Developments in Class Inheritance and Method Selection

John M. Chambers

June 2, 2009

This paper describes some recent developments in the R mechanisms for defining new classes and for selecting methods. The overall goal is to provide more flexible, unified and consistent use of these programming tools. Most of the changes relate to combined use of S4 classes, S3 classes and basic object types. In particular, modifications allow the use of both S3 classes and all basic object types as superclasses contained in new S4 classes. The S3 method selection mechanism has also been adapted for objects from S4 classes. Tests for inheritance (the `is()` and `inherits()` functions) have been made aware of both class mechanisms. Within the S4 class and method model itself, a consistent ordering of superclasses and a criterion for unambiguous selection of methods are imposed, when possible.

1 The Starting Point

The developments described in this paper form the main extensions of the R class and method mechanism since mid-2008 (starting with version 2.8.0 of R), and can be read as an update to the discussion in Chapters 8 and 9 of Chambers (2008). This section summarizes the implementation of class representation and method selection as of version 2.7.0 of R. In the remainder of the paper, developments in these areas are summarized. Sections 2 to 4 discuss extensions to class definition to inherit from S3 classes and arbitrary object types, and to select from mixtures of S3 and S4 methods. These sections contain the new programming features you may want to use. The remaining sections describe internal changes in S4 class and method definitions to provide a more explicit and consistent selection. Those concerned with packages that have complicated or extensive patterns of class inheritance should understand the rules; otherwise, the material is mainly intended as part of the R system definition.

S4 objects—that is, objects generated by a call to `new()` for an S4 class—are represented internally by the same C structure as all R objects. They can be distinguished in either R or C programming by checking a special bit in the structure, by calling the R function `isS4()` or the C macro `IS_S4_OBJECT()`.

As with other R objects, the interpretation of the data in the object is controlled by the type field of the structure (`typeof()` or `TYPEOF()`). A specific

type, "S4", is used for general S4 objects. However, if a class definition contains, directly or indirectly, one of the basic object types such as "numeric" or "function", one of the pseudo-classes "matrix" or "array", or the virtual class "vector", then the type of the S4 object is determined by the inherited basic type or by the data supplied when the specific object is created. To give three simplistic examples:

```
setClass("c1", contains = "character")
 x <- new("c1", "Testing")
setClass("c2", contains = "matrix")
 x <- new("c2", matrix(0.5, 3, 4))
setClass("c3", contains = "vector")
 x <- new("c3", (1:10) > 5)
```

The type of x in the three examples will be "character", "numeric", and "logical".

The basic types that can be inherited directly are limited to those that are treated as normal R objects, in the sense that they are duplicated on request, to prevent side-effects from functional computations. Certain object types are not duplicated and these can not be used directly to represent S4 objects; the types "environment", "externalptr" and "name" (symbol) are examples. Because objects of these types are not duplicated, slots or attributes cannot be added to them without the side effect of appearing in all instances of the object. For this reason, such object types were prohibited as superclasses in S4 class definitions. A goal of recent changes has been to allow these object types as superclasses by an indirect mechanism, while retaining "S4" as the object type.

The online R documentation (as well as the discussion in Chambers (2008)) also discouraged programmers from defining classes that contained S3 classes. S3 classes have no formal definition with respect to content but can and should be registered via a call to setOldClass(). Once registered, they can appear formally anywhere a class name is specified—as a slot in a definition, as part of a method signature or as a superclass. The last use was not well supported. S4 computations would interpret an object from such a class as inheriting from the S3 class, but S3 method selection, for example, would not recognize the inheritance. Even if selected, the S3 methods would not recognize the object, meaning that computations using the class information directly could fail. Also, computations that copied and changed the object would quite likely create an invalid object from the S4 class. A second goal of the changes was to make S3 superclasses work as well as possible, in particular to have S3 method selection aware of the superclasses and to supply an object to the selected method that was more likely to be treated as a legitimate object from the S3 class.

Class inheritance also influences programming through its use in selecting a method for a function call based on the classes of the arguments. In all languages with such method selection, the goal is to choose a method either directly defined for the classes supplied or, if no such method exists, one defined for as "close" a combination of classes as possible. Method selection in the S4 system in R is relatively complicated for two reasons. R allows *multiple*

inheritance, so that the actual class may inherit superclasses from more than one direct superclass. And, being a functional language, R naturally supports *multiple dispatch*, that is, selection of a method based on the class of more than one argument. Both of these lead to some ambiguities about the best selection, which recent developments have tried to reduce.

2 Inheritance from S3 Classes

The S3 class and method software dates back to the work on statistical models in S (Chambers and Hastie, 1992) and to early versions of R. Technically, the software implements an *instance-based* object system, in that methods are selected by looking at the character strings in `class(x)` for an individual object `x`, without requiring that any of these corresponds to a class definition from which `x` was created. Merging S3 class software with formal classes and methods is therefore an unpredictable activity. Programmers can use class strings in arbitrary ways, and have done so. The original introduction of the S4 model came with the hope that the major applications using S3 classes would be converted, as assumed in Appendix B of Chambers (1998), for example.

A decade later, there is instead a much larger body of software using S3 classes, most of which is unlikely to migrate to formal classes and methods. To build on this software while using more modern tools needs a mechanism to deal with S3 classes and methods in a reasonably general and convenient way. The developments described in this section attempt to move towards such a mechanism. S3 classes can be registered, including their S3 inheritance patterns, assuming they have such patterns consistently. Optimists may even define the effective slot structure of the class, again assuming that makes sense (examples are discussed below). S4 class definitions can extend registered S3 classes, as well as specifying them as slots or in method signatures. Changes to the internal R software are gradually attempting to manage combinations of S4 and S3 classes reasonably, in terms of method dispatch and object interpretation.

The base of the mechanism is the registration of the S3 class pattern by a call to `setOldClass()`:

```
setOldClass(Classes)
```

where `Classes` is the character vector of class names expected in corresponding objects. For example, `"data.frame"` registers the single name while `c("ordered", "factor")` says that objects are expected having these two strings in this order as their `"class"` attribute, implying that S3 class `"ordered"` inherits from `"factor"`. All the strings will have S4 class definitions as a result, reflecting the implied inheritance and all extending `"oldClass"`. Optional arguments to `setOldClass()` allow for additional information about corresponding objects and also try to handle some of the inconsistencies that arise. See the documentation, `?setOldClass`, for details.

The single-argument call to `setOldClass()` creates an S4 proxy class that is virtual, with no information about the contents of objects carrying this class.

For some S3 classes, one may assert that the objects contain attributes of known name and class, which can then be specified as slots, because R implements slots as attributes (not coincidentally). For example, if we are willing to assert that all classes of S3 class `"data.frame"` have type `"list"` and attributes `"names"` and `"row.names"`, and if the attributes themselves can be asserted to be of a known class, then all data frame objects will be equivalent to objects from a well-defined S4 class. This will allow us to apply tools to these objects and will have extended the S3 class in a useful way. This particular example illustrates typical difficulties however. The list data and the (character) names attribute are straightforward, but what R expects for the `"row.names"` attribute has actually changed over time: originally required to be `"character"`, they can now be `"integer"`, and are by default. Assuming that this decision will stick for a while, we have defined a class union containing `"character"` and `"integer"`, and specified this as the class for an S4 version of `"data.frame"`.

Not all S3 classes use attributes, however. For example, `"lm"` and other classes of fitted models use lists that are asserted to have components of known name (and sometimes of known class). The current software for registering S3 classes does not provide a formal way to specify components of a list, but that is a possible future extension.

Any object from a class that extends `"oldClass"` has a slot `".S3Class"`, intended to contain the character vector that should be used as `class(x)` by S3 class and method computations. This slot will be found in any object from a class that extends a registered S3 class. If no other action is taken, the slot will contain the S3 class or classes implied by the call to `setOldClass()`. For special requirements, the slot can be used to pass an arbitrary S3 class down. For example, some computations use S3 classes inconsistently, so that different objects apparently from the same main class have different inheritance (see `?POSIXt`, for example). While the S4 class must have consistent inheritance, objects from the class could reflect the differences via the `".S3Class"` slot.

3 Inheritance from Object Types

The object types in R have corresponding formal classes (although `"numeric"` replaces `"double"` and `"name"` is used instead of `"symbol"`, mainly for compatibility with S). Formal classes in R have always had the ability to inherit from one of the types provided that the type behaves "normally". All the vector types as well as the types for functions and primitives and type `"language"` are included. Not allowed were types such as `"environment"`, `"symbol"`, and `"externalptr"`. The prohibition followed from the implementation of the inheritance, which in turn was motivated by making the inheritance useful.

If an S4 class contains a normal object type then objects from the class will have this type. The following definition for class `"cText"` defines the object type `"character"` as a superclass:

```
setClass("cText",
  contains = "character",
```

```
    representation(count = "integer"))
```

Objects from class "cText" will have type "character". The utility of this implementation is that computations for standard functions and operators written for the basic type can be inherited immediately by the new class with no explicit transformation of the objects. These computations are often implemented in base C code and so are relatively efficient. The slots of the S4 object, implemented as attributes, may or may not be treated appropriately for the meaning of the new class. Fairly often they are, but if not the method required to correct the problem typically uses the base computations and then modifies the result to suit the class.

However, objects from a class that extends one of the abnormal types can not have that type precisely because attributes do not work for them, which in turn results because objects of these types are effectively *references* to some information rather than local objects in the standard R and S sense. For example, an object of type "environment" is a reference to a single internal structure. Any function that modifies the object by assigning or removing objects from the environment changes that environment for all functions that are using it. For type "symbol" the issue is one of efficiency. Any character string used as a symbol is installed in an internal table, so that symbol equality can be tested by comparing the reference rather than the contents of the string. If objects from a class extending one of these types had the same type, then assigning an attribute (such as the class attribute itself) would overwrite the attribute for any other "copy" of that object.

Classes containing one of the abnormal types are now implemented by using a reserved slot name to hold the "data" part, while leaving the actual type as "S4". The definition of the new subclass can appear just as it would for extending a normal type; for example,

```
setClass("datedEnv",
  contains = "environment",
  representation(lastAssign = "POSIXct"))
```

Objects from class "datedEnv" can be coerced to "environment", by either S4 or S3 computations. As with classes that contain a normal object type, objects are eligible for S4 methods that have "environment" in their signature, without explicit coercion.

The internal computations to pass the object to the method are different however. Subclasses that are defined by sharing slots or containing a normal object type are "simple" extensions in R terminology. An object from a simple subclass of a normal type can usually be passed with no modification to a method or function that expects an object of that type. The data corresponding to the type is in the object in the expected form. With a non-simple subclass, the method dispatch software will explicitly compute an object from the type and pass this to the selected method.

In addition, the *programmer* may need to coerce the object by a call to as() before passing it to a low-level function that expects an object of the corresponding type. For example, if e1 is an object from class "datedEnv", to use

it as the `envir` argument to function `exists()`, which expects an environment, the programmer needs to do the coercion:

```
exists(what, envir = as(e1, "environment"))
```

In this case, somewhat perversely, if `e1` was supplied as the general `where` argument rather than the "smarter" choice of the `envir` argument, the default expression for `envir` would evaluate

```
as.environment(where)
```

and get the desired answer.

Basic R or C code that expects an object from one of the abnormal types can be modified simply to work for S4 subclasses with little overhead for non-S4 objects. In a function requiring that object `x` is of type `"environment"`, for example, the simplest patch is of the form:

```
if(isS4(x))
  x <- as(x, "environment")
```

A similar computation applies in C code:

```
If(IS_S4_OBJECT(x))
  x =  R_getS4DataSlot(x, ENVSXP)
```

Including explicit conversions of such S4 objects has a negligible efficiency penalty for most applications, since computations are all conditional on testing the S4 bit in the object. No valid S4 object can be used as an abnormal type unless it inherits from the corresponding class, meaning that the coercion will fail as it should for other S4 objects. Depending on the context, the test could be expanded to give a more informative error message for invalid S4 classes, all still conditional on the `isS4()` test.

Much of the base code underlying primitive and internal functions does in fact include such a test and coercion to the desired object type, with knowledge of the S4 mechanism.

4 Combining S4 and S3 Methods

Once the notion of combining the two class paradigms is accepted, some strategy is needed to select and dispatch methods in a mixed situation. This arises in both directions, defining new (S4) methods for old classes and applying S3 methods to objects from S4 classes. In both cases, the central issue is how to apply a suitable selection strategy, as well as worrying about whether the selected method will behave correctly for an object from a "foreign" class.

S4 methods for S3 classes are relatively straightforward, so long as the class has been registered. The S4 inheritance is defined by the registration, via the call to `setOldClass()`. For normal S3 classes that follows the expected pattern; for example, an object from the S3 class `"mts"` would match signatures containing `"mts"` and `"ts"` in that order, as with S3 method selection. The major

caveat is that S4 inheritance is always consistent, determined by the class definitions, whereas S3 selection is instance-based. So, for example, S3 objects with main class `"POSIXt"` can differ in the second class string and select different S3 methods accordingly. The corresponding S4 class has a conditional inheritance, but its method selection will not use that information. Some workaround would be needed, such as noted in (Chambers, 2008, page 369). These cases are rare, however, and otherwise the approach required is relatively clear.

In the other direction, S3 method selection is instance-based, using the class attribute of the object to select the best-matching method. Therefore, we have introduced the `S3Class()` mechanism, as described in section 2, to define the classes that should be used for S3 method selection for an S4 object when appropriate. Appropriate objects include any from an S4 class that contains an S3 class, as discussed in section 2. Method selection is now reasonably well supported here, although there are some details to consider, the basic approach is to set the `".S3Class"` slot to match the S3 class contained.

In principle, one might choose to allow S3 methods only for S3 classes, using the mechanism to apply the methods to S4 objects. There are practical reasons not to do so, as well as the generally permissive philosophy of R.

Two practical issues are important in this case. First, a number of S3 methods have been written for S4 classes that have no relationship to S3 classes. The original S4 design tacitly assumed no one would do this, since it was obvious that S3 code would know nothing about S4 inheritance. Unfortunately, the chance to make such definitions illegal slipped by the initial R implementation, partly because it was not easy at first to identify S4 objects. And, by a fundamental principle of user-oriented software, if something is possible someone will do it. So, by the time the problem was recognized, a number of packages had invested significantly in S3 methods for S4 classes.

Second, the current implementation of R does not recognize S4 methods in calls that take place directly from the base namespace. Currently this limitation requires writing S3 methods for S3 generic functions that are called from other functions in the base namespace. See the example at the end of this section. The requirement does not apply to primitive functions, which use a uniform method dispatch mechanism regardless of how they are called.

For these reasons, a mechanism has been provided to register the intention to write S3 methods for an S4 class when the class is created, by including the argument `S3methods = TRUE` in the call to `setClass()`. With this option, S4 subclasses of that class will also select the corresponding S3 methods for that class, if no better-matching method exists. The mechanism implementing the policy is to add the same `".S3Class"` slot to the object, but now set to the vector of eligible S4 classes for method selection, that is, to the value of `extends()` for this class. The S3 method selection uses this slot as in the previous case, making S3 method functions for this class and its superclasses eligible. Also, subclasses of the class will inherit this slot and so will select S3 methods similarly, but not for the subclass itself, unless that is also registered by `S3methods = TRUE`.

What about S4 classes that are not registered and do not extend S3 classes? The long-term intention is that no non-default S3 methods will be selected for

these classes, to avoid accidental conflicts with the S3 naming convention. That there happens to be, say, a function `print.foo()` somewhere should not cause problems for a programmer defining an unrelated S4 class `"foo"`. Restricting methods to those defined explicitly was an obvious and central part of the S4 design. In the interim, not to break too much existing code, S3 method selection will still select the method directly for objects from such a class, but not for objects from its S4 subclasses. Future plans include a tool to detect and report apparent S3 methods for non-registered S4 classes.

Even if registered, S3 methods for arbitrary S4 classes are somewhat deprecated because they can create unintuitive method selections. An S3 method for a class, even if an exact match or a direct superclass, will fail to be chosen in S4 method selection if there is any candidate S4 method, no matter how indirectly related, because in the usual situation all S3 methods are chosen by the default S4 method, typically the previous S3 generic function.

Turning back to methods for inherited S3 classes, method selection should now work without restrictions, although special cases such as the `"POSIXt"` example may need some programmer intervention. The use of a per-object slot to hold the S3 class information makes the full generality of instanced-based selection available, provided the programmer takes the step of setting the slot. The S3 `"POSIXt"` behavior can be obtained, as long as the programmer uses the `S3Class` assignment function to assign the appropriate S3 class for the individual object; for example,

```
S3Class(x) <- c("POSIXt", "POSIXlt")
```

There is another consideration in applying S3 methods however: will the S3 method work as intended on the S4 object? Usually yes, but some existing methods have failed. If the method looks directly at the class attribute, for example, it will not see what it expects. If we don't trust the methods, the alternative is to coerce the S4 object to the corresponding S3 class. This is equivalent to considering the inheritance relationship to be non-simple (Chambers, 2008, page 346). The classes extending abnormal object types discussed in section 3 are examples for which such coercion is needed. As with any non-simple superclass, the penalties are some extra computation and more seriously that the programmer will have to restore any additional information from the subclass if the method is intended to return an object from the original class.

In the current mechanism, R maintains a table of classes for which explicit coercion will be performed before dispatching a corresponding S3 method. Version 2.9.1 of R sets the table up to include all the S3 classes registered in the methods package plus the abnormal object types. Future versions are planned to introduce a mechanism to turn conversion on or off, in the call to `setOldClass()`.

One can also combine S3 methods defined directly for an S4 class *and* S3 inheritance. For example, suppose S4 class `"myFrame"` extends `"data.frame"` and the designer of the class also wants to write an S3 method for `"myFrame"`. This is allowed, although not recommended. The class must registered by the `S3methods` argument, in which case selection should work as expected because

the ".S3Class" slot will now contain "myFrame" and its superclasses, including "data.frame".

As an example requiring an S3 method for an S4 class, suppose we have an S4 class, "L" say, that has its own method for sort(). The usual technique would be:

```
setGeneric("sort")
setMethod("sort", "L", .....)
```

This works fine for calls to sort() from the global environment or from any package importing the S4 method. But suppose someone calls the function median() on an object from this class. The default method for median() calls sort() and selects the appropriate order statistics. One might expect this to work for class "L", but it will not. Because median() is in the base namespace, the version of sort() that it calls ignores S4 methods. To ensure that median() and other base functions select the desired method, an S3 version must be defined. The simplest and clearest technique is usually to define the S3 method first and then call that from the S4 method. In the example,

```
sort.L <- function(x, decreasing = FALSE, ...)  ....
setMethod("sort", "L", function(x, decreasing = FALSE, ...)
    sort.L(x, decreasing, ...) )
```

5 Superclass Ordering

A formally defined class may have one or more *direct superclasses*, which can be specified in three ways:

1. via the contains argument to setClass(), when the class definition is created;

2. as a member of a class union;

3. via a direct call to setIs().

See (Chambers, 2008, sections 9.3-9.4). In addition, all the superclasses of the direct superclasses are by definition superclasses of this class as well.

The contains slot in the class definition is a list of objects defining the relation to each superclass. The ordering of the list is used when selecting an inherited method (page 11 below). A method defined for a class earlier in the list of superclasses is preferred.

It's convenient to speak of the *generation* of a superclass. The direct superclasses are the first generation, their direct superclasses the second generation, etc. R stores the generation as the distance slot in each element of the contains list.

Two principles to guide superclass ordering in R have been (and still are):

1. Superclasses that are closer in generation to the main class should appear closer in the list. That is, direct superclasses should appear before others; after these, their direct superclasses should appear before others; and so on.

2. Direct superclasses should appear in the order they appeared in the class definition. Similarly, superclasses of the first direct superclass should appear before those of later superclasses, and so on.

R uses these two principles to generate the initial superclass ordering. A list is computed of the relation to each superclass, by merging the `contains` slot from the definition of each of the direct superclasses. This list is sorted by generation, and since the sort is stable and the superclasses are introduced in the order of the direct superclasses, the second principle is followed as well. Provided the set of all superclasses of the direct superclasses has no duplicates, no further action is needed.

However, the same class can appear in more than one of the superclass lists of the direct superclasses. The list produced by the initial sorting will then have some duplicate names (the class extension objects in the list will not generally be identical). Through version 2.8 of R, uniqueness was not enforced. When the list was used for method selection, the effect was to pick the first instance of the superclass that appeared with the smallest generational distance, not always the desirable solution.

To resolve (when possible) ambiguities from superclasses appearing more than once, a third principle is invoked:

3. Superclasses should appear in a consistent order; that is, if class `"B"` precedes class `"C"` in the `contains` list, then it should do so also in the `contains` list of any superclass that extends both `"B"` and `"C"`.

Aside from some intuitive appeal, this principle is desirable to avoid possible surprises in method selection (see the Appendix, page 14). If there are no ambiguities, the first two principles as applied in R ensure the third as well.

Starting with version 2.9.0, R removes multiple occurrences of superclasses. If duplicates can be removed to satisfy the third principle, this will be done. If not, a warning message reports the inconsistencies remaining. The Appendix shows examples where consistency is not possible.

One of the few other functional programming languages with multiple inheritance and multiple dispatch is Dylan (Shalit, 1996). It has similar rules for consistent class inheritance, with the differences that it does not use a concept of generational distance, does not have the whole-object view of class properties, and takes a less tolerant view of ambiguous inheritance than R. See (Shalit, 1996, pages 54-55) for details.

6 Selecting Inherited Methods

The introduction of a consistent superclass ordering has been used to support a more specific criterion for selecting inherited methods, avoiding some potential inconsistencies in the previous selection algorithm. The previous algorithm computed the generational distance between the class of the actual argument and the corresponding class in the signature of each method for this function. If more than one argument appeared in the signature, the distances for each argument were added, and the total distance used as a penalty score for the candidate methods. In the case of equal scores, the choice was considered ambiguous, a warning issued and the lexically first of the tied candidates chosen. See (Chambers, 2008, section 10.6) for examples.

Although this heuristic criterion behaves adequately most of the time, it has several infelicities. As noted above, the generational distance is not always meaningful when a superclass can be reached by more than one path. In particular, the universal superclass `"ANY"` does not have a generally meaningful distance from actual classes. Also, the use of distance as a criterion does not take explicit account of the second principle above, that direct superclasses are ordered by the order of their appearance (although choosing the lexically first possibility effectively applies this principle if there are no duplicate superclasses).

With a fixed and, if possible, consistent ordering of superclasses in all cases, a more explicit set of criteria will be used:

1. For each argument in the signature, the closest defined class is the first occurring class in the superclass list of the actual argument's class. The best matching method(s) are those with that class in their signature for the argument in question.

2. The best matching method, if any, is a best matching method for all arguments in the signature. If no best method exists, selection is ambiguous.

For functions with only one argument in the signature of the methods, there will always be a best method. This includes the case that the generic function has more than one argument in its signature, but all the defined methods only use one of these arguments. In the one argument case, the method selected will be the same as with the previous algorithm, but no ambiguities will be reported.

When an ambiguous case occurs, some heuristics are applied to eliminate some of the candidates. If more than one candidate still remains, the lexically first candidate is used, which amounts to choosing the first of the best methods for the first argument in the signature. The following heuristics are applied in order, exiting when and if only one preferred method remains.

- Methods that qualify by conditional inheritance are eliminated (for example, methods defined for class `"ts"` when the actual class is `"mts"`). Hopefully, this is a rare occurrence.

- Methods are selected with the least total generational distance between the target and defined classes (this was the primary comparison before consistent superclass ordering was enforced).

- If some, but not all, candidate methods are from a group generic function, these are discarded in favor of methods for the specific function.

- Preference is given to exact matches on individual arguments; that is, to methods whose signature includes the class of one of the corresponding actual arguments.

The disambiguating function in the methods package signals a condition, which by default prints a message noting the signature selected, the alternative candidates and any of the above heuristics applied. This action has been downgraded from a warning in previous versions, since typically it's the package writer, and not the user who calls for the method selection, who is responsible for the ambiguity. (See the next section.)

A different action in response to this condition can be programmed by setting the option `"ambiguousMethodSelection"`. The option should be set to a calling handler, that is a function of one argument, which will contain the message. When the handler function is called in this case, the argument object also has attributes as discussed in the next section describing the ambiguity.

Detecting Ambiguous Method Selection

Ambiguities in method selection tend to be created by package designers but detected by package users, who may have done nothing wrong but are often unnerved by the reported ambiguity. For this reason, the condtion reported has been downgraded from a warning and phrased in an attempt to just report the result neutrally.

Even so, it is much more desirable to detect ambiguities during package design and either eliminate them or at least provide some reassurance to users. In principle, not all ambiguities arise from a single package; it is quite possible for two packages to create competing methods neither of which dominates the other. The majority of ambiguous selection, however, likely arises because a package creates a set of new classes and corresponding methods for operating on them in pairs. (Note that the current mechanism, unlike the previous one, always selects an unambiguous method for functions of one argument, so it's only multi-argument selection that matters, typically for operators or functions with two or more data objects.)

A new function, `testInheritedMethods()`, has been added to the methods package to test method selection for the currently known subclasses that are relevant to the methods for a particular generic function. Normally the call is simply:

```
testInheritedMethods(f)
```

where `f` is a generic function or its character string name. The value returned is an object from the class `"MethodSelectionReport"`. This has a slot reporting all the results of the method selections, plus slots reporting the results of any ambiguities found. A `show()` method for the class itemizes all these ambiguities.

It's important when calling `testInheritedMethods()` to have attached any packages that contain relevant methods or classes. This should include of course any packages with methods for `f()` but also any software that defines relevant subclasses extending those for which such methods are defined.

The rest of this section examines the computations involved, the results reported, and the question of what action might be taken by package designers. To look at examples in practice, we will use the Matrix package, which has a very rich set of classes and methods. I hasten to add that this is not to pick on or criticize the package. The package design introduces a wide range of classes and attempts to give thorough coverage including fall-back methods to detect invalid or unsupported combinations. The ambiguous method selection will often end in an error message, making the reported ambiguities irrelevant. In the process, it provides a rich testbed for detecting ambiguous methods.

The computations and reporting are necessarily nontrivial. There may be a large number of subclasses, particularly if basic classes such as `"vector"` are included in some method signatures. The computations begin by identifying, for each argument in the signature, all the classes that are subclasses of the class in one or more of the method signatures. The complete set of relevant signatures is the "outer product" of the set of classes for the various arguments. This may be a very large set, but in fact only one signature is retained from each subset of equivalent signatures. Signatures are equivalent if they have the same inheritance pattern.

For example, suppose `f()` is a function of two arguments with two methods defined:

```
setMethod(f,
  c("dMatrix", "nMatrix"),
    ....)

setMethod(f,
  c("sparseMatrix", "nsparseMatrix"),
    ....)
```

These are classes in the Matrix package, each with a number of subclasses. All the subclasses of `"dMatrix"` or of `"sparseMatrix"` are relevant for the first argument, and all the subclasses of `"nMatrix"` or of `"nsparseMatrix"` for the second. In principle all pairs might be tested, for a total of `22 * 73 == 1606` possibilities in the version of Matrix examined. In fact, each pair of a subclass of `"dMatrix"` and of `"nsparseMatrix"` triggers the same inheritance computation, so we only need one representation of each pattern, `4` in total. The computations for `testInheritedMethods()` begin by finding all the equivalences among combinations of subclasses and retaining one of each set for testing. In addition, one normally does not need to consider virtual classes, so long as these have at least one non-virtual subclass, so these are eliminated in advance.

With a large number of methods there will still be a substantial number of distinct computations. For example, the operator `"+"` had `529` combinations to test from direct and group methods. To provide a manageable but

detailed record of ambiguities, `testInheritedMethods()` returns an object of class `"MethodSelectionReport"`. The `show()` method for printing the class itemizes any ambiguities found. Slots in the class record detailed information, each of which is a vector of the corresponding attribute of the condition object passed to the calling handler when the individual ambiguity is detected: `"target"` for the target signature; `"candidates"` for all the matching best signatures; `"selected"` for the signature selected; and `"note"` recording any of the heuristics used to resolve ambiguities. Signatures are returned in the form of single strings, for ease of manipulation. For example, the signature `c("dMatrix", "sparseMatrix"` is represented by `"dMatrix#sparseMatrix"`.

Appendix: Failure of Superclass Consistency

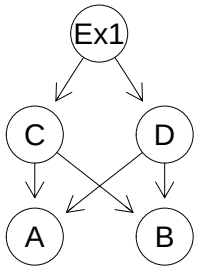
As may be intuitively obvious, classes can be defined that individually have superclass definitions consistent with the principles, but which when combined produce a new class that is no longer consistent.

A counter-example to preserving the ordering of direct superclasses is trivial. Here is some R code, followed by a plot of the superclass inheritance.

```
setClass("C", contains = c("A", "B"))

setClass("D", contains = c("B", "A"))

setClass("Ex1", contains = c("C", "D"))
```



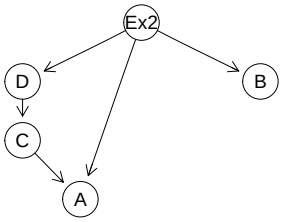
(Classes not defined in the example are assumed to have trivial definitions, with no superclasses.)

The conflict here is a basic one: Classes `"C"` and `"D"` take opposite views on the relative closeness of their two direct superclasses. No class can extend both `"C"` and `"D"` in a consistent way. I have not yet encountered examples in practice, but it's not hard to imagine them, particularly if `"C"` and `"D"` came from different packages.

Ambiguities in the generational distance to a superclass are much more likely, but fortunately not usually fatal for superclass consistency. An artificial example that is fatal is the following:

```
setClass("C", contains = "A")
setClass("D", contains = "C")
```

```
setClass("Ex2", contains = c("D", "A", "B"))
```

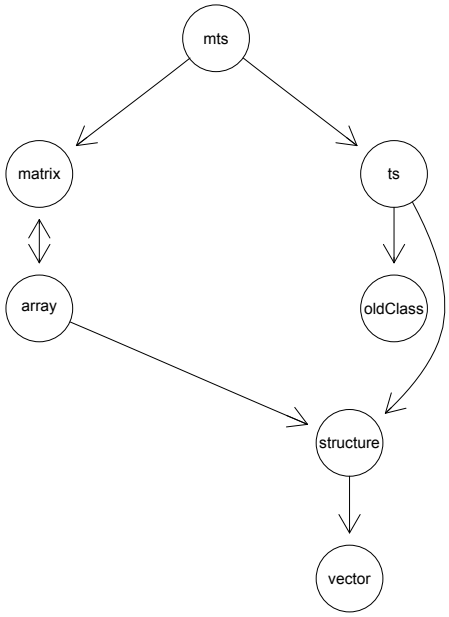


The generation distance between `"Ex2"` and `"A"` is either 1 or 3, depending on which route we take. In this simple example one might argue that the direct inheritance should prevail, but an analogous example can be constructed with two indirect relationships.

There is no ordering of class `"Ex2"` and its superclasses in which all the pairwise orderings are preserved. Class `"A"` must come at the end to preserve the ordering of `"D"` and its superclasses, but then the order of the direct superclasses of `"Ex2"` is not preserved.

Ambiguity in superclass distances that does not prevent finding a consistent ordering is quite common. The stats package in the core R software includes S3 classes for univariate and multivariate time series. If the user calls the `ts()` function with a matrix of data, the object returned has `"mts"` as its main class. The objects created have attributes corresponding to both matrices and time series. An S4 definition for `"mts"` then has the definition and class inheritance graph:

```
setClass("mts", contains = c("matrix", "ts"))
```



The class definition itself simply states the obvious: objects from class `"mts"` have all the properties of both matrices and time series. The inheritance from class `"structure"` is the interesting feature of the graph.

Both `"matrix"` and `"ts"` are "structure" classes in the classic sense of the S language: objects from the classes consist of a vector with additional properties that define its layout, independent of the type or values in the vector. For this reason, class `"structure"` appears twice in the inheritance of `"mts"`. Class `"ts"` has `"structure"` as a direct superclass. Because `"matrix"` is defined as a specialization of general multi-way arrays, however, it extends `"structure"` through `"array"`. The two paths from `"mts"` to `"structure"` have distance 3 and 4, but the ordering produced for the superclasses of `"mts"` is consistent with both direct superclasses.

```
> extends("mts")
[1] "mts"       "matrix"    "ts"
[4] "array"     "structure" "oldClass"
[7] "vector"
> extends("matrix")
[1] "matrix"    "array"     "structure"
[4] "vector"
> extends("ts")
[1] "ts"        "structure" "oldClass"
[4] "vector"
```

When the superclass order of a class is inconsistent with that for one of its superclasses, the new class may fail to inherit methods as expected, even if no

new methods are written. With a reversal of order in the two superclasses of class "Ex1" on page 14, the problem is easy to see. Suppose f() is a generic function of one argument, with methods for both classes "A" and "B". A programmer writing code for class "C" would expect to inherit the former method, a programmer writing for class "D" the latter. Whatever way the superclasses of class "Ex1" are ordered, some methods may not be inherited as expected.

Somewhat subtler but similar problems arise from the definition of class "Ex2" on page 14. Ignoring one or the other occurrence of "D" gives the alternative results for extends("EX2").

```
EX2 C D C1 A

EX2 C C1 A D
```

In the first ordering, methods defined for class "D" will be chosen in preference to those defined for either "C1" or "A"; in the second ordering the opposite is true.

In this example, neither ordering can be considered "correct". Both give potentially surprising results for the user. The second ordering means that methods defined for a direct superclass, "D", will not be chosen in preference to, say, a method defined for the indirect superclass "A".

The defect in the first ordering is more subtle, but consider a programmer writing methods for class "C". Suppose the programmer needs to override a method for class "D", say because it throws an error. If there is a suitable method for class "A", the programmer assumes that this method will be chosen. But it will not, if the object comes from class "EX2", because the bad method from "D" will now be chosen. The situation in this case is admittedly more subtle, but just for that reason it can potentially produce more confusing errors. And the situation is not as obscure as one might think. Multiple occurrences of a class may occur when the class in question is a form of "base" class for other classes—one to which method selection falls through if no specific class is intended. Just in this situation the defect described here can occur.

The example here is fairly minimal. If class "D" had superclasses itself, these would be affected similarly, and if more than one class had inconsistent inheritance, consequences could be further complicated.

References

J. M. Chambers. *Software for Data Analysis: Programming with R.* Springer, New York, 2008. ISBN 978-0-387-75935-7.

J. M. Chambers. *Programming with Data: A Guide to the S Language.* Springer, New York, 1998.

J. M. Chambers and T. J. Hastie. *Statistical Models in S.* Chapman & Hall, London, 1992. ISBN 9780412830402.

A. Shalit. *The Dylan Reference Manual*. Addison-Wesley Developers Press, Reading, Mass., 1996.