RBioinf Introduction

May 3, 2016

Introduction

The *RBioinf* package has a handful of functions that support the monograph, *R Programming for Bioinformatics*. It also has a few functions that were developed for a more in depth study of S4 OOP that were never included in that monograph, but for which the discussion may be of use to some (and which unfortunately may no longer be correct as S4 has evolved a lot in recent times).

S4 Classes

Details on S4 classes are given in a number of other locations. This is not a tutorial on their use, but rather moves quickly to a discussion of the code for computing linearizations.

0.1 Class Linearization

For many different computations, especially method dispatch, an algorithm for specifying a linear order of the class inheritance tree is needed. All object oriented programming languages support the computation of a linearization. Here we provide a succinct description of the procedure and give some simple examples.

The methods used to compute a linearization are different for different languages. For languages that only support single inheritance the precedence rule that is used is that a class is more specific than any of its superclasses. Methods defined for subclasses will override methods defined for superclasses. When multiple inheritance is supported then it may not be clear which one of two superclasses is more specific. The issue arises when there are two superclasses, and these classes do not themselves have a subclass – superclass relationship. In S4 the style of linearization named C3 (**?**) has been adopted and will be used. An implementation of this linearization is provided by the `computeClassLinearization` function with the `C3` argument set to `TRUE`.

In languages with multiple inheritance some mechanism must be provided to resolve the appropriate order in which to inherit properties. There can be conflicts or multiple

methods which might be applicable and it is important these issues be resolved. Ideally they should be resolved at class definition time, as there will be obvious efficiency losses if this is done at dispatch time.

C++ and Eiffel provide programming constructs that allow/require the programmer to specify the class precedence hierarchy. In languages such as Common Lisp (and the CLOS object system) and Dylan the approach that is taken is usually to have an automatic method of computing a linearization. A linearization of a class hierarchy is a linear ordering of the class labels contained in the hierarchy, such that each class appears once. There has been a reasonable amount of research in these automatic methods and in this section we discuss some of those and propose the one that was recommended in **?**.

In most object-oriented languages (especially those with single inheritance) the precedence rule that is used is that a class is more specific than any of its superclasses. And methods defined for sub-classes will override methods defined for superclasses. When multiple inheritance is supported then it may not be clear which one of two superclasses is more specific. The issue arises when there are two superclasses, and these classes do not themselves have a subclass – superclass relationship.

Both CLOS and Dylan use the local precedence order (LPO) the order of the direct superclasses given in the class definition in computing the linearization, with earlier superclasses considered more specific than later ones. If there are no duplicate class labels in the hierarchy then this is then simply a bread-first search of the superclass definitions. But when one or more classes are inherited from different superclasses this definition becomes more complicated, and can in fact not be satisfied, as we will see below.

We begin with an example from **?** and note that our examples will mainly be from this reference. We also note that we will make use of the *graph*, *RBGL* and *Rgraphviz* packages from the Bioconductor project to provide us with the necessary infrastructure for representing, computing on and drawing the class hierarchies.

```
> library(RBioinf)
> setClass("object")
> setClass("grid-layout", contains="object")
> setClass("horizontal-grid", contains="grid-layout")
> setClass("vertical-grid", contains="grid-layout")
> setClass("hv-grid", contains=c("horizontal-grid", "vertical-grid"))
> LPO("hv-grid")

[1] "hv-grid"         "horizontal-grid" "vertical-grid"   "grid-layout"
[5] "object"

>

> computeClassLinearization("object")
```

```
[1] "object"

> computeClassLinearization("grid-layout")

[1] "grid-layout" "object"

> computeClassLinearization("vertical-grid")

[1] "vertical-grid" "grid-layout"   "object"
```

An inheritance graph can be inconsistent under a given linearization mechanism. This means that the linearization is over-constrained and thus does not exist for the given inheritance structure. In the next code segment we create a class, *confused*, for which the LPO will attempt to place *horizontal-grid* before *vertical-grid* because this is their order in *hv-grid*, and it will also attempt to place *vertical-grid* ahead of *horizontal-grid* since this is their order in *vh-grid*; but clearly both of these properties cannot be satisfied simultaneously. We use the function `LPO` to compute the local precedence order.

In the code below, we see that there is no linearization of the classes possible for *confused*, which is why it is wrapped in a call to `tryCatch`.

```
> setClass("vh-grid", contains=c("vertical-grid", "horizontal-grid"))
> setClass("confused", contains=c("hv-grid", "vh-grid"))
> LPO("vh-grid")

[1] "vh-grid"         "vertical-grid"   "horizontal-grid" "grid-layout"
[5] "object"

> tryCatch(LPO("confused"), error=function(x) "this one failed")

[1] "this one failed"

>
```

A plot of the class hierarchy graph for the *confused* class is given here.

In order to fully describe and understand the mechanism used to determine which method should be applied, for a given argument list, and more fully to specify an ordering of methods, we must have a linearization of the class hierarchy. In many regards determining this hierarchy quickly and according to some widely recognized design principles will be essential for efficiency and for writing programs that work as their authors intended. The order in which one class precedes another is often called the class precedence list. In languages with single inheritance it is quite easy to compute but when multiple inheritance is used some problems can arise. Some of the history, and a detailed discussion of the issues is given in **?**. The strategy used in Dylan as well as an implementation are described in **?**, pg. 55.

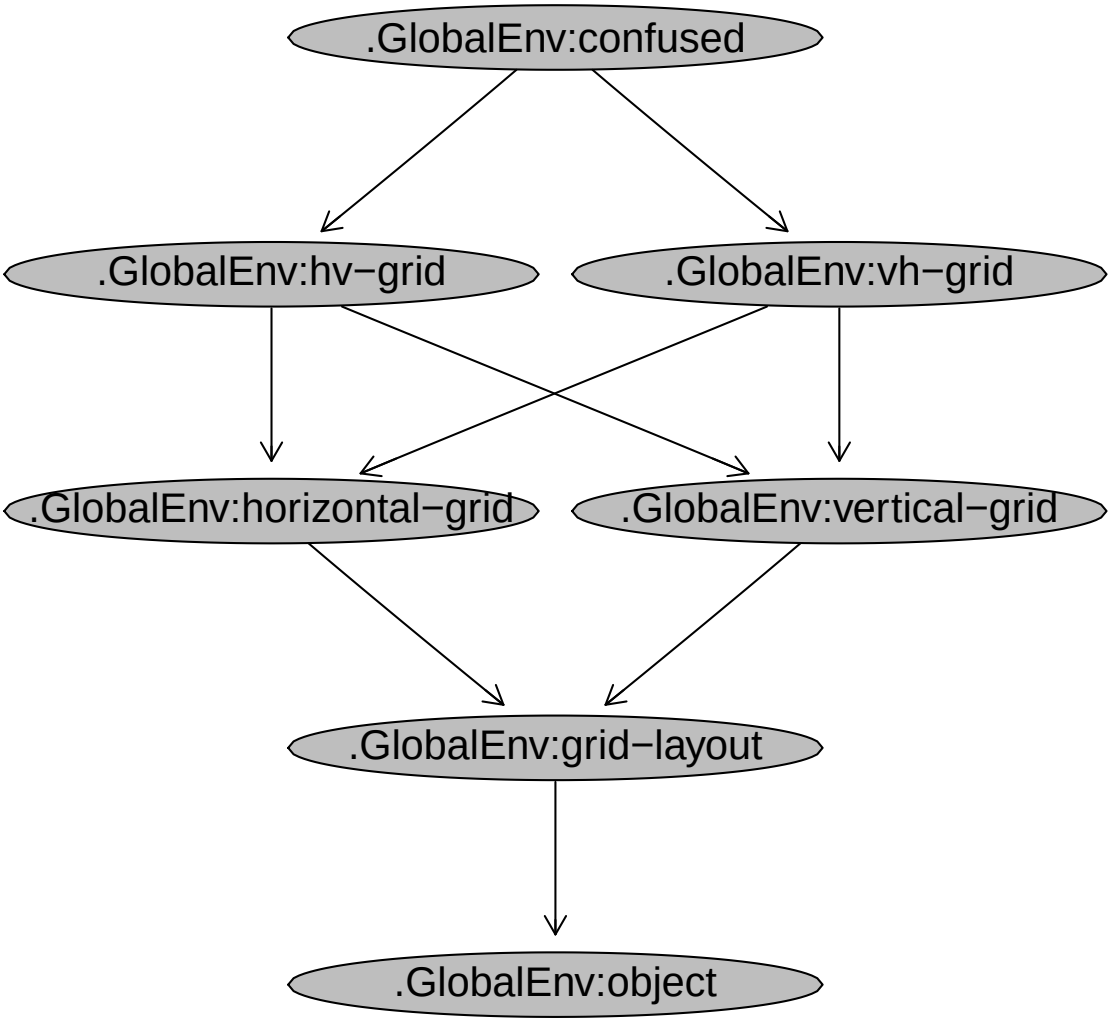


Figure 1: A class hierarchy that cannot be resolved.

We begin by considering only the class hierarchy that comes from using the `contains` argument when defining classes. Later, we will return to some of the more complex issues that arise if the `setIs` function is used to further affect the class hierarchy.

A class definition specifies a local ordering on that class and its direct superclasses using the following two rules:

- the class precedes all direct superclasses
- the superclasses have precedence in the order in which they are provided in the `contains` argument in the call to `setClass`, the ordering is right to left (highest to lowest).

The class precedence list for a class is total ordering on that class and all of its superclasses that is consistent with the local ordering and with the class precedence lists of all of its superclasses. Shalit indicates that two lists are consistent if for every A and B that are in both lists then either A precedes B in both or B precedes A in both. There may be several orderings that are consistent – some one of these must be chosen. (this an implementation detail). There may be no possible orderings and in that case an error is given on the call to `setClass`.

The first, and most basic rule, is that the ordering on direct superclasses of a class is defined by the order in which they are given in the `contains` argument to `setClass`. One way of determining the class precedence list is via the function `extends` which if called with one argument returns the class precedence list.

We first introduce a set of classes that will be used for our initial explorations. In the code chunk below, we see that the only real difference between the class c and d is the order in which they include a and b and that this difference is reflected in the output from `extends`.

```
> setClass("a")
> setClass("b")
> setClass("c", contains = c("a", "b"))
> setClass("d", contains = c("b", "a"))
> extends("c")

[1] "c" "a" "b"

> extends("d")

[1] "d" "b" "a"

> setClass("e", contains=c("c", "d"))
>
```

There are some functions that allow the user to understand the superclass relationships. Function from the *methods* package include `getAllSuperClasses` and `superClassDepth`. In addition we have added another helper function in the *RBioinf* package called `superClasses` which finds and reports only the direct, or most immediate superclasses. This last function will be used to build a graph of the class hierarchy for visual inspection.

```
> getAllSuperClasses(getClass("e"))

[1] "c" "d" "a" "b"

> cD = superClassDepth(getClass("e"))
> cD$label

[1] "c" "d" "a" "b"

> cD$depth

[1] 1 1 2 2

> superClasses(getClass("e"))

[[1]]
[1] "c"
attr(,"package")
          c
".GlobalEnv"

[[2]]
[1] "d"
attr(,"package")
          d
".GlobalEnv"

>
```

We now turn our attention to a more complicated situation, and one in which the ordering of the classes in the class precedence list might not be so obvious.

```
> cH = class2Graph("e")
>
```

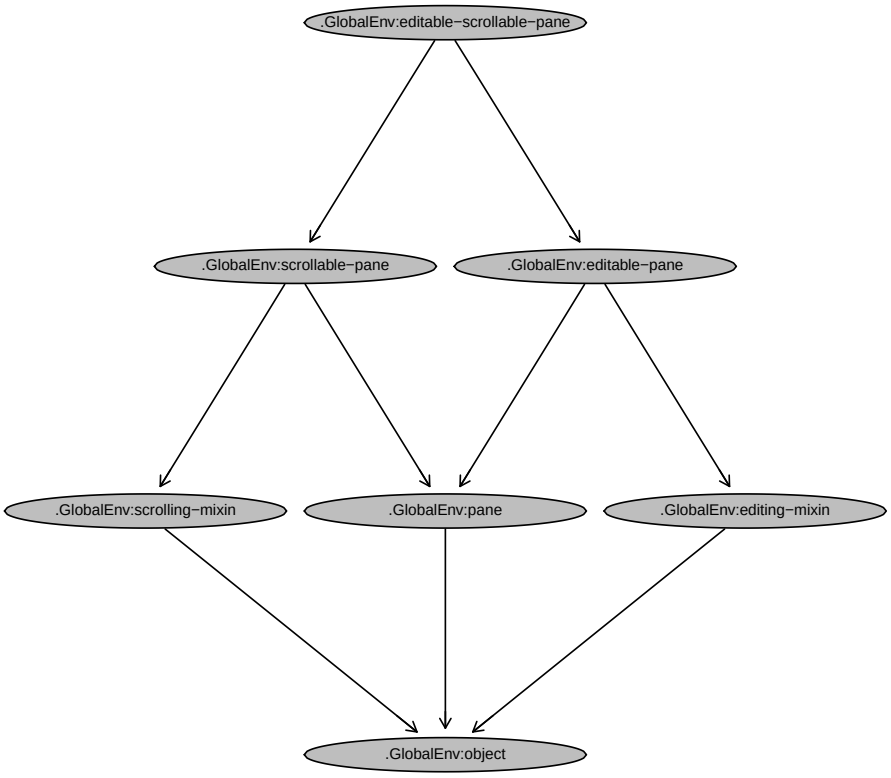
```
>  setClass("pane", contains="object")
>  setClass("editing-mixin", contains="object")
>  setClass("scrolling-mixin", contains="object")
>  setClass("scrollable-pane", contains=c("pane", "scrolling-mixin"))
>  setClass("editable-pane", contains=c("pane", "editing-mixin"))
>  setClass("editable-scrollable-pane",
+           contains=c("scrollable-pane", "editable-pane"))
>
```

For the *editable-scrollable-pane* class the two linearizations, Dylan-style and C3 provide different answers. It can be argued that the C3 linearization is less surprising since it puts *scrolling-mixin* ahead of *editing-mixin* and that is somewhat sensible since this would be the same ordering as their direct superclasses.

```
> LPO("editable-scrollable-pane")

[1] "editable-scrollable-pane" "scrollable-pane"
[3] "editable-pane"            "pane"
[5] "editing-mixin"            "scrolling-mixin"
[7] "object"

> LPO("editable-scrollable-pane", C3=TRUE)

[1] "editable-scrollable-pane" "scrollable-pane"
[3] "editable-pane"            "pane"
[5] "scrolling-mixin"          "editing-mixin"
[7] "object"

>
```

In the figure below we see the class graph for the *editable-scrollable-pane* class.

```
>  eWG = class2Graph("editable-scrollable-pane")
>  eWGattrs = makeNodeAttrs(eWG, shape="ellipse", fill="grey", width=4)
>  plot(eWG, nodeAttrs=eWGattrs)
>
```



Session Information

The version number of R and packages loaded for generating the vignette were:

```
R version 3.3.0 RC (2016-04-26 r70550)
Platform: x86_64-apple-darwin13.4.0 (64-bit)
Running under: OS X 10.9.5 (Mavericks)

locale:
[1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] grid      parallel  stats     graphics  grDevices utils     datasets
[8] methods   base

other attached packages:
```

```
[1] Rgraphviz_2.16.0    RBioinf_1.32.0      graph_1.50.0
[4] BiocGenerics_0.18.0

loaded via a namespace (and not attached):
[1] tools_3.3.0  stats4_3.3.0
```