

# Prototype-Based Programming in Statistical Computation

Louis Kates      Thomas Petzoldt

July 2, 2004

To the extent that object-orientation has influenced languages for statistical computation, the approach taken to date has been predominantly based on the notion of classes as the key organizing principle; however, there exists another possibility – the object-based or prototype approach to object-oriented programming. The use of prototype-based programming could provide a fundamental alternative for statistical computation. Prototype programming is particularly suitable for situations where there are a small number of objects and exceptions are frequent. This is often the case in statistical applications suggesting that this approach may have widespread applicability to statistical analyses.

As an example, the statistical language R can be employed to compactly describe and construct a variety of prototype-based approaches at a high level. We also explore a number of alternative approaches within the prototype-based paradigm.

**Key Words:** OOP, Object-based, R-language, Prototype Programming

## 1 INTRODUCTION

The class-based approach to object-oriented programming is the dominant approach among major widely used object-oriented languages such as C++ and Java; however, there exists another possibility – the object-based or prototype approach to object-orientation (Ducasse 2003; Lieberman 1986; Noble et al. 1999; Taivalsaari 1996a). The use of prototype-based programming could provide a fundamental alternative for statistical computation. Prototype programming is particularly suitable for situations where there are a small number of objects and exceptions are frequent. This is often the case in statistical applications suggesting that this approach may have widespread applicability to statistical analyses.

The R language (Ihaka and Gentleman 1996; R Development Core Team 2004) is a high level interpreted language used most often for statistical analysis. It is influenced by Scheme (Sussman and Jr. 1975; Steele Jr. and Sussman 1978; Kelsey et al. 1988) through its scoping, by functional programming through its

first-class functions and iteration-free ability to process whole objects at a time and by the Dylan language (Shalit 1996) through its approach to polymorphism. R can be regarded as an independent implementation of the S language (Becker et al. 1988) having similar syntax, but importantly for our examples, different scope rules. Its facilities include two class-based object-oriented systems known as S3 and S4 in which every object is associated with a class.

The purpose of the following is to describe the prototype-based approach in the context of statistics using the R language in order to make it accessible to statisticians and the R community without the need to understand languages such as Self (Agesen et al. 1992), Kevo (Taivalaari 1993, 1996b) or Lisp. A second purpose is to explore a number of alternative approaches within the prototype-based paradigm. The facilities of R make such descriptions possible in a relatively compact form at a high level.

In the context of R, a number of other object oriented approaches have been proposed including the R.oo add-on package (Bengtsson 2003) and the OOP add-on package (Chambers and Lang 2001). A demonstration of scoping that is provided with the R software, accessed via the R command `demo(scoping)` and discussed in the *Introduction to R* manual (Venables et al. 2004), illustrates not only scoping but an approach to object-oriented programming. All these approaches share the common thread of being class-based while the approach discussed here is purely object-based or prototype-based. In fact, the only use of prototype-based programming in statistical software of which the authors are aware is in Lisp-Stat (Tierney 1995, 1990); however, that exists in the context of Lisp and appears not to have been pursued by other statistical software packages in the intervening years.

The following will review the fundamental building blocks, lists and environments, available in R for constructing object-oriented systems and then proceed to discuss their properties as well as various prototype-based approaches that utilize these building blocks. The discussion will review object representation, object identity, delegation and embedding and also show how class-based object orientation can be expressed in terms of prototype-based object orientation. Short pieces of code will be provided to illustrate these concepts as concretely as possible.

## 2 OBJECT-BASED SYSTEMS

Classes are an organizing or classifying concept which allow program construction to occur by modelling abstract concepts as classes. The classes generate instance objects which represent concrete occurrences of the abstract ideas they embody. Such a process of identifying the general case and then proceeding from general to specific is the essence of the class-based approach in object-oriented programming.

An alternate paradigm that still falls within the domain of object-orientation is the pure object, object-based or prototype approach. In this approach, classes are no longer a language primitive. With no special language constructs for

classes, language rules can be simpler and more regular and specialized behavior can be more easily assigned to particular objects, freeing those objects from the constraint of conforming to specific classes. In this paradigm, objects are not instantiated from classes but rather created *ex nihilo* (from scratch) or created from an existing object by cloning or extending it (Dony et al. 1992, 1999).

A system with pure objects is typically constructed from specific examples or prototypes rather than through predetermined abstractions. The process is more concrete and thought to more closely mimic the ways in which the human mind represents knowledge. Object-based approaches can model class-based approaches but the other direction is somewhat problematic so, in practice, they have added power and flexibility despite their greater simplicity. (Lieberman 1986; Taivalsaari 1996a)

### 3 IMPLEMENTATION & REPRESENTATION

In the following we shall directly show two implementations of objects suitable for prototype-based systems in R. In order to make the presentation accessible to those with a basic understanding of R, we shall initially rely on built-in facilities of R even though a more sophisticated infrastructure could be built that hides and streamlines certain details.

#### 3.1 Lists

Originally “named components of lists were used as an informal approach to classes” in R (Chambers 1998). R lists can hold arbitrary named components including both variables and functions. An object represented as a list with one data component and two methods might look like this:

```
# Example. Object implemented as list.
oo <- list(.x = c(10, 20, 15, 19, 17),
          location = function(this) mean(this$.x),
          scale = function(this) sd(this$.x)
)
# display value of location parameter for object oo
oo$location(oo)
```

Note that the methods, `location` and `scale` refer to variables within the same object using the notation `this$.x`. The data component, `.x` in the example, is not encapsulated but may be accessed from outside the object using the notation: `oo$.x`; however, if we impose the coding convention that variable names beginning with a dot are only to be accessed from methods of the object then we get a weak form of encapsulation, albeit only through coding discipline.

We can introduce a new object `oo2` by *cloning* the old object `oo` like this:

```
oo2 <- oo
```

We can make modifications to the new object after which the new and old objects differ. In the following example, we replace the `location` method with a new method that uses the median rather than the mean:

```
oo2$location <- function(this) median(this$.x)
```

Note that the new object `oo2` was not instantiated from a class but rather was cloned from an existing object. The entire process involved no classes and exhibited a marked simplicity over class-based methods.

## 3.2 Environments

Another structure in R that can form the basis of objects is environments. Environments, like lists, are collections of component variables and functions. These components are accessed using the same notation, `oo$.x`, used to access list components. Here `oo` is the environment representing the object and `.x` is a variable in that environment which belongs to that object. Similarly `oo$location` refers to the function or method `location` belonging to the object `oo`.

We review some concepts associated with environments and introduce some notation:

- The local variables and functions that come into being during the execution of a function are said to form the local environment or *current environment*, denoted `environment()` in R. For this reason, environments are sometimes referred to as *execution contexts*.
- The environment within which a function is defined is referred to as its *lexical environment* or *enclosing environment*. The lexical or enclosing environment of function `f` is referred to in R as `environment(f)`.
- The environment within which a function is invoked is known as its *parent frame* or *caller environment*. From within an executing function, the caller environment can be referred to using the notation `sys.frame(-1)` in R. Note that this point refers to where the function was called or invoked whereas the previous point refers to where the function was defined.
- Each environment has a parent environment. In R, if variable `e` holds an environment then `parent.env(e)` is the parent environment of `e`. When a variable or function name is referenced and that variable or function is not found in the current environment a search takes place through the ancestors starting with the parent and ascending through the parent of the parent and so on until it is found.

- The parent of the current environment of an executing function `f` is its lexical environment. This accounts for the description of R as lexically scoped (Gentleman and Ihaka 2000). Restated in terms of the R notations discussed above, the previous discussion implies that the expression `parent.env(environment())` run directly from within `f` equals `environment(f)`.

Note that the parent environment is not necessarily the same environment as the parent frame.

Environments are a common concept across many languages. In R they are formalized and can be stored in variables and manipulated. Such manipulations include displaying the environment's identity, examining the components of an environment, discovering the parent of an environment, creating new environments, setting the parent of an environment and adding, changing and deleting components of an environment. Like lists, the components of environments are unencapsulated, although the same dot name convention discussed previously can be used and is observed by some utility functions.

An object represented by an environment might be written like this:

```
# Example. 1st try as object implemented as environment.
# (Flaws addressed later.)
oo <- local({.x = c(10, 20, 15, 19, 17)
            location = function() mean(.x)
            scale = function() sd(.x)
            environment()
          })
# display value of location parameter for object oo
oo$location()
```

The example is syntactically very close to the previous list implementation. The main difference is the `local` function which causes a new environment to be created and the `environment()` function within it that causes the environment to be returned so it can be assigned to variable `oo`. Also, this example relies on lexical scoping so that `.x` is referenced directly without using `this` as a mechanism to pass the object to each method.

A second object might be created called `oo2` pointing to `oo` as its parent like this:

```
# Example. 1st try at child implemented as environment.
# (Flaws addressed later.)
oo2 <- local({
            location <- function() median(.x)
            environment()
          }, envir = new.env(parent = oo))
with(oo2, location())
with(oo2, scale())
```

Note the line containing `parent = oo` in the example showing the parent being specified. Also, the notation `with(oo2, scale)()` is used to refer to methods in objects that have parents. In this case, `scale` is not found in `oo2` so a search is made through the parent environment of `oo2` effectively causing `oo$scale()` to be executed. The R `oo2$scale()` notation introduced earlier does not search into the parent which is why `with` was used here.

Note that `oo2` can be regarded as a child object to parent `oo` even though no classes have been defined to instantiate `oo` or `oo2`. The parent/child relationship is at the object level and at not the class level.

At this point we have introduced the two major implementation building blocks, lists and environments, available in R and have shown one representation of a prototype-based approach for each. In the following we discuss additional variations of the above, address some flaws and delve into a number of relevant properties.

## 4 OBJECT IDENTITY

The identity of an object is an identifier of the object that is independent of the contents of the object. An identifier will sometimes be referred to simply as *id*. Two objects can have contents which are equal but may nevertheless be distinct objects in the sense that the contents of one can be changed without modifying the other. On the other hand, if two objects have the same identity then they *are* the same object and changing either one changes the other. Object identity is typically implemented using pointers (i.e. addresses) often described as references.

R lists do not have an obvious object id. Assigning a list or passing it to a function causes a copy of the contents of the list to be assigned or passed rather than a copy of the list's identity. This can be problematic for implementing objects but can readily be addressed using R environments to implement objects rather than lists; however, in doing so we would miss important issues as well as a chance to provide alternate implementations for certain prototype-based concepts and thereby reduce the effectiveness of our understanding. Thus we proceed with a limited exploration of this problem using lists as the foundation structure before returning to environments. Some readers may wish to skip the remainder of this section.

In R, assigning a list to another list copies its contents, not its identity. After assignment, the two lists have separate identities so the assignment of list `L1` to list `L2` followed by a modification to `L2` does not change the original list `L1`:

```
# Example. Copy semantics of lists.
L1 <- list(m = 1)
L2 <- L1
L2$m <- 3
# at this point L2$m is 3 and L1$m is 1
```

The nature of the sharing relationship between the two lists is one of *creation-time* sharing only. That is the two lists share the same values at the time the second is created and from that point on they diverge.

Note that copying occurs with lists, not only in the above situation with explicit assignment, but also implicitly when lists are passed to functions. A copy of the list, rather than the list itself, is passed through arguments to a function so modifications to the list within the function are actually made on the passed copy and not on the original list. Returning to a variation of our original example, the call to `bias` in the code below has no effect on `oo`.

```
# Example.
# Erroneous processing due to pass-by-value semantics.
oo <- list(.x = c(10, 20, 15, 19, 17),
          # following line is wrong!!!
          bias = function(this, b) this$.x <- this$.x + b
        )
# add bias to .x
oo$bias(oo, 1)
oo$.x # .x not changed!
```

A copy of `oo`, not `oo`, is passed to `bias` and the bias is added to the `.x` in that copy leaving `oo$.x` itself unchanged. We now discuss several approaches to this problem.

#### 4.1 Pass Back Modification

The usual way of addressing this problem is to pass the object to the method and then *pass back* the modified object. The following example passes `oo` to `bias` which modifies it and then returns the modified copy of `oo` which is assigned back to `oo`.

```
# Example. Pass-back modification.
oo <- list(.x = c(10, 20, 15, 19, 17),
          bias = function(this) {
            this$.x <- this$.x + 1; this
          }
        )
# add bias to .x
oo <- oo$bias(oo)
oo$.x # bias has been added
```

The reason to pass back the modified object was not really due to the fact that we are passing lists themselves rather than an object id but also because of the copy-by-value semantics of R function calls. For example, consider the MacAnova language (Oehlert and Bingham 1998), which is similar to R in syntax but has no functions—only macros. In MacAnova, lists are called structures and, like R, have no obvious object id. However, in MacAnova macros may directly

modify their arguments so there is no need for passing back the object. The following MacAnova code illustrates this:

```
# Example. MacAnova structures as objects.

# define macro to convert string to macro and execute
exec <- macro("@tmp <- macro($1); @tmp($2)")

# object. $1[1] means first argument's first component
oo <- structure(x: vector(10, 20, 15, 19, 17), \
               bias:"$1[1]<-$1[1]+1;;" )

# add bias to x
exec(oo$bias, oo) # oo not passed back
oo$x
```

## 4.2 Object Identifiers

Another way to address this problem is to pass an object identifier — a unique identifying characteristic of the list that is unchanged when the list representing the object changes. At the same time it must be traceable back to the original list. This identifier can be passed to and used by the method to discover which object to use. Since the object identifier is unchanged even if the contents of the object are changed it matters not whether the identifier or a copy of it is used at any particular point.

Two possible candidates for object identifiers for lists are:

1. **get/put:** An object identifier can be formed from two routines: one to get the contents of the object and another to set the contents. We can even store the object identifier in the object itself. In the example, below, each time we create a new object or clone of an existing object we create

the object identifier consisting of `get` and `put` functions and place them in the object itself:

```
# Example. get/put as object identifier for list.
oo <- list(.x = c(10, 20, 15, 19, 17),
  bias = function(this) {
    this <- this$get()
    this$.x <- this$.x + 1
    this$put(this)
  },
  location = function(this) {
    this <- this$get(); mean(this$.x)
  }
)
oo$get <- function() oo; oo$put <- function(x) oo <<- x
oo$location(oo)
oo$bias(oo)
oo$location(oo)
```

R's lexical scoping is crucial to make the above work properly. If `oo` is passed to a function then that function receives a copy of `oo` and a copy of the object identifier, namely the `get` and `put` routines in it. A copied function has the same environment as the original function and due to lexical scoping the `oo` inside the `get` and `put` routines necessarily refer to the original `oo` where the `get` and `put` routines were defined and not any copy of `oo` that might lie within the function where they are invoked.

Although the above example is awkward, the syntactic annoyances could be camouflaged using various devices in R. Since our aim is just to illustrate the concepts we shall not pursue that here.

2. **name and environment:** A different object identifier can be formed from the name and environment of the list as shown in the following where

we store these two quantities in the `name` and `env` list components of the object identifier:

```
# Example. Name and environment as list object identifier.
oo <- list(.x = c(10, 20, 15, 19, 17),
  bias = function(this, b) {
    this <- get(this$name, this$env)
    this$.x <- this$.x + 1
    assign(this$name, this, this$env)
  },
  location = function(this)
    mean(get(this$name, this$env)$.x),
  name = "oo",
  env = environment()
)
oo$location(oo)
oo$bias(oo)
oo$location(oo)
```

The example is quite similar to the prior examples except that we use the R `get` and `assign` commands to convert the object identifier to and from the object in `bias`.

Although it is possible to laboriously create object identifiers for use with lists, R already has a construct that provides object identifiers directly without such workarounds so we shall not pursue this line of reasoning further. Rather we shall use Pass-Back-Modification when using lists as this approach is more list-like and reserve our use of object identifiers for environments where they are more natural.

### 4.3 Environments and Object Identity

Unlike R lists, environments in R are references and so are independent of their contents. The assignment of environment `e1` to `e2` simply causes `e2` to point to the same environment as `e1`:

```
# Example. Environments.
e1 <- new.env() # create a new empty environment
e2 <- e1        # copies object identifier
e1$x <- 123
e2$x # 123
```

Unlike the situation with lists where there was only sharing at creation time, with environments there is *life time sharing*. The two environments are the same and share the same variables and functions for life. In other words: the symbols `e1` and `e2` are two references to the same object.

Similarly, passing an environment to a function copies the identifier but not the contents of the environment. The various problems cited in the previous

discussion of lists are not present with environments. Cloning is no longer as simple as just making an assignment but a clone utility can be developed in only a few lines of code which eliminates any difficulty. This utility copies all the components of the environment in its first argument to a newly created environment which it returns. As functions are copied, the copied functions have their environments reset to the new environment.

```
# Example. Clone an object.
# uses clone.environment from appendix

oo <- local({.x = c(10,20,15,19,17)
            location = function(this) mean(this$.x)
            scale = function(this) sd(this$.x)
            environment()
})
oo2 <- clone.environment(oo)
```

## 5 DELEGATION & EMBEDDING

The counterparts to inheritance in class-based systems are delegation and embedding in prototype-based systems. Inheritance typically refers to a parent/child relationship between classes whereas delegation and embedding refer to certain such relationships between objects.

We shall begin our discussion with a problem. Let us modify our previous example. We shall use an object `oo` with components consisting of a data vector and `location` and `rms` methods. The `rms` method calculates the root mean square of the data relative to the location measure. We also create a new child object `oo2` with parent `oo` which also has a method `location`. The intention here is that the child object `oo2` has the same behavior as the parent object `oo` except that we have overridden `oo$location` with `oo2$location`.

A naive implementation is to create an environment for each of `oo` and `oo2` setting the parent environment of `oo2` to `oo`.

```
# Example. Wrong way to do delegation.
oo <- local({.x <- c(10, 20, 15, 19, 17)
            location <- function(x = .x) mean(x)
            rms <- function()
                sqrt(mean((.x - location())^2))
            environment()
})
oo2 <- local({location <- function(x = .x) median(x)
             environment()
}, new.env(parent = oo))
with(oo, location)()
with(oo, rms)()
with(oo2, location)()
with(oo2, rms)()
```

In this example, each object contains methods and variables and the parent object is set as the parent environment of the child object so that searches through the child will continue into the parent if they fail in the child.

The example ends in four test runs. The first two are calls to `oo` methods (i.e. parent methods) and both work as expected. The third is a call to the `location` method in the child `oo2`. Since search begins in the child it is found before the parent `location` is searched and so has the effect of overriding the `location` method in `oo`. Unfortunately, the last test gives the same answer as the second. The call to `rms` in `oo2` acts as if `oo2$location` has not overridden `oo$location` (even though it was overridden correctly in the third test).

The following execution sequence ensues (see left side of figure 1):

1. In the last line `rms` is called with object `oo2`.
2. Since there is no `oo2$rms`, the system searches the parent of `oo2` and finds `oo$rms`.
3. `oo$rms` then calls `location`

However, when `oo$rms` calls `location` it finds `oo$location` since the search path for the `oo` environment starts in `oo`. We had intended that `oo2$location` override `oo$location` so that the `rms` would use the new `location` in `oo2` rather than the `location` in the parent, `oo`. That is, what we really had wanted was that when `oo$rms` calls `location` it first looks for `oo2$location` and only if that fails does it look for `oo$location`.

First we introduce some definitions to facilitate statement of the problem. Let us denote the object that was referenced externally as the *receiver*. In the example above, the call that started everything was `with(oo2, rms)()` making `oo2` the receiver. We shall refer to a method call that potentially calls another

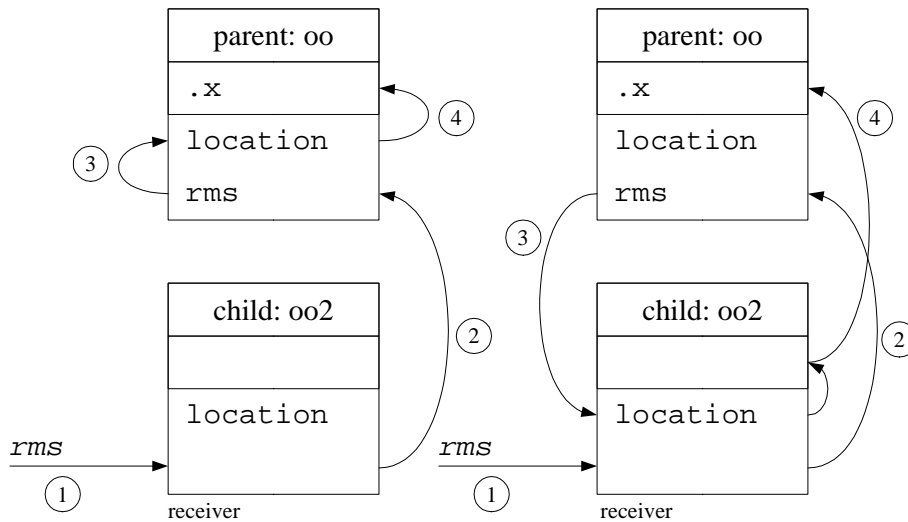


Figure 1: Illustration of wrong (left) and correct (right) method lookup when using delegation.

On the left, search is done via lexical scoping so: (1) message `rms` is sent to object `oo2`, (2) `rms` is not found in `oo2` so search continues in `oo` where it is found, (3) `rms` calls `location` using lexical scoping and so finds the one in `oo`. (4) Finally, `location` uses `.x` which it also accesses via lexical scoping and finds in `oo`.

On the right, search is always done starting at the receiver so: (1) message `rms` is sent to object `oo2`, (2) `rms` is not found in `oo2` so search continues in `oo` where it is found, (3) `rms` calls `location` which is found in the `oo2`, the receiver. (4) Finally, `location` looks in the receiver for `.x` and not finding it looks in `oo`.

method call in the same object a *self-call*. For example, the call to `location` from within `oo$rms` is a self-call since `location` exists as a method in the same object as `oo$rms`.

With these definitions, we can restate the prior paragraph as saying that method and variable searches should start with the receiver. The problem with the naive approach is that self-calls may not start with the receiver. Other calls do start with the receiver but self-calls may not. In fact, we shall refer to this as the self-call problem.

The solution to the self-call problem will lead us through many of the important considerations with prototype-based systems.

## 5.1 Embedding

In the example above, the child object, `oo2`, made use of the parent object, `oo`. It was almost as if the child were split across two objects with a portion of it existing within the parent. Another approach addresses the self-call problem by copying all the parent and then child variables and methods into the child object when the child is created. Essentially the search is done at object creation

time so that there is no longer any search required through the ancestors of the environment at method invocation time. In fact, we can just use the R `oo2$rms()` notation—`oo2$rms` in R looks only in `oo2` and does not search up the path of ancestors as does `with`. On the other hand, this approach has the limitation that if we later change any of the variables or methods in the parent, after the child is created, then the child variables and methods will not reflect that change.

The technique just discussed is referred to as embedding (as opposed to delegation) since a copy of the parent is embedded in the child. It features *creation-time sharing* between the child and parent since the values of variables and methods in the parent are shared at the time the child is created but from then on the values of the variables and methods may diverge.

Here is the above example, redone with embedding. It makes use of the `clone.environment` function defined in the appendix. The example code defines a parent object `oo`, clones it using `clone.environment` and copies the components of the child object to that clone. The result is returned as object `oo2`. When copying the components of `oo2` to the clone, `location` in `oo` is overwritten by `location` in `oo2`. Thus `oo$location` is correctly overridden in `oo2` regardless of whether `location` is called externally or via a self-call.

```
# Example: Embedding.
# uses clone.environment from appendix
oo <- local({this <- environment()
  .x <- c(10, 20, 15, 19, 17)
  location <- function(x = .x) mean(x)
  rms <- function()
    sqrt(mean((.x - location())^2))
  bias <- function() this$.x <- .x + 1
  this
})
oo2 <- local({this <- environment()
  location <- function(x = .x) median(x)
  this
}, env = clone.environment(oo))
```

One attractive feature with embedding is the simplicity. All methods and variables are in the same environment. Externally we can access methods and variables by simply using R's `$` notation as shown in the example—there is no need to use more complex constructs such as `with` since execution-time search through multiple environments is unnecessary. From within a method we can use lexical scoping to read methods and variables by just referring to them without any `this` prefix: `.x`. The only exception is that to create or modify a variable from within a method one must preface it with the predefined object identifier, `this`, as in the `bias` method in the example. Alternately, R's double arrow notation (`.x <<- .x + b`) can be used to modify a variable from within

method (but not to create one). Methods in the above example could be invoked like this:

```
# Example. Embedding - invoking methods.
oo$location()
oo$rms()
oo2$location()
oo2$rms()
```

Embedding is also straight-forward using lists. Cloning is as simple as list assignment.

```
# Example: Embedding using Lists.
# copy e components to clone of f
# overwriting same named variables
oo <- list(.x = c(10, 20, 15, 19, 17),
          location = function(this, x = this$.x) mean(x),
          rms = function(this) {.x <- this$.x
                                location <- this$location
                                sqrt(mean((.x-location(this))^2))
          },
          bias = function(this) {this$.x <- this$.x + 1; this}
)
oo2 <- oo # clone, i.e. copy oo to oo2
oo2$location <- function(this, x = this$.x) median(x)
oo2$rms(oo2)
```

Unfortunately embedding is inefficient space-wise in the absence of lazy evaluation features that delay copying of individual list components. Such a feature is found in the Kevo language (Taivalsaari 1993). Although R does not actually make a physical copy of a list when it is copied, as soon as the list is modified—and it will be once the child variables and methods are added in—a physical copy of the entire list is silently performed, not just the components that were modified.

## 5.2 Partial Embedding—Methods Only

In the last section, we copied the variables and methods of the child into a clone of the parent to avoid the self-call problem; however, consider that in the original example that exhibited the self-call problem, we did not have a problem with variables—it was only methods. This suggests that a mixed strategy could be formulated in which we just copy the parent methods to the child but leave the variables in the parent. This provides partial sharing between the parent and the child. It gives us creation-time sharing of methods and *life-time sharing* of variables. Changing a method in the parent after the child is created will not affect the child but changing a variable in the parent will. This is different from the embedding scheme where no change in the parent after creation of the child

affects the child. In embedding there is no sharing of variables or methods after creation of the child.

In this approach, we can use the R `oo2$location` notation for methods; however, read access of variables from outside the object requires `with` notation to cause a search to occur. Alternately, regard variables as encapsulated within methods so that no variable is accessed from outside the object in the first place. From within methods a variable `.x` can be read by just referring to the variable name, e.g. `.x`, without any prefix due to R's scoping rules. In the case that a variable `.x` is to be modified from within a method one must take care that the copy in that method's object is the one that is modified. If no copy currently exists in the object then one one must be created as a child should not be allowed to modify its parent for sake of modularity. Thus, from within a method we write `this$.x <- 3`.

Here is our example redone with the partial embedding approach. Note that the example is identical to the embedding example with environments except for the arguments to `clone.environment`. Also, note the use of `this` in `bias` to force the new value of `.x` to reside in the child so that the child does not modify the parent.

```
# Example: Partial Embedding.
# uses clone.environment from appendix
oo <- local({this <- environment()
            .x = c(10, 20, 15, 19, 17)
            location <- function(x = .x) mean(x)
            rms <- function()
                sqrt(mean((.x - location())^2))
            bias <- function() this$.x <- .x + 1
            this
        })
oo2 <- local({location <- function(x = .x) median(x)
            environment()
        }, env = clone.environment(oo, parent = oo, vars = FALSE))
oo2$rms()
```

### 5.3 Full Delegation

In embedding and partial delegation we made use of R's scoping rules to perform the search for methods and variables in the desired way. The key reason that we could rely on scoping was that all methods were located in the receiver object. In particular, we could simply add a variable called `this` to each object at object creation time and allow any method to use it. In the full delegation model it is no longer true that all methods are located in the same object as the method invoking them. Methods cannot know the receiver simply by virtue of knowing in which object they reside. Methods need an explicit way to identify the receiver object. We pass the receiver around to each method to give it a chance to start its searches from the receiver rather than from the object to

which the method belongs. Within methods, method and variable accesses must be of the form `with(this, stat)`, where `this` refers to the receiver object, in order to ensure that search always starts at the receiver. Furthermore, external references to variables and methods must use `with`, as well, to force search to occur.

```
# Example: Full Delegation -- defining objects.
# uses clone.environment from appendix
oo <- local({.x = c(10, 20, 15, 19, 17)
  location <- function(this,
    x = with(this, .x)) mean(x)
  rms <- function(this) {.x <- with(this, .x)
    location <- with(this, location)
    sqrt(mean((.x - location(this))^2))
  }
  bias <- function(this)
    this$.x <- with(this, .x) + 1
  environment()
})
oo2 <- local({location <- function(this,
  x = with(this, .x)) median(x)
  environment()
}, env = new.env(parent = oo))
```

Unfortunately the necessity to pass around the receiver object in `this` and the repeated use of `with` to force searches from the receiver makes the code itself awkward. Before addressing this let us demonstrate how objects `oo` and `oo2` might be used:

```
# Example: Full Delegation -- calling objects.

# use .x, location and rms in oo
with(oo, location)(oo); with(oo, rms)(oo)

# use .x and rms in oo and location in oo2
with(oo2, location)(oo2); with(oo2, rms)(oo2)

# modifying .x from oo2 creates a .x in oo2 !!!
with(oo2, bias)(oo2)

# use .x and location in oo2 and rms in oo
with(oo2, location)(oo2); with(oo2, rms)(oo2)
```

The syntax above may be a bit awkward but that is solely because we are attempting to directly show the reader how it might work without interposing more utilities and techniques. In actual use, we could use various facilities of R to build an infrastructure to simplify it.

Such streamlining has been implemented in an R package called `proto`. Using `proto`, objects can be created using the `proto` function which is used like `local` but automatically makes available the receiver, `this`, the parent of the receiver, `super`, and the object to which the currently executing method actually belongs, `that`. This eliminates having to explicitly pass these variables through arguments. Also, it overloads `$` so that the notation `oo2$rms()` can be used to search starting with the receiver replacing the bulkier `with(oo2, rms)(oo2)` used above.

Using `proto` from the `proto` package, the example becomes:

```
# Example. Delegation with proto package.
oo <- proto({.x = c(10, 20, 15, 19, 17)
            location <- function(x = .x) mean(x)
            rms <- function()
                sqrt(mean((.x - this$location())^2))
            bias <- function(b) this$.x <- .x + b
})

oo2 <- proto({location <- function(x = .x) median(x)},
            parent = oo)

oo2$rms()
```

The key internal functionality lies within the implementation of the `$` operator. An expression such as `oo$rms()` is implemented in terms of the R `get` function which searches for method `rms` starting with the receiver `oo` and proceeding upwards through the ancestor environments. Once the method is located, it is copied to a new temporary method with the same functionality but whose environment has been set to a proxy environment whose only components are `this`, `super` and `that`. The parent of the proxy environment is then set to the receiver. Finally, this newly constructed method is executed.

It should be noted that despite these simplifications it is still necessary to use the form `this$location` for making self-calls in order to ensure that the called method gets the appropriate proxy environment. Also note that the effect of the processing described in the previous paragraph is to *bind* the receiver to the method at dispatch time. This is the key difference between full delegation and the other approaches. In embedding and partial delegation, the receiver environment always equals the currently executing method's environment so the receiver can be bound at object creation time whereas with full delegation it is bound at dispatch time.

From this point on we shall make use of the `proto` package to simplify our presentation.

## 5.4 Fragile Base Object Problem

The fragile base object problem is the counterpart to the fragile base class problem in class-based object-oriented programming (Aldrich 2004). To understand this problem compare the following two versions of the same object:

```
# Example. Delegation with proto package.
oo.a <- proto({.x = c(10, 20, 15, 19, 17)
  location <- function(x = .x)
    (mean(x) + median(x))/2
  rms <- function()
    sqrt(mean((.x - this$location())^2))
})

oo.b <- proto({.x = c(10, 20, 15, 19, 17)
  location <- function(x = .x)
    (mean(x) + median(x))/2
  rms <- function()
    sqrt(mean((.x - (mean(.x)
    + median(.x))/2)^2))
})
```

`oo.a` makes a self-call to `location` from `rms` whereas `oo.b` computes `rms` without making any self-calls at all. Both versions have identical functionality in that they both give the same outputs for the same inputs. However, if we create a child object `oo2` which has a method:

```
location <- function(x = .x) median(x)
```

then that child object will have different behaviors for the two parent objects. If `oo.a` were the parent object then `oo2$rms` would be calculated relative to the median due to the self-call; however, if `oo.b` were the parent object then `oo2$rms` would continue to be calculated relative to the mean as `oo.b$rms` does not make a self-call.

The problem with this example is that the external behavior of the child depends on the internal behavior of the parent. In order for code not to be brittle we want the correctness of the code to depend on the external but not the internal behaviour of objects. The idea is that if we want to be able to easily change a system in a modular way then the various parts of the system should be as decoupled as possible. If we change the external behavior of an object then we can reasonably expect that we may have to change clients of it; however, we would like to be able to change the internals independently of the rest of the system (Parnas 1972). Without this guarantee, as object hierarchies get larger and larger there is an ever increasing chance that an internal change in one object will affect the behavior of another object in an unexpected way. This dependence on internals is the fragile base object problem.

In `oo.a`, we regard `rms` as a deviation measure around the value produced by the `location`. In `oo.b` we regard `rms` as an independent scale measure. However, since `oo.a` and `oo.b` have identical functionality we easily could have used one when we meant the other. Even if we understand this difference and are resigned to having the internals affect the delegation behavior we see a problem. In the case of `oo.b`, we only avoided having `rms` depend on `location` by redundantly placing the code for `location` both in `location` itself and in `rms`. To allow one to avoid the redundant code and yet not inadvertently override a method, there is a three argument form of `this$location` in which the receiver and the object from which search is to start can be separately specified. A special variable `that` is provided which refers to the object containing the current method. Using these constructs we have:

```
oo.c <- proto({.x = c(10, 20, 15, 19, 17)
  location <- function(x = .x) (mean(x) + median(x))/2
  rms <- function() {
    # like that$location passing this as receiver
    location <- "$.proto"(that, location,
                          receiver=this)
    sqrt(mean((.x - location())^2))
  }
})
```

The effect of specifying that search is to start at `that` is that the self-call to `location` will not be overridden. Note that in `oo.c` we only wrote `(mean(.x) + median(.x))/2` once avoiding the redundancy. Thus we have:

```
oo2 <- proto({
  location <- function(x = .x) median(x)
}, parent = oo.c)

oo.c$location() # uses (mean + median)/2
oo2$location() # uses median
# oo.c$rms uses non-overridden self-call to location
oo.c$rms() # uses (mean + median)/2 for location
oo2$rms() # uses (mean + median)/2 for location
```

The self-call `"$.proto"(that, location, receiver = this)` calls `location` starting its search at `that` passing it receiver `this`. The expression is similar in intent to the `with(that, location)(this)` expression that might have been used with our earlier examples that used environments directly had `that` been available as an environment variable. Because the search starts at `that`, which in this case is `oo.c`, we are guaranteed that the self-call will not be overridden by a child.

Another design might have been to associate a non-overridable property with the called method rather than specify it through the calling sequence; however, the three argument form of `$` has the advantage that it can be used in other contexts too, as we shall see.

## 5.5 Explicit Delegation

In the situations above, we automatically searched the parent if the child did not have a method or variable. Another possibility is to search only the object and not automatically go further up the search path. If the user wants to search up the ancestor path for a method that is not in the child then the user must write a wrapper function which calls the parent explicitly.

### Multiple Delegation

One possible use for the explicit delegation approach occurs in the situation where we would like to delegate from two different parents. Our implementations so far have only allowed an object to have a single parent.

For example, suppose that we have a `bias.` object from which we wish to delegate the `bias` method. This could be implemented like this:

```
# Example. Explicit delegation.
bias. <- proto({bias <- function(x, b) x + b })
oo <- proto({.x = c(10, 20, 15, 19, 17)
            location <- function(x = .x) mean(x)
            rms <- function()
                sqrt(mean((.x - this$location())^2))
            bias <- function(b)
                this$.x <- bias.$bias(.x, b)
            })
oo2 <- proto({location <- function(x = .x) median(x) },
            parent=oo
            )
oo2$location()
oo2$bias(1)
oo2$location()
```

Note that `bias.$bias` does not have access to the variables and methods of `oo2` so we must pass and return them explicitly. This is not true multiple delegation in which multiple parents are automatically searched but given only single delegation it is a workaround that may be adequate in many cases. The advantage of keeping with single delegation and using such workarounds is that confusion due to name clashes, where multiple parents have variables or methods with the same name, is avoided.

### Proxies

A second application of explicit delegation is proxies. These are objects which sit between two other objects and mediate the interaction. Suppose we wish to monitor those invocations of `rms` in parent `oo` that come from child `oo2`. We can create an object that sits between `oo` and `oo2` and intercepts all calls to `oo$rms` coming from `oo2` (or any child of `oo2`). We can do this without modifying the

objects on either side – other than the parent link in the child. The following example, displays `rms` called each time `oo$rms` is invoked from `oo2`.

```
# Example: Proxy.
proxy <- proto({
  rms <- function() {
    print("rms called")
    parent <- that$parent()
    "$.proto"(parent, rms,
              receiver=this)()
  }
},
parent = oo
)
# reset parent of oo2 to proxy
oo2$parent(proxy)
oo2$rms() # will trigger proxy prior to oo$rms
```

The `proxy` object is defined using `proto` like any other object and its parent is set to `oo`. The call to `oo2$rms` finds no `rms` in `oo2` and so searches the parent of `oo2` finding `proxy$rms`. Finally, `proxy$rms` issues a print statement and calls `rms` in its parent using the three argument form of `$` specifying that search is to start at its parent.

## 5.6 Dynamic Delegation

One intriguing possibility in the prototype-based case is dynamically changing the parent on-the-fly. We have already seen an example of this in the last section where we reset the parent of an object to a proxy of the parent.

A typical application occurs when the object can be in one of the several modes. For example, consider a Markov chain with states A and B. When in

state A we change state with probability 0.25 and when in state B we change state with probability 0.40. The following simulates such a Markov chain.

```
# Example: Dynamic Delegation. Markov Chain.
state.A <- proto({id <- "A"; p <- .25
  next.state <- function()
    if (runif(1) < p) state.B else state.A
})
state.B <- proto({id <- "B"; p <- .40
  next.state <- function()
    if (runif(1) < p) state.A else state.B
})
chain <- proto({
  show <- function() print(id)
  step <- function() chain$parent(next.state())
}, parent = state.A)
chain$show()
for(i in 1:10) {chain$step(); chain$show()}
```

In this example, the two objects, `state.A` and `state.B`, each represent a state. Each has a character `id` and a `next.state` method which can be invoked to find the next state. Whether the Markov chain is deterministic or non-deterministic and other aspects of changing state are encapsulated in this function. The `chain` object has two methods which display the current state and step forward one iteration. The current state is the parent object of `chain`. Stepping forward invokes whichever `next.state` method is found in the current parent of `chain`. That invocation returns the new state and the parent of `chain` is reset to it.

## 5.7 Comparison to Class-Based Systems

Object-based systems can emulate class-based systems but the reverse is not readily obtained. In this section we illustrate how prototypical objects can be used in place of classes. This will be the preferred style; however, for completeness, we also show the process of creating class-based constructs from prototype-based ones.

### Prototypical Objects

We illustrate a prototypical irregular time series object and how we can stamp out several copies, all without classes. The `TS` object contains a method to extract times and a method to calculate a two sided moving average. We shall simply use `TS` as a virtual object whose children, `ts1`, `ts2`, `ts3` are the actual

time series. One of the children, `ts3` is actually a regular time series which is implemented by overriding `times`.

```
# Example. Irregular and regular times series
TS <- proto({
  times <- function() times.
  mov <- function(delta) {tt <- this$times()
    sapply(tt, function(x)
      mean(values.[abs(tt - x) <= delta]))
  })
ts1 <- proto({values. <- times. <- 1:5}, parent=TS)
ts2 <- proto({values. <- 1:5;
  times. <- c(1, 2, 4, 5, 6)}, parent=TS)
ts3 <- proto({values. <- 1:10; times. <- c(1, 10, 1)
  times <- function()
    seq(times.[1], times.[2], times.[3])
  }, parent=TS)

ts1$mov(1); ts2$mov(1); ts3$mov(1)
```

A second possibility, not illustrated here, would be to place the methods and variables in the same object `ts1`, say, and then clone and it to produce `ts2` and `ts3` and then modify each of those objects. With this approach each object would contain all its methods and variables.

## Classes

We now redo the above example using classes showing how classes can be emulated using objects.

Recall that a class is a mechanism for creating object instances of a standard form. Subclasses are classes which inherit attributes and methods from superclasses. Class variables are variables that belong to a class rather than an object.

To emulate this with objects, we define a class as an object which generates child objects of a particular form. Class variables and class methods (as opposed to variables and methods in the objects produced by classes) are simply variables and methods that belong to the class object. A superclass is simply a parent of the class. Thus we have: object is child of class is child of superclass.

Each class will construct new objects by having a method called `new` which returns a new instance object for that class. `new` is an example of a class method.

We will use delegation for the classes themselves and embedding for the objects they produce.

We now illustrate a class as an object and its `new` method which constructs objects that belong to that class.

```
# Example. Class.
TS <- proto({
  new <- function(vv, tt)
    proto({values. <- vv; times. <- tt
      times <- function() times.
      mov <- function(delta) {
        tt <- this$times()
        sapply(tt, function(x)
          mean(values.[abs(tt - x) <= delta]))
      })})
ts1 <- TS$new(1:5, 1:5)
ts2 <- TS$new(1:5, c(1, 2, 4, 5, 6))
ts1$mov(1)
ts2$mov(1)
```

Note that structurally the above code is a `proto` in a function in a `proto`. The function executes the inner `proto` to create an object. Just wrapping the inner `proto` in a function would have been sufficient to stamp out new objects; however, we wrap that in an outer `proto` too so that the class itself is an object—not just a function. By making the class an object, it allows us to use the existing delegation mechanism with classes so that we can have subclasses and superclasses. For example, we can define a class `TS.Reg` for regular time series, i.e. equally-spaced times, which is a subclass of `TS` like this:

```
# Example. Subclass.
TS.Reg <- proto({
  new <- function(vv, tt)
    proto({
      times <- function()
        seq(times.[1], times.[2], times.[3])
    }, embedin=TS$new(vv, tt)), parent=TS)

ts3 <- TS.Reg$new(1:10, c(1, 10, 1))
ts3$mov(1)
```

Note that we have specified `parent=TS` as the second parameter to the `proto` that defines the class `TS.Reg` (i.e. the outer `proto`) so that `TS.Reg` becomes a child of `TS`. Also, we have specified the `embedin = TS$new(vv, tt)` as the second parameter to the `proto` that defines instance objects (i.e. the inner `proto`). This creates a new parent object and then embeds the `TS.Reg` object into it.

Note that we set up an entirely new class even though just one object, `ts3`, deviated slightly from the others. In a prototype-based approach we can just modify that one object but in a class-based approach we must go to the bother of setting up a new class, even if there is just one instance.

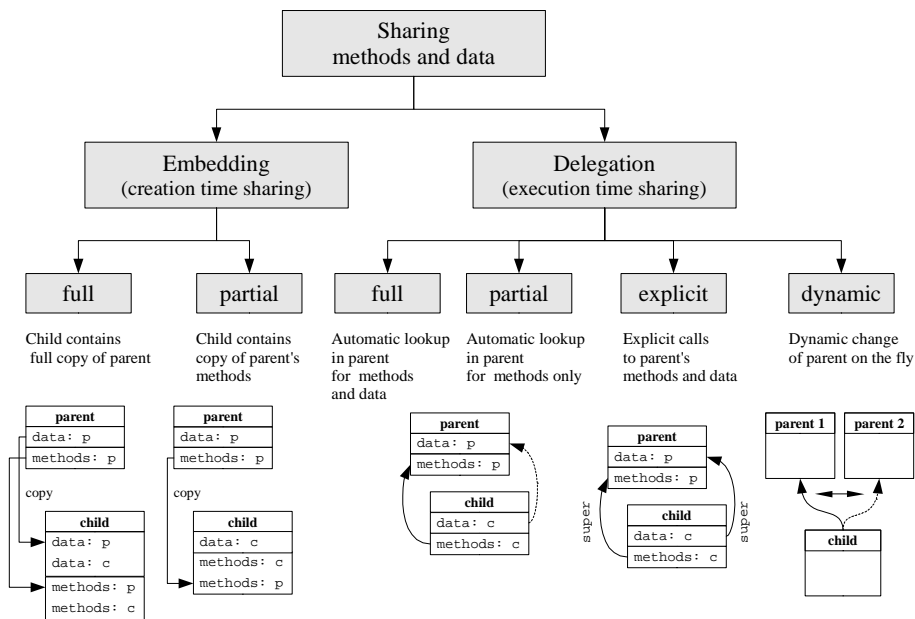


Figure 2: Sharing methods and data in prototype-based programming

## 6 DISCUSSION

As we have shown, multiple ways of data and method sharing exist in prototype-based programming. These can be classified into two groups: embedding and delegation (figure 2).

### Embedding

Although the sharing model of embedding is less dynamic there can be some advantages. In embedding it is possible to remove or rename a variable that came from the parent. With delegation, removing a variable from a child might just expose the corresponding variable in the parent rather than remove it. Also, although the ability to change the parent after creation of the child (which is possible in delegation but not embedding) is powerful it can also lead to errors if changes are made that were not anticipated.

Note that delegation is possible even in the embedding approach. The methods of the child can still call the methods of the parent object explicitly, if desired. The main limitation is that with explicit delegation, parent methods which call other parent methods will not be overridden by child methods (the problem that we started with). This is no worse than in class-based inheritance where an object of a child class can call an object from a parent class but the child object cannot override the methods of the parent object with the methods

of the child once those particular objects have been instantiated. (Note carefully that this last sentence refers to the inability of child objects in class-based programming to override parent objects. Child classes can still override parent classes.)

Partial embedding where the methods are embedded but the variables are delegated is a particularly attractive compromise for R in that minimal infrastructure is required to implement it. Changing a method in the parent, which is not that likely, is the only advantage given up and even that is possible, with some limitation, using explicit delegation. This scheme still allows the child methods to be changed and it allows for shared memory for variables. It is simpler than full delegation since the environments for methods do not have to be dynamically set as they do in full delegation; rather, they are set at object creation time once and for all.

## Delegation

Full delegation is how most prototype-based systems work. This is the most powerful paradigm as it is the most dynamic allowing both variables and methods to be changed in both children and parents with the effect in parents percolating down to all children.

Unlike embedding, where method dispatch is fixed at object creation time, in full delegation method dispatch must take into account the receiver object at each method call. Our implementations carry this out by modifying the method environment dynamically or passing around the environment in the `this` variable.

## Other Languages

The early prototype-based language, Self (Agesen et al. 1992), uses full delegation and most other prototype-based languages have followed its lead. One of the few prototype-based languages to have used embedding is Kevo (Taivalsaari 1993). Kevo delays actual copying of contents until the values of a parent and child component actually diverge. Until that occurs, references are used to point to the same contents. Thus, to the user it appears that there are independent copies yet the copying and space overhead is saved unless actually needed.

## 7 CONCLUSION

Prototype-based programming could be usefully employed in statistical computations. Such systems have fewer primitives and can be simpler, particularly, in the case that there are only a few objects of a particular sort. R provides a convenient framework for creating and understanding prototype-based systems. Both delegation and embedded models are readily implemented in terms of R constructs making them readily available for use. Prototype-based pro-

gramming could be a potentially widely applicable concept to use in organizing statistical computations in R and other computing environments.

## APPENDIX: CLONE ENVIRONMENT

The `clone.environment` utility is shown here:

```
# Copy contents of environment e1 to e2. Create e2 if its omitted.
# Set environment of functions to target environment as they are copied.
# If vars = FALSE then only methods are copied else functions too.
clone.environment <-
function(e1, e2 = new.env(parent = parent),
        parent = parent.env(e1), vars = TRUE) {
  stopifnot(is.environment(e1))
  for(s in ls(e1, all = TRUE))
    if (is.function(e1[[s]])) {
      assign(s, e1[[s]], e2)
      eval(substitute(environment(s) <- e,
                      list(s = as.name(s), e = e2)), e2)
    } else if (vars)
      # vars = TRUE if variables as well as fns to be copied
      assign(s, e1[[s]], e2)
  e2
}
```

## References

- Agesen, O., Bak, L., Chambers, C., Chang, B.-W., Hölzle, U., Maloney, J., Smith, R. B., and Ungar, D. (1992), *The SELF Programmer's Reference Manual*, 2550 Garcia Avenue, Mountain View, CA 94043, USA, version 2.0.
- Aldrich, J. (2004), "Selective Open Recursion: A Solution to the Fragile Base Class Problem," Submitted, <http://www-2.cs.cmu.edu/~aldrich/papers/selective-open-recursion.pdf>.
- Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988), *The New S Language*, London: Chapman & Hall.
- Bengtsson, H. (2003), "The R.oo Package – Object-Oriented Programming with References Using Standard R Code," in *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*, eds. Hornik, K., Leisch, F., and Zeileis, A., Vienna, Austria.
- Chambers, J. M. (1998), *Programming with Data: A Guide to the S Language*, New York: Springer-Verlag.
- Chambers, J. M. and Lang, D. T. (2001), "Object-Oriented Programming in R," *R News*, 1, 17–19, <http://cran.R-project.org/doc/Rnews/>.

- Dony, C., Malenfant, J., and Bardou, D. (1999), “Classifying Prototype-Based Programming Languages,” in *Prototype-Based Programming: Concepts, Languages and Applications*, eds. Noble, J., Taivalsaari, A., and Moore, I., Singapore: Springer-Verlag, chap. 2, pp. 17–45.
- Dony, C., Malenfant, J., and Cointe, P. (1992), “Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation,” in *Proceedings of OOPSLA '92*, Vancouver, Canada: ACM Sigplan Notices, vol. 27(17), pp. 201–215.
- Ducasse, S. (2003), “A closer look at prototype-based languages,” <http://www.iam.unibe.ch/~scg/Teaching/IOOM/PPT/PrototypesAdvanced.pdf>.
- Gentleman, R. and Ihaka, R. (2000), “Lexical Scope and Statistical Computing,” *Journal of Computational and Graphical Statistics*, 9, 491–508.
- Ihaka, R. and Gentleman, R. (1996), “R: A Language for Data Analysis and Graphics,” *Journal of Computational and Graphical Statistics*, 5, 299–314.
- Kelsey, R., Clinger, W., and Rees, J. (1988), *Revised(5) Report on the Algorithmic Language Scheme*, <http://swissnet.ai.mit.edu/~jaffer/Scheme>.
- Lieberman, H. (1986), “Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems,” in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ed. Meyrowitz, N., New York, NY: ACM Press, vol. 21(11), pp. 214–223.
- Noble, J., Taivalsaari, A., and Moore, I. (1999), *Prototype-Programming*, Springer-Verlag Singapore Pte. Ltd.
- Oehlert, G. W. and Bingham, C. (1998), “MacAnova User’s Guide,” Tech. Rep. 617, School of Statistics, University of Minnesota, Minnesota, St. Paul.
- Parnas, D. L. (1972), “On the Criteria to be Used in Decomposing Systems into Modules,” *Communications of the ACM*, 15, 1053–1058, <http://www.acm.org/classics/may96/>.
- R Development Core Team (2004), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, ISBN 3-900051-00-3, <http://www.R-project.org>.
- Shalit, A. (1996), *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*, Addison Wesley.
- Steele Jr., G. L. and Sussman, G. J. (1978), *The Revised Report on Scheme, a Dialect of Lisp*, MIT Artificial Intelligence Memo 452, January 1978.
- Sussman, G. J. and Jr., G. L. S. (1975), *Scheme: an interpreter for extended lambda calculus*, MIT Artificial Intelligence Memo 349, December 1975.

- Taivalsaari, A. (1993), “A Critical Review of Inheritance and Reusability in Object-Oriented in Object-Oriented Programming.” Ph.D. thesis, University of Jyväskylä, Finland, ISBN 951-34-0161-8.
- (1996a), “Classes vs. Prototypes Some Philosophical and Historical Observations,” *Journal of Object-Oriented Programming*, 10, 44–50.
- (1996b), “On the notion of inheritance,” *ACM Computing Surveys*, 28.
- Tierney, L. (1990), *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, Wiley, New York, NY.
- (1995), “Recent Developments and Future Directions in Lisp-Stat,” Tech. Rep. 608, School of Statistics, University of Minnesota, Minnesota, St. Paul.
- Venables, W. N., Smith, D. M., and the R Development Core Team (2004), *An Introduction to R: A Programming Environment for Data Analysis and Graphics*, ISBN 3-900051-05-4, <http://www.R-project.org>.