language.) Current work is proceeding to implement the new version of S in S itself to a detailed level. This will allow us to test the essential features of the new version before making a decision about the base language.

2 Classes and Methods.

On balance, the use of the class/method mechanism has been notably helpful in the recent work with S. Considering our experience and various other approaches to classes and methods suggests several directions for change.

- The information about available methods, about object classes, and about generic functions should be explicitly available. We will refer to this information as the *methods definition.* It will be used by the S evaluator and by the compiler, and will also be a source of information for users.

- Methods definitions will be part of new projects implemented in S. There must be a convenient mechanism for developers to use and modify the definitions for evolving collections of S software.

- The class/method mechanism needs to be uniformly applied for all S code, in the sense that natural intuition about classes in the informal sense of Becker, Chambers and Wilks (1988, chapter 8) needs to be made precise and useful using the new mechanism.

- There needs to be a more precise and flexible way to guide the choice of a method than in the current version.

- As work progresses on the other areas for future development, the class/method mechanism needs to be kept consistent with that work.

- The selection of the method to be used to evaluate a call to an S function must be efficient, since this operation occurs so frequently.

Issues like improving the efficiency of the search for the right method are important practical aspects of designing the new version of S. At a more conceptual level, the main goal is to provide a clear and expressive mechanism that can increase the effectiveness of S as a programming environment. The details will evolve over time; however, a broad view can be sketched at this stage. In doing so, we hope to stimulate comments and suggestions.

3 Classes, Methods, and Databases.

The approach to classes and methods in Version 3 of S provides no central information about their organization. Information about class behavior comes from the class attribute of individual objects, while information about available methods is obtained by searching for functions named by concatenating generic function and class names. The latter, especially, was a choice made to keep things simple and is not ideal either for efficiency or for clarity. For the next version, we will provide a more explicit way of specifying the choice of method as well as other related information about classes, the *methods definition*. Its organization will be discussed in the remainder of this section. The information in the definition and the way this will be specified in S are discussed in sections 5 and 6.

The organization of information about classes and methods is a central issue in all systems that use these concepts. S has some long-standing characteristics that need to be considered in planning this organization. The use of separate, persistent assignments in S has led to the notion of databases as the fundamental organizing entity for S objects. In particular, S *libraries* are databases that organize S functions according to their purpose (images, maps, statistical modeling, etc.). In Version 4, a library database will contain special information, the methods definition, about the methods and related objects contained in that database. The S evaluator will merge the information from all the databases on the current search list to define the method selection rules. These rules will be used in interactive evaluation and also to analyze expression semantics (e.g., as part of compilation).

Localizing the methods definition at the level of S databases serves several of the goals in section 2. The S evaluator will have complete information on the currently available methods. The contents of this information, at the database level, will change only occasionally. Therefore, substantial effort is justified in preprocessing the information into a form that can be used efficiently in selecting a method. The merging of information from the currently attached databases will require less computing, but even this need not be done that often. Only when a database that contains a methods definition is attached or detached does the information need to be changed. Frequently attached databases containing only data for analysis will make no change. Also, the current plan is to require that the methods definition remain constant throughout a single top-level S expression. Therefore, temporarily attached databases will not require re-merging of the database information.

Localizing the methods definition by databases makes development of new class/method information straightforward. Developers will naturally aim at creating an S library database containing their new software. This library will contain a methods definition for the project; that is, information about method selection, generic functions, and class properties. The same approach will work when revising existing S libraries, since methods definitions are merged so that earlier databases on the search list override later ones.

There are a number of substantial questions to be answered. How should the methods definition be viewed by the programmer? What should the internal format of the methods definition be — what sort of tables, how complete and how organized? Who has responsibility for ensuring that the merged definitions are complete?

The point of the third question will become clearer when we discuss what information is to be stored in the definition. On the second question, details will need to be revised many times as the new version evolves. It is the first question that needs most discussion at the present stage. This paper presents some current ideas.

4 Specifying Classes and Generic Functions.

In Version 4 of S, both classes and generic functions will be explicitly specified. Classes and generic functions will themselves be objects (typically of class `CLASS` and `GENERIC` respectively). We will sketch very briefly how such objects are organized, as background for our discussion of method definition, the main topic of this paper.

A class object is specified by a name and, optionally, the representation for objects belonging to the class. A representation typically consists of some named *slots*, each slot being specified to belong to a particular class. Slots can be specified directly or by including other classes in the representation.

```
dataFrame <- setClass("dataFrame",
    representation(Data = Object, row.names = Character))
```

```
matrix <- setClass("matrix", representation(array))
```

The first example specifies class `dataFrame`, with a representation having two slots, of class `Object` and `Character`. The second example specifies a class `matrix` whose representation includes (and, in this case, is identical to) that of class `array`. Other versions of class definitions allow for unstructured classes, such as numeric vectors, and for virtual classes, which will have no actual objects belonging to them.

Functions for which methods are to be defined are specified to be generic:

```
plot <- setGeneric("plot", function(x, y, ...)UseMethod())
```

The specification for calls to `plot()` defines the formal arguments by the function definition supplied. Typically, the body of the generic function does no computation other than selecting the method, as is the case here.

A useful feature of the current class/method mechanism in S is that generic functions can be organized into *groups*. Methods defined for a group generic will be used for generic functions that are members of the group, if no specific method definition overrides. The group mechanism will be carried over to the new version, where group membership can be included in the specification of a generic function:

```
Plus <- setGeneric("+", function(e1, e2)UseMethod(), Arith)
```

This specifies the generic function for the operator, `"+"`, and says that it belongs to group `Arith`.

5 The Methods Definition.

The central task for the S evaluator in implementing the class/method mechanism is to take a call to a generic function and determine the method corresponding to the actual arguments. This method selection procedure will be used for interactive evaluation, and also for semantic analysis and for compilation of methods into the base language. The methods definition will be used to match actual arguments to available methods. A generic function can be matched to a method in several ways. The most important is by a *method signature*.

A signature is a named vector of classes; a method signature associates each of these classes with a formal argument of a generic function. These signatures can be used when specifying methods. Consider the following specification of a method.

```
setMethod(plot, signature(x = numeric, y = numeric), xyscatter)
```

The arguments are a generic function, a signature and a function definition to use as a method when the actual arguments match the clases in the signature.

More precisely, the specification will result in a version of the method being stored in the library (after semantic analysis). When a call to `plot()` is being analyzed or evaluated, the classes of the actual arguments will be compared to the signatures in the methods definition to see if any match. The precise meaning of "match" in this context is an important and subtle question. The methods definition contains tables relating pairs of classes, stating whether an actual argument of class $\mathcal{B}$, say, can be used for a method signature having class $\mathcal{A}$ for the corresponding formal argument. This is possible if either the classes are equal or if $\mathcal{B}$ can be substituted for $\mathcal{A}$, in a sense to be discussed in section 6. Comparing all the relevant arguments will produce 0, 1, or more potential methods. How to handle more than one matching method is a subject of controversy in the design of languages using methods. Currently, the design of S uses a measure of closeness of association (part of the methods definition) to disambiguate the method choice.

A method signature can require an argument to be missing, by supplying it as class `MISSING`. Thus, if `Plus` is the generic function for `"+"` as above,

```
signature(e1 = numeric, e2 = MISSING)
```

is a signature for the unary version of the operator. In contrast, formal arguments not appearing in the signature are taken to have class `ANY`, meaning that any actual argument will match them.

Matching methods by signature will be the standard technique for selecting a method. However, it is not sufficiently general for all requirements in S. In particular, it is not suited to functions taking an arbitrary number of arguments, such as `c()` or `cbind()`. Consider a typical example. Suppose `cbindMatrix()` is a function suitable to bind together S vectors or matrices as columns of a matrix. This is a method for `cbind()`, but its conditions for applicability cannot be specified as a signature of the form shown so far. In words, `cbindMatrix` is a suitable method if all the arguments to `cbind()` are of class `vector` or `matrix` (or of classes equivalent to these classes).

More general method selection of this sort will be handled in S by defining S functions to test explicitly for the applicability of one or more methods. The mechanism as currently planned works as follows. A specification nominates one or more test functions; e.g.,

```
setTestMethod(cbind, canCbindMatrix, cbindMatrix)
```

This specifies a function `canCbindMatrix()` to test for methods for the generic function `cbind()`. The function expects to be called with one argument, `classes`, the signature of the classes of all the actual arguments in the call to `cbind()`. The test function `canCbindMatrix()` returns `TRUE` if the corresponding method should be called. For example, the condition for `cbindMatrix` to be applicable could be expressed in S by the test:

```
all(is(classes, vector) | is(classes, matrix))
```

where `is()` expresses equivalence of classes as discussed in section 6. There can be any number of method tests for a particular generic, with a mechanism (not shown) to name them so they can be updated. The test functions are called until one selects a method.

Default methods can also be supplied, as in Version 3:

```
setMethod(plot, plot.default)
```

says to use `plot.default()` if no method can be matched otherwise. Default methods are somewhat deprecated; usually, they either spend much time in selecting the actual method to use or they are in fact only suitable for a small range of signatures. Making the cases handled explicit by separate signatures both speeds up selection and ensures that applying the generic to some unexpected class is caught and reported as an error. The same method definition can be specified for several signatures. Semantic analysis can often adapt the definition efficiently to each of the cases.

The full method selection process as currently envisioned is then as follows.

1. The signature of the actual arguments is matched against the table of signatures specified for methods. If there is a possible match or a single best method of several matches, this method is chosen.

2. If there are any test method functions, these are called until one of them selects a method, if any do.

3. Finally, if there is a default method and no method matches by signature or test, the default method is chosen.

Signature match will likely be the predominant way to choose a method. It is both the most efficient and the simplest to understand. The increased generality provided by test functions for methods, however, is an important addition to method selection. Without it, methods cannot be defined effectively for functions taking an arbitrary number of arguments.

6 Inter-Class Information.

Much controversy about object-oriented programming systems concerns inter-class relationships. What are the natural concepts relating classes and what are the implications for defining methods? The term *inheritance* is often used when a class $\mathcal{B}$ in some sense is an extension or derivation from another class $\mathcal{A}$. However, there is a wide range of relationships in applications. Different systems give different shadings to the relationships, with the confusion sometimes compounded by inconsistent choice of terminology.

Version 3 of S used a casual approach to inheritance. An object's class could contain more than one name. The first name was the object's principle class and remaining names indicated classes from which the object inherited. The first method found (by matching object names) corresponding to any of these classes would be selected. This was a simple idea that served to provide for considerable automatic re-use of software, as shown in Part I of this paper. However, inheritance in this form was not a precise concept and could easily be overused.

Version 4 of S uses a more precise notion, *object substitution*, represented by the S function `is()`. An object b of class $\mathcal{B}$ `is` (can be substituted for) an object of class $\mathcal{A}$ if any computation applicable to class $\mathcal{A}$ — in particular any method for class $\mathcal{A}$ — can be applied substituting the object b where the class $\mathcal{A}$ object was required. As an example, consider matrices and general multi-way arrays in S. Intuitively, any matrix `is` an array: Whatever array computations we do can be applied to the matrix. This information may be specified by an `is` assignment:

```
is(matrix, array) <- TRUE
```

This says that any object of class `matrix` may be substituted when an object of class `array` is wanted. Notice that substitution in the other direction may or may not make sense. In this example, an array is sometimes a matrix, but the substitution must be tested. We can express this by an `is` assignment as well. If the right side is a function, *conditional substitution* is allowed if the function evaluates to `TRUE`. In this example:

```
is(array, matrix) <- function(x)length(dim(x))==2
```

The function on the right side of the assignment is stored and called to test a possible substitution of `matrix` for `array`. The `is` assignment may also be used to prohibit a substitution:

```
is(matrix, ThreeWay) <- FALSE
```

Substitution is transitive; for example,

```
is(array, structure) <- TRUE
```

along with the previous assignments, implies that a `matrix` object can be substituted where an object from class `structure` was specified. Transitivity plus conditional substitution can lead to some unwanted connections. Whenever a class has mutually exclusive specializations, there may appear to be a conditional substitution between the specializations, unless we explicitly prohibit it. The substitution will always fail when tested, but wastes time in identifying methods. If someone decided to make 3-way arrays a class, `ThreeWay`, then such a path might appear to exist from `matrix` via `array` to `ThreeWay` unless prohibited as above.

Substitution relations often arise automatically from class representations. If class $\mathcal{B}$ includes class $\mathcal{A}$ in its representation, then $\mathcal{B}$ substitutes for $\mathcal{A}$ automatically. Thus, the relation `is(matrix, array)` need not have been specified explicitly. The substitution implied is trivial to carry out if the target class representation is contained in the current class representation. If not, a conversion function can be supplied in the specification:

```
is(sequence, numeric, expd.sequence) <- TRUE
```

says to call the function `expd.sequence()` to substitute a `sequence` object where a `numeric` is wanted. The notion here is that of representing, say, the result of the expression `1:1000` by the parameters of the sequence, not the entire set of numbers. Then an object of class `sequence` can be substituted without ambiguity wherever a `numeric` is wanted, but to deliver the numeric we need to call the conversion function `expd.sequence()`. This is also an example where substitution cannot be inferred automatically from the representation.

Substitution information is assigned in terms of pairs of classes. In our prototype code this is expanded, however, to include a table of all possible class substitutions when transitivity is applied. As noted, the precomputation will pay for itself since method selection is performed vastly more often than class relations are changed.

The value of the "Keep it simple" philosophy in class design needs to be emphasized, of course. While the concepts provided here allow very complicated interrelationships to be built up among classes, complication brings a penalty in efficiency and, more seriously, in confusion. Like most good tools, the class/method mechanism can be dangerous if carelessly used.

7 Comparisons with Other Systems.

In both C++ and CLOS, classes are related to each other in terms of their representation. The two systems use different terminology for inclusion — C++ says that $\mathcal{B}$ is a derived class of $\mathcal{A}$, CLOS says that $\mathcal{B}$ is a subclass of $\mathcal{A}$. In both systems, a class can include more than one other class, and inclusion implies expansion; for example, if $\mathcal{C}$ explicitly includes $\mathcal{B}$, it also implicitly includes $\mathcal{A}$.

However, inclusion is not entirely satisfactory as a mechanism for specifying how classes are related conceptually. Clearly, an included $\mathcal{A}$ object can be extracted from a $\mathcal{B}$ object, so that one could always mechanically apply a method for class $\mathcal{A}$ to an object from class $\mathcal{B}$. There are at least three kinds of problem that arise, however. First, inclusion offers no way to incorporate class constraints that are not reflected in the representation, such as the relation from `array` to `matrix`. Second, substitution can apply even though the representations are entirely different, as in the `sequence` example in section 6. Third, without finer control such as provided by the two-way relation between `matrix` and `array`, some intuitively natural operations cannot be accommodated.

To illustrate this last point, suppose some method operates on an array to return an array. This method can be inherited by class `matrix`, but it will not return a matrix, even though the resulting array may in fact be equivalent to one. This is reasonable enough, since class `array` isn't supposed to know about all the classes that might specialize it. However, the inclusion relation leaves no mechanism for back-substituting the returned array so that it can be used automatically as a matrix. As an example, suppose `aperm()` in its usual definition is an array method and `solve()` is a matrix method. Then

```
solve(aperm(X, i))
```

will fail because inclusion will not automatically make an array into a matrix. Using the `is` specification as we did in the previous section, however, the back conversion will succeed and all will go as it intuitively should. At the same time, the syntactically equivalent

```
solve(outer(X, X))
```

will fail, as it should, because the array returned by `outer()` will not be a matrix.

Both CLOS and C++ have method-matching mechanisms with some similarity to that of section 5. CLOS calls method signatures with more than one argument multi-methods; classes match if the actual argument class includes the corresponding class in the method signature. Like S, but unlike C++, CLOS has the notion that not all arguments need be included in the specification, in which case any argument matches. C++ requires that one method dominate all the possible matches (be at least as good for every argument and better for some), otherwise the match is termed ambiguous and is treated as an error. See Keene (1989, section 4.3) and Ellis and Stroustrup (1990, section 13.2) for further discussion. Method selection by signature is somewhat more flexible in S; also, the test functions that can be used to extend method matching in S have no analogues in CLOS or C++. This becomes particularly important for functions that take arbitrarily many arguments. C++ provides for such functions, (e.g, Ellis and Stroustrup 1990, section 8.3), but actual arguments matching `"..."` match a method signature regardless of class (Ellis and Stroustrup 1990, page 317) [2].

[2] ANSI C and C++ adopted (presumably coincidentally) the notation `"..."` used in S, but as a more restricted mechanism, *ellipsis*, than S provides.

8 Conclusions.

The object-oriented programming paradigm, or the class/method mechanism to use the more modest term preferred here, is a controversial but, on the whole, valuable mechanism for organizing substantial programming tasks. S has traditionally emphasized exploratory, non-repetitive use. Users should be encouraged to experiment, which implies that they must be able to program new ideas easily and effectively. For substantial projects, the class/method mechanism has been shown to aid the programmer to get a good implementation quickly. Therefore, the class/method mechanism is a natural approach for S. Beginning with the revision of Version 3 to accommodate modeling software, the mechanism has been brought into S, but gradually so as to retain existing good features of the system and customize the mechanism suitably for S.

The results of recent projects have confirmed that the mechanism is useful for the style and applications typical of S. Future plans call for substantial extension of the mechanism, in directions that should allow convenient and precise use of classes and methods in substantial programming projects.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*, Wadsworth, Pacific Grove, California.

Chambers, J. M. (1991) Data Management in S, *AT&T Statistics Research Report No. 99.*

Chambers, J. M. and Hastie, T. J. (1992) *Statistical Models in S*, Wadsworth, Pacific Grove, California

Ellis, M. A. and Stroustrup, B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley, Reading.

Keene, S. E. (1989) *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*, Addison-Wesley, Reading.