Mutable objects in R

Hadley Wickham

December 15, 2010

Abstract

Programming paradigms help us understand the differences and similarities between fundamental choices in language design. This paper looks at R in the context of three paradigms of object oriented programming: mutable vs. immutable objects, generic-function vs message-passing methods, and class-based vs. prototype-based inheritance.

The paper also describes a new OO package for R, mutatr, which provides mutable objects with message-passing methods and prototype-based inheritance. The mutatr package is available on CRAN.

1 Introduction

A programming paradigm is a fundamental style of computing programming, like object-oriented, functional, declarative, procedural and logical. Knowing what paradigms a language supports allows us to quickly get a feel for the attributes of that language. This paper explores some lower level paradigms of object oriented (OO) design. This will help us understand some of consequences of design decisions made when R was written, and will identify areas of interest for future exploration.

The paper is broken down as follows. Section 2 discusses the programming paradigms that impact the design of an object system. I compare and contrast mutable vs. immutable objects, message-passing vs. generic-function methods, and class-based vs. prototype-based inheritance. Section 3 introduces a new OO package for R, `mutatr`, available from CRAN. It provides mutable objects with message-passing methods and prototype-based inheritance. Three case studies that these features shown in Section 4 and Section 5 wraps up the paper with general conclusions and suggestions for future directions.

2 Paradigms

Programming paradigms help us to understand the high-level differences and similarities between different programming languages. A good knowledge of paradigms is useful when pairing a programming language (or very general approach) with a particular problem. This paper discusses three of the many paradigms, focussing on those of importance when designing an object oriented system:

- Mutable vs. immutable objects, Section 2.1.
- Generic-function vs. message-passing methods, Section 2.2.
- Class-based vs. prototype-based inheritance, Section 2.3.

Table 1 shows how R's OO systems fall into these categories. This table is somewhat of a simplification because both S3 and S4 can be used in conjunction with R's reference-based environment object to produce mutable objects (this is how R.oo works), but it reflects common practice. The creation of ad hoc OO systems

System	Mutability	Methods	Inheritance	Reference
S3	immutable	generic functions	class-based	
S4	immutable	generic functions	class-based	
R.oo	mutable	generic functions	class-based	Bengtsson (2003)
OOP	mutable	message passing	class-based	Chambers and Temple Lang (2001)
proto	mutable	message passing	prototype-based	Kates and Petzoldt (2007)
mutatr	mutable	message passing	prototype-based	

Table 1: Object oriented systems in R.

using lexical scoping is also common, as typified by Section 2.3 of Ihaka and Gentleman (1996) and Section 4 of Gentleman and Ihaka (2000)

If you are interested in learning more, a brief overview is available in Van Roy (2009), and a full exposition in van Roy and Haridi (2004). These resources are highly recommended reading for understanding the tradeoffs of the programmatic paradigms that R, or any other language, has adopted.

2.1 Mutable vs immutable

R has many similarities to a functional programming language, and supports a functional programming style. A functional style strives to stay close to the mathematic definition of a function, that is, a fixed relation between an input and output set. In R, most functions return the same output when given the same input, and their only way of interacting with the outside world is through their return value. There are exceptions, such as loading data or exporting graphics which must affect the world outside R, and pseudo random number generators which are not so useful if they always return the same "random" numbers.

Unlike other programming languages, it is difficult for a function to modify its arguments (very little is impossible in R because it is so flexible, but this is generally true). Figure 1 shows a simple example. If you ran this code, you would see that `l$a` is not modified. Here an imperative interface masks an underlying functional approach. Internally, `x$a <- 2` is a combination of two steps: first construct a modified copy of `x`, then change the binding of `x` to the new object; the bindings change, but the original `x` does not. This makes R objects immutable: whenever it looks like you are modifying an object, you are actually creating a modified copy. There are a few exceptions to this rule, such as external pointers, environments and weakrefs, but these are typically only used by those modifying the internals of R, or when connecting R to a programming language with non-functional semantics.

```
l <- list(a = 1)
f <- function(x) {
  x$a <- 2
}
l$a
# [1] 1
f(l)
l$a
# [1] 1
```

Figure 1: When an R function tries to modify its inputs, it instead creates a modified copy. In this example, the value of `l$a` remains unchanged. Expression outputs are shown in comments.

Systems that use primarily immutable objects are sometimes called pass-by-value or copy-on-modify, while those that use mutable objects are called pass-by-reference. While technically correct, these terms

are confusing because they imply that this is a language-level behaviour when it is actually object-level: a language can have both mutable and immutable objects.

Code that uses immutable objects tends to be easier to reason about because effects are local. This is a big advantage: it is easier to understand how code works, and to test (or even prove) its correctness. The disadvantage of immutability is its lack of modularity: since we can not modify objects directly, each function needs to return a copy of all objects that it modifies. For example, imagine we wanted to memoise a function. Memoisation stores the results of a function so that if it is called again with the same arguments, a cached result is returned. This is an example of the classic space-speed tradeoff; here we are trading space (the in-memory cache) for speed. To do this without mutable objects we need an additional argument to the function to store the cache, and to return both the result and the updated cache from the function. R code to do this is shown in Figure 2.

```
slow_function <- function(x) {
  Sys.sleep(2)
  x ^ 2
}
system.time(slow_function(2))
system.time(slow_function(2))

slow_function_memo <- function(x, cache = list()) {
  if (x %in% names(cache)) {
    res <- cache[[as.character(x)]]
  } else {
    Sys.sleep(2)
    res <- x ^ 2
    cache[as.character(x)] <- res
  }
  list(result = res, cache = cache)
}

system.time(res <- slow_function(2))
system.time(res <- slow_function(2, res$cache))
```

Figure 2: Implementing memoisation without mutable objects requires that the memoised function returns both the original result and the updated cache, and calling the function requires explicit reference to the cache.

This is unappealing because the behaviour of the function has not changed (it still returns the same inputs for the same outputs), but its interface has. Making this change violates modularity: we need not only to modify this function, but every function that calls it, and so on down the call tree. Tangling specification with implementation makes modularity hard. If we have mutable state, there is no need to expose the difference in implementation to the user, and in fact it becomes easy to write a general memoisation function, as shown in Section 4.1.

Mutable state increases modularity, but it also adds non-local effects, which make it much harder to understand how programs work. Is it possible to enjoy the best of both worlds? van Roy and Haridi (2004, p. 412) suggest to keep as much code as possible functional, and only use mutable state when absolutely necessary, and when it is necessary, it is advantageous to "write stateful components so that they behave declaratively" (van Roy and Haridi, 2004, p. 417). For example, memoisation can use mutable state internally, but not expose this implementation detail to the outside world, so reasoning about the function can be

performed in a pure framework.

Immutable objects do not have to cause poor performance because of they must be copied whenever they are modified. It is possible to have immutable objects with the same performance characteristics as mutable objects (Okasaki, 1999; Kaplan, 1995). The study of how to do this efficiently is the domain of persistent data structures (Driscoll et al., 1986). Most persistent data structures work by storing a reference to the previous version and a list of changes. The programming language Clojure (Hickey, 2008) uses this technique extensively. These techniques have yet to be implemented in R.

2.2 Message-passing vs generic-function methods

Object oriented programming is based around the idea of methods, functions that are selected based on the class of their parameters. There are two fundamentally different styles of methods depending on whether they are based around functions or objects. The function-centric style is called generic-functions (GFOO), and the object-centric style is called message-passing (MPOO).

In GFOO, methods are based around generic functions that dispatch on the class of any (or all) of their parameters. In R, S3 and S4 are both generic function systems, based on that of Dylan (Feinberg et al., 1997), which in turn was inspired by the common lisp object system (CLOS) (DeMichiel and Gabriel, 1987). Figure 3 illustrates the use of generic functions to write a vehicle inspection function. Because generic functions can dispatch on all of their arguments, it is easy to specialise the function for different types of vehicles (a general procedure, plus specific procedures for cars and trucks), and for different types of inspectors (as well as all other tests, the state inspector also checks the insurance). The most specific procedure is called first, and then `callNextMethod()` ensures the more general methods are invoked.

In MPOO, messages are sent to objects, and the messages are interpreted by object-specific methods. This means that the method selection is based only on the object to which the object is sent, not the parameters of the message. Typically this object has a special appearance in the method call, usually appearing before the name of the method/message. MPOO was used by the first object-oriented programming languages, Simula and smalltalk, and their legacy remains today: most popular OO languages, like Java and C#, are message passing.

There are advantages and disadvantages to each style of OO. The big advantage of the GFOO approach is that it can choose a method based on any number of the arguments. This is particularly useful for certain tasks. For example, the matrix package (Bates and Maechler, 2009) has 12 class of matrices to concisely represent patterned matrices. When performing matrix multiplication it is very useful to be able to dispatch on both matrices to take advantage of all possible optimisations. This is difficult to do with single dispatch, although there are some workarounds, like double dispatch.

There are two small advantages of MPOO, name-spacing and nesting. MPOO is implicitly name-spaced; because you are sending a message to an object, different objects can interpret the same message differently. GFOO requires a global namespace that all methods have to share, and while the explicit namespace support in R does help, it can be frustrating to discover the method name that you want has been claimed by another package with different semantics. Another minor advantage of MPOO is the order of function composition leads to constructs that can be read from left-to-right. Figure 4 shows an example. It is easier to read the MPOO (imaginary syntax, not working code) because we can read from left-to-right: take the file, open it, read the first five lines and replace spaces with dashes. Converting the GFOO syntax to English yields: perform a text replacement on the lines read from the opened file, but just the first five, changing spaces to dashes.

These advantages are much smaller than multiple dispatch, but they accrue more frequently. Implicit name-spacing makes code faster to write because you don't need to think so hard about creating globally unique names, and the order of function composition makes it more natural (at least for speakers of European languages) to read. While multiple dispatch solves certain problems very elegantly, it is not needed for most tasks.

```
setGeneric("inspect.vehicle",
  function(v, i) {
    standardGeneric("inspect.vehicle")
  })

setMethod("inspect.vehicle",
  signature(v = "vehicle", i = "inspector"),
  function(v, i) {
    look.for.rust(v)
  })

setMethod("inspect.vehicle",
  signature(v = "car", i = "inspector"),
  function(v, i) {
    callNextMethod() // perform vehicle inspection
    check.seat.belts(car)
  })

setMethod("inspect.vehicle",
  signature(v = "truck", i = "inspector"),
  function(v, i) {
    callNextMethod() // perform vehicle inspection
    check.cargo.attachments(v)
  })

setMethod("inspect.vehicle",
  signature(v = "car", i = "state.inspector"),
  function(v, i) {
    callNextMethod() // perform car inspection
    check.insurance.car(v)
  })
```

Figure 3: An example of S4 generic functions. Example adapted from `http://www.opendylan.org/gdref/tutorial.html`.

```
file$open()$read_lines(1:5)$replace(" ", "-")
replace(read_lines(open(file), 1:5), " ", "-")
```

Figure 4: A comparison of an (imaginary) MPOO syntax (top) vs. the standard R GFOO syntax (bottom).

2.3 Class-based vs prototype-based inheritance

There is a choice for how inheritance works in OO system: is it class-based or prototype-based? Class-based inheritance erects a firm boundary between classes and instances. Typically classes are generated statically at compile-time, while instances are generated dynamically at run-time. There are no inheritance relationships between objects, only between classes. Prototype-based inheritance blurs this boundary by making it possible for objects to inherit from other objects. This has two consequences: the distinction between class and instance is blurred, and the objects that play the role of classes become much more dynamic.

Prototype-based inheritance is particularly well suited for exploratory statistical and scientific programming when the target is not well known in advance. Instead of expending a large amount of upfront effort to come up with a suitable class structure for the problem domain, much of which might end up to be wrong or unnecessary, you can gradually build up interfaces as you discover them, and you are not locked down too soon. Hofstadter (1979) calls this the prototype principle: "The most specific event can serve as a general example of a class of events". There is some evidence that the prototype model better models how our brain works (Hofstadter, 1979, p. 352), building up general models from specific examples.

Notable prototype-based languages are Javascript, a popular web programming language, and Self, the first language to experiment with prototypes (Smith and Ungar, 1995; Ungar et al., 1991; Ungar and Smith, 1991). These languages are both MPOO. Prototype-based inheritance with generic functions has been implemented in Slate (Rice and Salzman, 2004; Salzman and Aldrich, 2005), and Cecil (Chambers, 1992). Class-based systems are more common: Java, C++ and C# all use class-based inheritance. Examples of prototype based code are shown in the next section.

3 Introduction to mutatr

The ideas of mutable state, message-passing methods and prototype-based inheritance are implemented in a new R package, mutatr. It explores different trade offs to existing R packages for OO programming:

- It separates lexical and object scope, based on ideas from javascript.
- Mutable, multiple inheritance, using a depth-first search of inheritance graph, based on ideas from io (Dekorte, 2005).
- Rich introspection, also based on io.

This section gives a brief introduction to the package, showing how the theoretical ideas described above come to life in code. The fact that it is possible to use environments and S3 methods to write an alternative OO system is a testament to the tremendous flexibility of the language.

3.1 Basic use

At the most basic level, there are only two things you need to know to use mutatr: how to add fields (methods and variables) to an object and how to create new objects.

To create a new object, call the `clone()` method on the object you want to inherit from. The only object mutatr provides is `Object`, so to create your first object you will use `Object$clone()`. This call also illustrate hows to access object methods, like accessing a component of a list. The following example shows how you might use mutatr to model one aspect of a dog's behaviour.

```
Dog <- Object$clone()
Dog$speak <- function() cat("Woof!\n")

mina <- Dog$clone()
mina$speak()
```

Note the convention in object names: objects used more like classes (i.e. to provide common methods for other classes) start with a upper-case letter, and objects that behave more like instances start with a lower-case letter.

To access object properties within a method, use `self`. The following example modifies speak to record how many times it is called.

```
Dog$speak_count <- 0
Dog$speak <- function() {
  self$speak_count <- self$speak_count + 1
  cat("Woof!")
}

mina$speak()
mina$speak()
mina$speak_count

barkeley <- Dog$clone()
barkeley$speak_count
```

Because new slots are always created in the scope of the object, this method behaves correctly when we create another dog: it has an independent speak counter. Section 4 for more examples.

When a function is assigned into a field within a mutatr object, it automatically becomes a method. Section 3.3 describes the consequences of that action. Roughly speaking the evaluation context of the function does not change (it can still use variables from its local environment) but it gains access to a special `self` variable that refers to the object it belongs to.

3.2 How it works

Mutatr is based around a core object that implements a minimal set of methods necessary to bootstrap out of R's classical scoping rules: `has_local_slot()`, `get_local_slot()`, `set_slot()`, `remove_slot()`, and `clone()`. For this reason, the core object is not a regular mutatr object, but is implemented with closures.

Mutatr supports dynamic multiple inheritance, so each object can have multiple parent prototypes, and we get an inheritance graph rather than a tree. Multiple inheritance is more general than single inheritance, but it does give rise to substantial possibilities for confusion. Figure 5 illustrates some of these. It is easy to come up with complex inheritance structures on paper (e.g. circular) but harder to come up with examples that arise as a result of modelling real objects. Self (Agesen et al., 2000), a prototype-based language with multiple inheritance, uses conventions to help ensure that multiple inheritance is helpful, not harmful. All objects are either classed as either traits or mixins. A trait has a single parent trait, and can have multiple mixins; and a mixin has no parents. Traits act like classes, while mixins provide functionality for cross-cutting concerns.

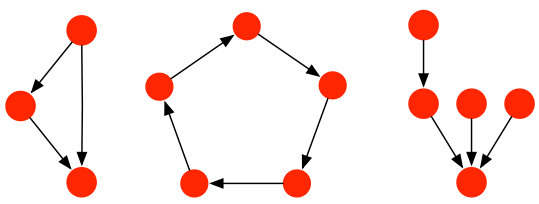

Figure 5: Three types of inheritance.

Implementing inheritance on top of this core requires two sets of methods: one to modify the parents of an object, and the other to create an iterator that will expose all ancestors without getting stuck in a endless loop. Three methods modify parents, allowing the inheritance graph to be dynamically changed:

- `append_proto`: adds a new parent to the beginning of the list
- `prepend_proto`: adds a new parent to the beginning of the list
- `remove_proto`: remove an parent

The ancestor iterator performs a depth-first search of the inheritance graph, ensuring that each node (ancestor prototype) is only visited once. This is the same model that io uses.

With these two components in place we can add `has_slot()` and `get_slot()`. The essence of `get_slot()` is shown below. The code is simple: we just iterate through all the ancestors, looking for the first ancestor that has a slot of the desired name, then return it with the correct scope. (The actual code is somewhat more complicated for performance reasons).

This method is aliased to `$.mutatr`, and similarly `set_slot()` is aliased to `$<-.mutatr()`. This supports the use of usual R subsetting functionality.

```
get_slot <- function(obj, name, scope = obj) {
  iter <- ancestor_iterator(obj)
  while(iter$has_next()) {
    ancestor <- iter$get_next()

    if (core(ancestor)$has_local_slot(name)) {
      res <- core(ancestor)$get_local_slot(name)
      res <- object_scope(res, scope)
      return(res)
    }
  }
  NULL
}
```

Currently, there is no way to explicit access methods of parent objects outside of normal inheritance. This is a challenge for future work.

3.3 Scoping

There are two parallel sets of scopes that a method must be able to access: the lexical scope in which it was defined and the object scope in which it lives. I solve this problem in the same way as javascript: by default all access is via lexical scope, except with an explicit self reference. For example, consider the following code.

```
f <- function() {
  cat("Lexical: ", a, "\n")
  cat("Object:  ", self$a, "\n")
}

O <- Object$clone()
O$m <- f()
O$a <- 10
```

```
a <- 5

O$f()
```

This illustrates that normal R lexical scoping rules apply, unless explicitly circumvented with `self`. The regular scoping rules are necessary for most function calls - otherwise `cat()` itself would not work.

Programmatically, combining lexical-scoping and object-scoping is a hack, where a new environment containing just the self object is inserted between the function and its parent environment, as shown in Figure 6. This effectively makes self a special value like `T` or `F`: while you can override it with your own value, it is not a good idea.

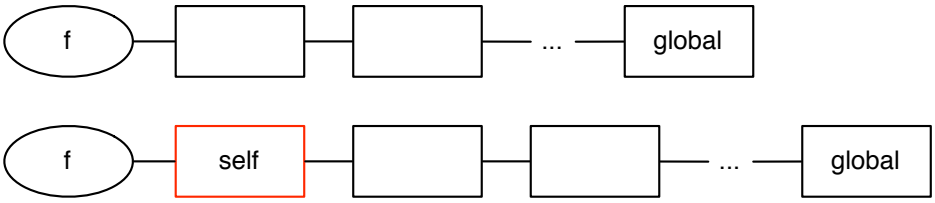

Figure 6: An illustration of how the self environment is interleaved into the stack of parent environments.

The `proto` package (Kates and Petzoldt, 2007) takes an alternative approach to solve this problem, requiring an additional explicit reference to self in the method argument list, in a similar way to python. However, proto does not separate lexical and object scope so that fields defined in object scope shadow objects defined in lexical scope..

4 Case studies

The following case studies illustrate the use of mutable objects in three problems for which they are particularly well suited. Section 4.1 shows how to create a generic memoisation function. Section 4.2 shows how mutable objects are useful in stateful simulations, in particular how they might be useful when comparing card counting techniques for blackjack. It also demonstrates how the separation between lexical and object scope can be used to create "private" variables and methods. Section 4.3 pairs a mutable representation with an object that is truly mutable object: a graphical user interface (gui).

4.1 Memoisation

Memoisation is a useful technique for avoiding excess computation. Figure 7 implements a simple cache and wrapper that implements memoisation. I have recently used this technique to good effect in an experimental version of the roxygen package: by memoising two functions I achieved a 10-fold speed up.

4.2 Card counting in blackjack

Mutable objects are useful in simulations of stateful systems. For example, to compare card counting strategies in blackjack, it is necessary to use a stateful object to model the shoe of six decks of cards from which the dealer draws cards. Figure 8 shows a minimal implementation of a shoe, with methods to shuffle the deck and draw one or two cards.

This code also illustrates an useful programming technique. We can take advantage of R's lexical scoping rules to create private variables that can not be accessed from outside of the class, `shoe`, `pos` and `draw_n`. This means that there is no way for a nefarious card counting algorithm to manipulate the shoe and show

```
library(digest)

Cache <- Object$clone()$do({
  init <- function() {
    self$hash <- new.env(hash = TRUE, parent = emptyenv())
  }

  self$get <- function(key) {
    get(key, env = self$hash)
  }

  self$set <- function(key, value) {
    assign(key, value, env = self$hash)
  }

  self$has_key <- function(key) {
    exists(key, env = self$hash)
  }
})

memoise <- function(f) {
  cache <- Cache$clone()

  function(...) {
    key <- digest(list(...))
    if (cache$has_key(key)) {
      cache$get(key)
    } else {
      value <- f(...)
      cache$set(key, value)
      value
    }
  }
}
```

Figure 7: Basic memoisation, using the `digest` package to generate a string key for any list of arguments. The `do` function is a convenient shortcut for defining multiple methods at once.

```
library(mutatr)

Shoe <- Object$clone()$do({
  shoe <- NA
  pos <- 0

  draw_n <- function(n) {
    cards <- shoe[pos + seq_len(n)]
    pos <<- pos + n
    cards
  }

  self$shuffle <- function() {
    shoe <<- shuffle_decks(6)
    pos <<- 0
  }

  self$draw_one <- function() draw_n(1)
  self$draw_two <- function() draw_n(1)
})
```

Figure 8: Modelling a deck of cards with a mutable object.

artificially favourable results. This is somewhat of an academic concern in this case, but the ability to hide internal details is useful for making stable interfaces.

4.3 Graphical user interfaces

Mutable objects are particularly well suited to problems where there really is one true underlying object, for example, GUIs and interactive graphics. Figure 9 shows the creation of a simple gui using mutatrGui, an experimental package that provides a mutable object interface to gwidgets (Verzani, 2007). There are couple of particularly nice things about this approach: the hierarchy of object names and indenting matches the gui hierarchy, and we don't have to worry about polluting the global namespace, so it is easier to give gui components descriptive names.

Smith (1995) point out that prototype inheritance is a natural fit to dialog boxes. All dialog boxes in application have something in common, but they are largely unique. To model these dialogs in a class-based environment requires creating a class for each dialog - rather a heavy requirement when most classes will have only a single instance. Self and XLISP-STAT (Tierney, 1990) also provided GUIs within a prototype-based environment.

While it is possible to design GUIs and interactive graphics in a purely functional setting (Elliott and Hudak, 1997; Courtney and Elliott, 2001), it requires a rather different way of thinking about the world, and rather esoteric mathematical constructs (monads and monoids). Web pages are also amenable functional implementation as they are typically stateless, depending only on the URL and query parameters.

5 Conclusions

This paper has discussed three important concerns when designing an OO system: mutable vs. immutable, generic functions vs. message passing, and classes vs. prototypes. Building on these ideas I introduced a

```
w <- Window$clone()$do({
  self$title <- "Window with buttons"
  self$value <- NA

  self$buttons <- Group$clone("v")$do({
    self$add(Label$clone("These are some buttons"))

    self$one <- Button$clone("One")
    self$add_space(10)
    self$two <- Button$clone("Two")

    self$add_spring()
  })

  self$status <- StatusBar$clone()
})

w$buttons$one$add_handler(function(...) w$value <- 1)
w$buttons$two$add_handler(function(...) w$value <- 2)
```

Figure 9: A simple gui, made with mutable objects.

new OO package for R, that has mutable state and prototype-based inheritance, and discussed some of the implementation issues. Three case studies showed how these mutable objects are useful for memoisation, stateful simulations and for creating graphical user interfaces.

Future plans for mutatr centre around the need to embed it more easily in the R ecosystem. Chiefly, this consists of coming up with a toolset that makes it easy to document an object's methods. I am looking into adapting the roxygen package (Danenberg and Eugster, 2008) to meet this need. Another important is performance. How much do the unusual function call semantics reduce performance and how can this be mitigated?

References

O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The SELF 4.1 Programmer's Reference Manual*. Sun Microsystems, Inc, 2000.

D. Bates and M. Maechler. *Matrix: Sparse and Dense Matrix Classes and Methods*, 2009. R package version 0.999375-29.

H. Bengtsson. The R.oo package - object-oriented programming with references using standard R code. In K. Hornik, F. Leisch, and A. Zeileis, editors, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, Vienna, Austria, March 2003.

C. Chambers. Object-oriented multi-methods in cecil. In *ECOOP'92 Conference Proceedings*, 1992.

J. M. Chambers and D. Temple Lang. Object-oriented programming in r. *R-news*, 1(3):17–19, 2001.

A. Courtney and C. Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.

P. Danenberg and M. Eugster. *roxygen: Literate Programming in R*, 2008. URL `http://roxygen.org`. R package version 0.1.

S. Dekorte. Io, a small programming language. In *OOPSLA05*, 2005.

L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In *ECOOP' 87 European Conference on Object-Oriented Programming*. Springer, 1987. URL `http://www.dreamsongs.com/NewFiles/ECOOP.pdf`.

J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM. ISBN 0-89791-193-8.

C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan programming: an object-oriented and dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. ISBN 0-201-47976-1.

R. Gentleman and R. Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9:491–508, 2000.

R. Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM New York, NY, USA, 2008.

D. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979.

R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

H. Kaplan. Persistent data structures. In *Handbook on data structures and applications*. CRC Press, 1995.

L. Kates and T. Petzoldt. *proto: Prototype object-based programming*, 2007. R package version 0.3-8.

C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

B. Rice and L. Salzman. *The Slate Programmer's Reference Manual*, 2004.

L. Salzman and J. Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In *ECOOP*, 2005.

R. B. Smith and D. Ungar. Programming as an experience: The inspiration for self. In *ECOOP '95*, 1995.

W. R. Smith. Using a prototype-based language for user interface: The newton project's experience. In *OOPSLA*, 1995.

L. Tierney. *Lisp-Stat: an object-oriented environment for statistical computing and dynamic graphics*. John Wiley & Sons, New York; Chichester, 1990.

D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and symbolic computation*, 1991.

D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing programs without classes. *Lisp and symbolic computation*, 1991.

P. Van Roy. Programming paradigms for dummies: What every programmer should know. In G. Assayag and A. Gerzso, editors, *New Computational Paradigms for Computer Music*, 2009. URL `http://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf`.

P. van Roy and S. Haridi. *Concepts, Techniques and Models of Computer Programming*. The MIT Press, 2004.

J. Verzani. gwidgets: a toolkit-independent api for building guis in r. In *useR! 2007*, 2007. URL `http://wiener.math.csi.cuny.edu/pmg/gWidgets/useR2007_presentation.pdf`.