Can interpreting be as fast as byte compiling? + Other developments in pqR

Radford M. Neal, University of Toronto

Dept. of Statistical Sciences and Dept. of Computer Science

`http://www.cs.utoronto.ca/∼radford`

`http://radfordneal.wordpress.com`

`http://pqR-project.org`

R Summit Conference, Copenhagen, June 2015

Why interpret rather than byte compile in pqR?

Some pqR optimizations are only implemented in interpreted code, mainly those using the “variant result” mechanism. Some examples:

- Automatic parallelization with helper threads — eg, `exp(colSums(M))`.
- Merging of operations — eg, `u <- 2*vec^2+1`.
- Short-cut evaluations – eg, fast `any(x>0)` if early element is positive.
- Avoidance of sequence creation — eg, `for (i in 1:n)` or `vec[i:j]`.

These optimizations tend to help more with vector and matrix operations, but not as much with operations on scalars. This seems a natural focus for R.

The run-time context available to the interpreter helps when implementing these optimizations. Could they be done with the byte-code compiler too? Maybe, but here I’ll consider the other option of getting rid of the compiler.

Why compilied code might be faster than interpreted code

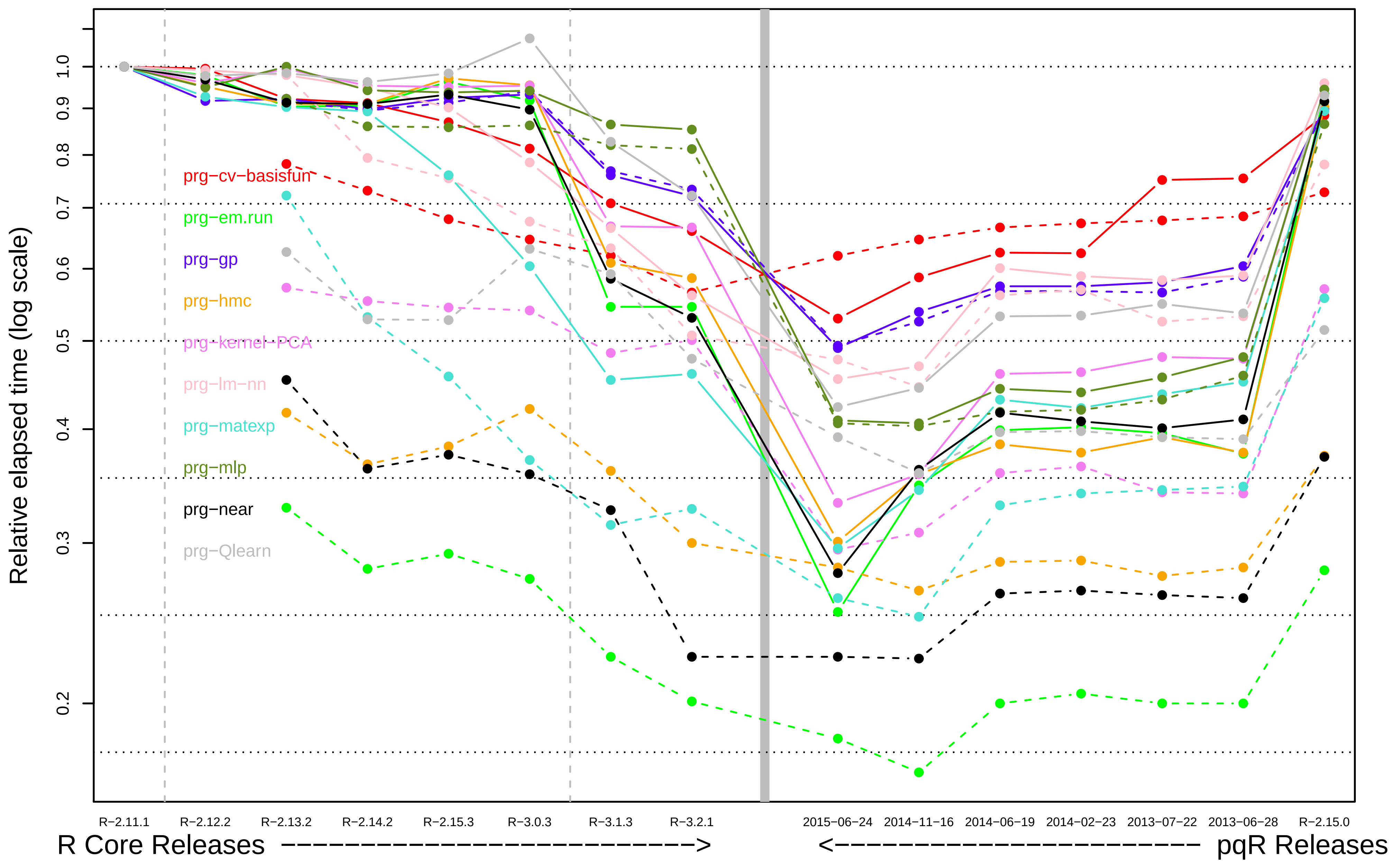
Of course, the point of byte compiling is to speed up execution.

Why might interpreting byte compiled code be faster than interpreting the abstract syntax tree of the program?

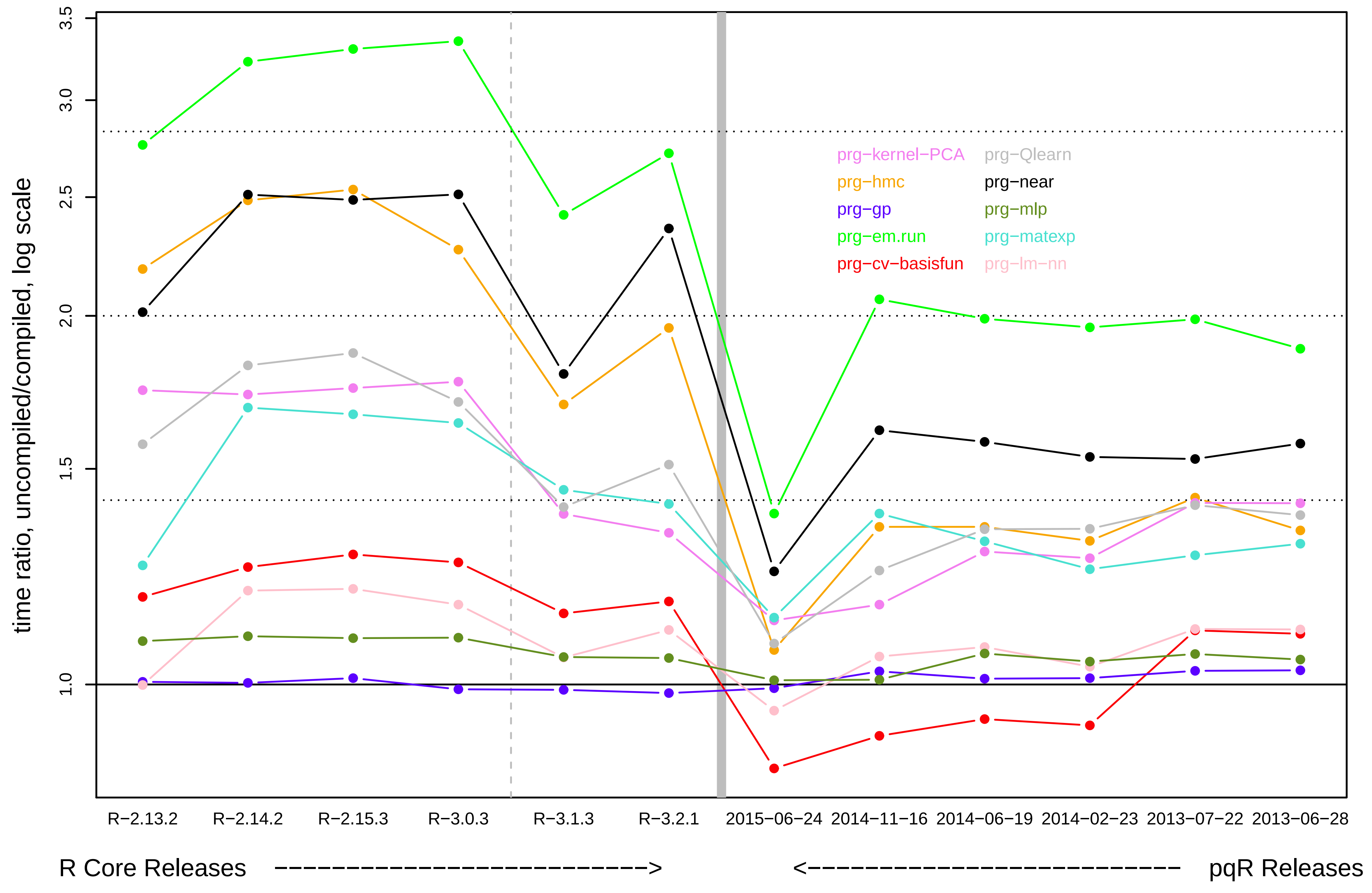
- Less general overhead for interpreting — fewer conditional branches, less fetching from memory (if code is more compact).
- Some operations done at compile time — eg, constant folding.
- Fast invocation of primitive operations — eg, `+` and `<-` — if they are assumed to not be redefined.
- Fast lookup of local variables.

In R Core implementations after R-2.13.0 and before R-3.1.0, the byte code compiler was also the focus of work on speed improvement. Several speed improvements applied only to byte compiled code, but could have been put in the interpreter as well (notably changes to subset assignment).

Speed of R Core & pqR versions with & without compiling



Changes in the relative advantage of byte compiling



Fast interface to primitive functions

R Core implementations pass arguments to primitive functions as a pairlist, allocating a CONS cell for every argument. This is a major contributor to slowness in the interpreter.

In pqR, there is a "fast" interface for primitive functions of one argument, which passes the argument directly to the C routine for the primitive. (Though not all unary primitives use this interface yet.)

Earlier versions of pqR also had a fast interface for binary operators, but now these operators are just passed the unevaluated arguments, and evaluate them themselves. This simplifies the code in `eval`, and allows these operators to ask (conditionally) for variant results when evaluating arguments.

Fast lookup of base functions and operators

It's crucial that base functions like `if`, `<-`, `+`, and `length` be found quickly.

Redefining these operators is allowed in R, though not supported by byte-compiled code. Current pqR optimizations still permit redefinition.

There are three distinct contexts to consider, differing in the environments that have to be searched to find a base function:

- Base functions referencing other base functions:

  local frames... → base

- Functions in loaded packages:

  local frames... → package namespace → package imports → base

- User functions read in with `source`:

  local frames... → global environment → loaded packages... → base

The global cache

To make searches for base functions from user functions tolerably fast, R Core implementations have for some time used a “global cache” — a hash table that holds bindings from the global environment, all loaded packages, and the base environment.

In pqR, this is extended by flagging cache entries that refer to the base environment, allowing their values to be fetched without looking in the hash table.

This helps with user functions, but not with package or base functions.

This pqR optimization was put into R-3.1.0.

Quickly skipping local frames without special symbols

In pqR, a lookup for a symbol in a set of "special" symbols can quickly skip local frames that don't contain any such symbols, as indicated by a flag in the environment.

Cost: Maintaining the flag for environments (but don't bother resetting if a special symbol is removed); overhead of checking the flag when the symbol isn't actually special.

Current special symbols:

```
"if", "while", "repeat", "for", "break", "next", "return",
"function", ".C", ".Fortran", ".Call", ".External", ".Internal",
"(", "{", "+", "-", "*", "/", "^", "%%", "%/%", "%*%", ":",
"==", "!=", "<", ">", "<=", ">=", "&", "|", "&&", "||", "!",
"<-", "<<-", "=", "$", "[", "[[", "$<-", "[<-", "[[<-"
```

Adding more would increase the chance that a local frame has one.

This pqR optimization was also put into R-3.1.0.

Another way of quickly skipping a local frame

In pqR, when looking up a function, F, a local frame is also quickly skipped if the symbol F has that frame recorded as the last local (unhashed) frame in which F was not found.

This record is kept in a new field in a symbol node, which is *cleared*, not followed, on every garbage collection.

It's also, of course, cleared when the symbol is defined in that frame.

This optimization helps for all lookups of base functions that are not special symbols. It also helps with other non-local function lookups.

But it only allows at most one local frame to be skipped.

Fast lookup of local variables

Most function lookups skip local frames. But most variable lookups are found in a local frame, usually the first one.

In pqR, this is often sped up using two more new fields in a symbol node, which record an environment in which the symbol has been found, and the node with the symbol's binding in that environment.

These fields are also *cleared*, not followed, on every garbage collection. They're also updated appropriately when symbols are defined or removed.

This change speeds up most repeated local variable accesses.

Inlining of `eval` and in `eval`

Once function and variable lookups have been improved, it becomes useful to inline at least part of the lookup code into the `eval` function.

It also becomes useful to partially inline some calls to `eval` itself.

In both cases, only the fast cases are inlined, with the inlined code calling an external function for the general case. This limits code expansion (which affects speed as well as memory usage).

Re-using space for intermediate values

The obvious way of evaluating `w <- x + y*z` allocates two new objects to hold `y*z` and `x + y*z`. Assuming it's not referenced elsewhere, the old value of `w` will eventually be garbage collected.

In pqR, a new object is allocated for `y*z`, but when `x + y*z` is computed, the result is stored in this same object (assuming all variables have the same type and length). This is recognized as possible becaused `NAMED` is zero for this object.

This is helpful for scalar computations, and even more helpful when `x`, `y`, and `z` are long vectors.

This pqR optimization was put into R-3.1.0.

Direct assignment when updating variables

More recent versions of pqR also avoid allocations on some updates of variables, such as `x <- x + 1`, where `x` might be either a scalar or a vector.

The `<-` operator peeks at the RHS expression, and sees that its first argument matches the symbol to be assigned to. It then evaluates the RHS with a request for a variant result in which the value is assigned to the first argument directly (here by `+` rather than by `<-`).

As well as avoiding allocation of a new object, this also allows for more task merging — for example, in

```
    x <- x + 1; x <- 2*x
```

the add and multiply will be done in one loop over elements of `x`.

Eliminating allocations by using static boxes

Consider evaluating the following, when `i` is scalar and `v` is a real vector:

```
    u <- 2*v[i+1] + v[i-1]^2
```

This produces six intermediate values, though the previously-mentioned method of re-using space will reduce the number of allocations done to four.

But in the latest version of pqR, there is no storage allocation at all when evaluating this expression, if `u` already contains an (unshared) scalar real value. (Confirming this is easily done using pqR's extension of `Rprofmem`.)

Instead, all intermediate results are stored in "static boxes".

Each static box has all the structure of an R object (eg, a length), fixed at what is needed for a scalar with no attributes. Only the data changes.

There are currently static boxes for scalar integers and for scalar reals. (Scalar logicals are handled by three static constants.)

How static boxes are used

Internally to the interpreter, static boxes may be used for a value from `eval` if the caller of `eval` said it was prepared to handled a variant return of a static box.

The primitives handling arithmetic and vector subscripting will potentially return static boxes.

The primitives handling arithmetic and comparisons say they are prepared to handle an operand in a static box. If the first operand comes in a static box, its value is saved in a local variable before the second operand is evaluated, since the static box might be used within the second evaluation.

For simple subscripting, `[` will ask for the index to come in a static box.

For simple assignments, `<-` will ask for the RHS to come in a static box. If the variable assigned to is unshared and has matching type, length, etc., the assignment is done by just copying the value from the static box.

Direct returns from functions

In the R Core interpreter, `return(v)` does a non-local jump, after searching the stack of contexts for a return destination. This is much slower than returning a value by reaching the end of the function.

In pqR, `return` is usually done without this overhead.

Functions evaluate their body with the variant of asking for a direct return value, without the non-local jump being needed.

This variant request gets passed on by `{` and `if`. The `{` primitive returns without evaluating further expressions once an expression delivers a direct return value.

Peculiar constructs such a `b <- a + if (a>0) return(2) else 1` are not handled by this mechanism. So unfortunately the return context still needs to be created.

Eliminating parentheses

In the interpreter, parentheses take time to evaluate, so `x <- (y*z)+2` is slower than `x <- y*z+2`. Byte compiled code has parentheses eliminated.

In the latest version of pqR, *necessary* parentheses — eg, in `x <- (y+z)*2` — are omitted from the pairlist representation of an expression. When an expression is deparsed, such parentheses are re-inserted, so they reappear when, for instance, a deparsed expression is used as a label in a plot.

Of course, people do sometimes use unnecessary parenthesis, for clarity. These will still slow interpreted code.

Another problem is that sometimes (eg, in the coxme package), formula expressions are examined by R code that considers parentheses to be significant for other than grouping. This could be handled by not suppressing parentheses in expressions inside formulas.

Redesigned subset assignment

In pqR, the subset assignment mechanism has been redesigned, to both speed it up, and eliminate various semantic anomalies, such as:

```
> f <- function () { cat("Hi!\n"); 1 }
> L <- list(c(4,7),"x"); L[[ f() ]] [1] <- 9
Hi!
Hi!
```

There isn't time to talk about this here, but I have a blog post about it at `radfordneal.wordpress.com`.

The byte compiler also uses a different mechanism, which gives it a speed advantage. This advantage has increased in the latest R Core versions.

The redesign in pqR provides the basis for faster subset assignment, and does speed it up significantly. But further optimizations could be done, such as handling the special case of scalar indexes more quickly. (They would have been mostly pointless before, when general overhead was larger.)

Possibilities for further reduction in interpretive overhead

- Extend direct assignment to updates like `x <- 2*x+1`.
- Somehow reduce code in `eval` devoted to special cases like `...`, `..1`, etc. Maybe `..` should be an operator?
- Reduce overhead in invoking functions that just call `.Internal`.
- Speed up lookup of base functions from functions in packages — currently first does two hash table lookups before getting to base.
- Speed up hash tables generally.
- Maintain a flag bit used to quickly check whether an object has names.
- Reduce the size of the CONS nodes used to represent the abstract syntax tree that the interpreter sees. This requires a redesign of the garbage collector…

Redesigning memory layout and garbage collection

We need to eliminate the two “next” and “prev” pointers in every object that are used only by the garbage collector. In 64-bit architectures, these two pointers occupy 16 bytes, out of 56 bytes total for a CONS node.

Idea: Replace these 16 bytes by 4 bytes that give an index to a table of objects and an index of this object within that table.

Due to alignment considerations, this likely reduces a CONS node from 56 bytes to 40 bytes. Other objects shrink too.

Total memory usage is slightly less, accounting for space for the gc tables. More importantly, such a scheme reduces the amount of memory referenced outside of garbage collection (important for cache performance).

Also, a full garbage collection will no longer write to every object, causing memory page duplication after a fork (eg, in `mclappy`).

The scheme also removes 5 gc bits from objects, freeing them for other uses.

Some other plans for pqR

- More operations done as tasks that can be parallelized or merged with other operations — eg, extracting subsets, as in `vec[100:200]`.
- Allow list elements to have their evaluation deferred, increasing potential use of task merging and parallel evaluation.
- Better integration of task merging with parallelization, allowing merging with operations that have already started to execute in a helper thread.
- Exactly-rounded sums and means. Exactly-rounded computation of means should be faster than the present method, and exactly-rounded sums only slightly slower. I have a blog post and an arxiv paper on this.
- Compact representations of objects. Represent logicals in one byte or in two bits, represent diagonal matrices by just the diagonal, etc. Do it automatically and invisibly. I have a blog post on this too, and an arxiv paper on compact floating-point representations.