

Some Performance Improvements for the R Engine

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

June 26, 2014





- There are a number of efforts underway to improve performance issues in R.
- This talk will focus on
 - reducing duplication
 - switching from **NAMED** to reference counting
 - duplication in complex assignment
- A few other directions will be mentioned if time permits.



Introduction

- Duplicating values takes time and uses memory.
- Most duplication in R occurs in the context of complex assignment/replacement operations like

```
> x[[i]] <- y
```
- Duplication is needed for two reasons:
 - to preserve the pass by value semantics
 - to prevent creating cycles (except through environments)
- Michael Lawrence contributed changes to reduce duplication by using shallow copying in nested structures when possible.
- This also involves using a check for when an assignment would create a cycle.
- Shallow copying increases sharing of structure; this sharing is *not* preserved when serializing.
- These changes were incorporated in R 3.1.0.
- At the time I had started to think about using reference counting to determine when duplication might be needed, so the changes made kept this in mind.



Reference Counting

The NAMED Mechanism

- In complex assignments/replacements like

```
> f(x, i) <- y
```

```
> f(g(x, j), i) <- y
```

the modification can be made without duplicating if the LHS values are only accessible through one R level variable.

- The **NAMED** mechanism counts the number of variables from which an object is reachable.
- The **NAMED** value is maintained in a lazy fashion — it is updated on extraction.
- Currently only the values 0, 1, 2 are allowed.
 - the value “2” means “2 or more.”



Reference Counting

Some Issues

- The implementation is hard to understand and maintain
 - implementation is distributed in many places
 - omissions of **NAMED** management code are hard to detect
- Decrementing **NAMED** values is difficult
 - not useful with a maximal value of 2
 - difficult to do automatically
- Proper reference counting seems like an alternative worth investigating.



Reference Counting

Basic Implementation

- Basic implementation is straight forward:
 - when a new value is assigned to an **SEXP** field the new values's count is incremented and the old value's count is decremented.
 - Count management happens in constructors and in updating functions.
 - These are already well isolated in **memory.c** because of the write barrier.
- Using the existing 2-bit **NAMED** field allows a maximal reference count of 3.



Reference Counting

Notes and Comments

- Complex assignment/replacement needs to track reference counts for all intermediate LHS values.
- Some fields should not increment reference counts:
 - `.Last.value` variable
 - promises used internally for LHS values
 - other internal lists, e.g. for arguments to `BULTIN` calls
- For now, this is addressed with a “do not track” bit.
- Non-tracking objects are created with `CONS_NR`, `R_mkEVPROMISE_NR`
- Explicit incrementing/decrementing can be useful in places.



Reference Counting

Notes and Comments

- This mechanism seems much easier to maintain:
 - almost everything is done right by default
 - all non-standard uses are easy to detect and review
 - omitting an exception results in more duplicating but still correct semantics
- This is available in the current R-devel sources.
 - Defining `SWITCH_TO_REFCNT` uses reference counting with the existing memory layout and maximal reference count of 3.
 - Switching to a larger maximal count is also possible but needs a small code fix.



Reference Counting

Further Developments

- All objects are reference counted, including environments.
- In closure calls, environments are almost always used in a stack-like fashion:
 - once a call returns the environment is no longer reachable
 - the values of the environments variables can have their reference counts decremented
- An example:

```
> x <- rnorm(1e6)
> m <- mean(x)
> x[1] <- 0
```
- With **NAMED** or simple reference counting the final line has to duplicate because the mean closure created a reference to `x`.
- With a (not yet checked in) modification that releases environment bindings at the end of closure calls, if possible, this does not duplicate.
- No change to the implementation of `mean` is needed.



Complex Assignment/Replacement

The Simple Case

- A frustrating example:

```
> d <- data.frame(x = rnorm(1e6))  
> for (i in seq_len(nrow(d))) d[[i, 1]] <- d[[i, 1]] + 1
```

- This duplicates `d` on every iteration.
- The `[[<-data.frame` function is implemented by a closure.
- When that closure is called, there are two variables that reference the value of `x`:
 - the top level variable `x`
 - the first parameter in the closure
- Packages can only define closures, not primitives.
- So all replacement functions defined in packages will require duplicating the LHS.
- Unless they cheat with C code, which could be dangerous.



Complex Assignment/Replacement

A Possible Approach

- We can address this by
 - keeping track of the number of references that are part of the replacement process
 - identifying when a closure call is in a replacement context
 - allowing low level primitives to modify without duplicating when this information allows.
- A mechanism to do this has been implemented.
- Some further testing and cleaning is needed before committing.
(Hopefully in the next month or so.)



Complex Assignment/Replacement

A Possible Approach

- With this enabled, replacement functions have to be careful not to signal errors after partial modifications.
- Many existing replacement functions are not careful about this, so turning this on by default is not possible.
- For now:
 - The internals keep track of whether direct modification is possible in principle.
 - The closure has to take some action to authorize direct modification.
 - Currently this means calling `.Internal(modifying(x))` — something better is needed.
- It would also be a good idea to disable user interrupts during these closure calls.



Complex Assignment/Replacement

A Simple Example

```
bar <- function(x) x[[1]]

'bar<- ' <- function(x, value) {
  .Internal(modifying(x))
  x[[1]] <- value
  x
}

x <- list(1)
bar(x) <- 2
```



Complex Assignment/Replacement

Nested Complex Assignment/Replacement Expressions

- A nested complex assignment/replacement:

```
> f(g(x, j), i) <- y
```

- If `f<-` and `g<-` are both primitives and both LHS values have only one reference, then they can be destructively modified *if all possible values returned by f<- would be OK as RHS values for g<-*.
- One problem case (the only one I believe):

```
> m <- matrix(0, 2, 2)
```

```
> dim(m)[2] <- 3L
```

```
Error in dim(m)[2] <- 3L :
```

```
dims [product 6] do not match the length of object [4]
```

- To deal with this the `dim` attribute is marked as immutable (i.e. always duplicated on modify).



Complex Assignment/Replacement

Nested Complex Assignment/Replacement Expressions

- If $f \leftarrow$ is a closure and $g \leftarrow$ is a primitive then the approach outlined previously should still work (e.g. a list of data frames is OK).
- If $g \leftarrow$ is a closure it could
 - reject the value produced by $f \leftarrow$
 - want to look at the unmodified original LHS value
 - do any number of wild and strange things
- There does not seem to be any way around this other than to (shallow) duplicate the inner LHS whenever the outer replacement function is a closure.
- This is done when a closure is used to extract an inner LHS in the complex assignment process.
- This is based on the heuristic that the replacement function will only be a closure if the extraction function is also.



Complex Assignment/Replacement

Nested Complex Assignment/Replacement Expressions

- One possibility that might handle closures in more cases would be to defer actual modifications until the very end when all primitive modifications are applied.
- This would be quite challenging to implement but might be possible.
- This would probably require considerable rewriting of replacement functions, which may be hard to get programmers to buy into.
- It is probably worth some more investigation.



- Byte code:
 - Add a typed stack to avoid boxing/unboxing of scalar results in byte compiled code.
 - Add instructions for handling vector/matrix indexing efficiently in byte compiled code.
 - Look into strictness analysis/declarations and inlining.
- Interpreter:
 - Explore releasing memory when reference count drops to zero.
 - Avoid allocating argument lists in **BUILTIN** calls, among others.
 - More efficient closure calling, handling of promises, etc.
- Larger data sets:
 - More efficient representation of arithmetic sequences, default row names, etc.
 - Avoid generating default row names, residuals, fitted values, full Q of QR factorization in `lm.fit` and others.
 - Parallel vector operations (`pnmath`), hopefully via OpenMP.
 - Consider full support for 64 bit integers.