Advances in Memory Management and Symbol Lookup 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

RIOT2017, Brussels

Some Recent Performance Improvements in pqR

- A new garbage collector, using my Segmented Generational Garbage Collector (SGGC) facility. SGGC and code using it in pqR could be used in R Core and other R implementations without great effort. SGGC could also be used for non-R projects.

- More compact memory layouts, made possible by the new GC, including the option of using "compressed" 32-bit pointers even on 64-bit platforms. This improvement is also not tightly tied to pqR.

- Improvements to symbol lookup, including a way to quickly skip environments that don't contain the symbol being looked for.

- A new parser, using recursive descent (no bison). Easier to modify. Faster. Avoids the old parser's quadratic time usage in some contexts. Should be easy to adapt for use in R Core and other implementations.

- Fast extension/contraction of vectors, mirroring a recent R Core improvement, but more general. Facilitated by use of SGGC.

Generational Garbage Collection

Both the old and the new garbage collectors use the "mark and sweep" method, improved by keeping track of objects in different "generations".

In a "full" collection, all accessible objects are found, and marked. Then those not marked can be collected. Looking at every object is slow.

Objects are considered to be in generation 0 if they have been allocated since the last GC. Objects in generation 1 have survived one GC. Those in generation 2 have survived more than one GC.

A level 0 collection looks only at generation 0, not trying to collect older generations. It's much faster than a full collection if most objects are not newly allocated. Similarly, a level 1 collection looks only at generations 0 and 1.

Special provisions are needed to handle the situation where an object of an older generation is modified to point to an object of a younger generation.

Problems with the Old GC and Memory Layouts

Some sources of inefficiency in the old garbage collector:

- Sets of objects — eg, those in generation 1, or those that have old-to-new references — are represented using doubly-linked lists, requiring two pointers in every object header.
- Objects are marked by setting a bit in their header (which is later cleared).
- Strings are kept in a hash table, so that identical strings will be shared, which is scanned in its entirety even for level 0 collections.

This all has rather bad memory cache behaviour.

The Segmented Generational Garbage Collector

I've written a general-purpose Segmented Generational Garbage Collector (SGGC), which is now used in pqR-2017-06-09 (which differs from pqR-2016-10-24 only in this respect).

SGGC and the code using it in pqR could be adapted for use in other implementations of R. SGGC could also be used for projects unconnected to R. (An interpeter for a toy Lisp-like language is included as a test program.)

Like the old R garbage collector, SGGC uses a mark-and-sweep method with two old generations, but it represents the required sets of objects (including those marked) using chains of bit vectors, not doubly-linked lists. This reduces memory usage, and improves cache performance by localizing references.

SGGC can be found at `https://gitlab.com/radfordneal/sggc`

Compressed Pointers — Segment Index + Offset

Internally, SGGC refers to objects using compressed 32-bit pointers — consisting of a segment index (26 bits) and an offset within a segment (6 bits). All objects in one segment have the same “SGGC type” — not their R type, but enough to determine where pointers are located.

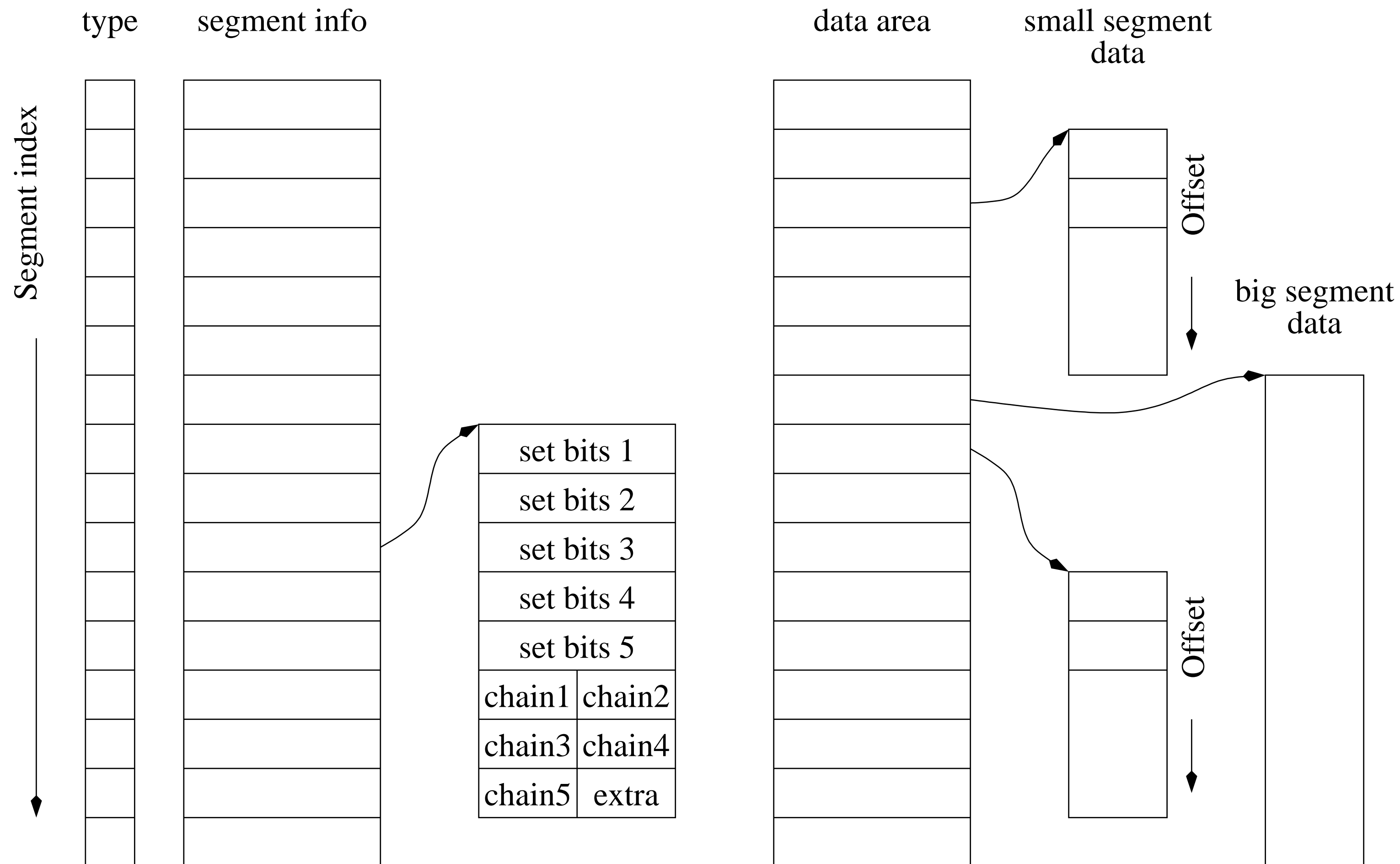
In pqR, the SEXP type used in the C API can be either an ordinary uncompressed pointer (as before), or a compressed pointer. On 64-bit platforms, using compressed pointers reduces memory usage significantly for language objects, small numeric vectors, and string vectors and lists of any length. On 32-bit platforms, compressed pointers produce smaller savings.

Compatibility:

- Using compressed pointers on 64-bit platforms currently makes RStudio crash when debugging a program — probably fixable.
- Rcpp doesn’t currently work with compressed pointers — may also be fixable.

Data Structures for Segments and Objects

The segment index part of a compressed pointer is used to index several tables — giving the SGGC type, a pointer to data, and a pointer to segment information (including five chains of bit vectors for sets).



"Small" segments contain several objects, allocated in 16-byte chunks, found from the object's "offset". "Big" segments hold only one object, of any size.

Compressed – Uncompressed Pointer Translation

If a SEXP is a compressed pointer, finding the data area for an object is done as follows:

1) Use the index part of the pointer to fetch the pointer to the data area for objects in its segment.

2) To this pointer, add 16 times the offset from the compressed pointer. (A compressed pointer to an object in a big segment will have offset 0.)

If a SEXP is an uncompressed pointer, the data area for the object it points to must store the corresponding compressed pointer in its header, so that the uncompressed pointer can be translated to a compressed pointer for use by SGGC.

(This is why there is a space saving from using compressed pointers even on 32-bit platforms.)

Performance and Memory Limits of Compressed Pointers

Speed effect of making a SEXP a compressed pointer:

- About a 30% penalty for code dominated by interpretive overhead, looking at language objects, argument lists, and symbol bindings.
- Negligible difference for code dominated by computations on large numerical vectors or matrices.
- A factor of two or more speed improvement for code dominated by storage allocation and copying of large numbers of pointers.

Memory limitations using SGGC (even if a SEXP is uncompressed):

- Can make use of up to about 64 GBytes if memory is used for billions of small objects (less than about a thousand bytes each).
- Can make use of effectively unlimited amounts of memory if it is mostly used for large objects (several thousand bytes or more).

Sizes of Objects in Old and New Memory Layouts

	R-3.4.0 and pqR-2016-10-24	Regular ptrs pqR-2017-06-09	Compressed ptrs pqR-2017-06-09
CONS cell	56	51	21
Scalar real	48	34	25
Long vector of small unique strings, per string	48+8+8 = 64	34 + 8 + 8 = 50	25 + 4 + 4 = 33

For 64-bit platforms. Assumes string hash table load factor of 1.
pqR-2017-06-09 sizes include overhead for segment information.

For a long vector of small (< 8 character) unique strings, pqR compressed pointers give essentially a factor of two space savings over the old memory layout. Compared to uncompressed pointers with the new layout, the savings is a factor of about 1.5.

Speed Tests of R-3.4.0 and pqR with Old and New GC

Times for a small test program (dominated by scalar arithmetic & function calls, two versions), then for 1000 level 2 garbage collections, followed by memory stats. Processor is a Xeon E3-1270 v5 (Skylake) @ 3.6GHz.

R-3.4.0, no byte compilation (and no JIT):

```
Test Program Times:  0.623  0.705
> system.time(for (i in 1:1000) gc())
   user  system elapsed
  7.144   0.000   7.155
> gc(T)
Garbage collection 1142 = 131+5+1006 (level 2) ...
10.8 Mbytes of cons cells used (44%)
2.4 Mbytes of vectors used (39%)
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 200976 10.8     460000 24.6   460000 24.6
Vcells 309805  2.4     786432  6.0   735132  5.7
```

Speed Tests (2)

R-3.4.0, with byte compilation (explicit, not JIT):

```
Test Program Times:  0.208  0.257
> system.time(for (i in 1:1000) gc())
   user  system elapsed
 12.564   0.000  12.585
> gc(T)
Garbage collection 1028 = 19+3+1006 (level 2) ...
17.2 Mbytes of cons cells used (54%)
5.3 Mbytes of vectors used (52%)
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 321752 17.2     592000 31.7   592000 31.7
Vcells 685996  5.3    1316098 10.1  1102339  8.5
```

It's notable that byte compilation has increased the number of objects substantially, which produces a corresponding increase in level 2 GC time.

The test programs are almost three times faster.

Speed Tests (3)

pqR-2016-10-24, no byte compilation (old GC, improved in pqR):

```
Test Program Times:  0.284  0.334
> system.time(for (i in 1:1000) gc())
   user  system elapsed
  4.324   0.000   4.331
> gc(T)
Garbage collection 1020 = 15+0+1005 (level 2) ...
9.8 Mbytes of cons cells used (32%)
0.8 Mbytes of vectors used (10%)
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 140636  7.7     442500 23.7   442500 23.4
Vcells 100240  0.8    1000000  7.7   512705  4.0
```

Without byte compilation, this previous pqR version runs the test programs about 30% slower than R-3.4.0 with byte compilation.

Speed Tests (4)

pqR-2017-06-09, no byte compilation, new GC, regular ptrs:

```
Test Program Times:  0.281  0.331
> system.time(for (i in 1:1000) gc())
   user  system elapsed
  2.860   0.000   2.866
> gc(T)
Garbage collection 1031 = 20+7+1004 (level 2), 23.764 Megabytes, rec
        Objects Megabytes Segments
Current  140856    23.764    22358
Maximum  141105    23.764    22358
```

Only change from pqR-2016-10-24 is new garbage collector. Test program times are very close — GC is a small part of total time for this program.

The output of `gc(T)` is different, reflecting changes in the new GC (the actual numbers of objects are almost identical).

GC time is 1.5 times smaller than for the previous version, and 2.5 times smaller than R-3.4.0 without byte compilation (though some of that reflects a somewhat larger number of objects).

Speed Tests (5)

pqR-2017-06-09, no byte compilation, new GC, compressed ptrs:

```
Test Program Times:  0.371  0.434
> system.time(for (i in 1:1000) gc())
   user  system elapsed
  2.128   0.000   2.127
> gc(T)
Garbage collection 1031 = 20+7+1004 (level 2), 11.812 Megabytes, rec
        Objects Megabytes Segments
Current  140856    11.812     8876
Maximum  141105    11.812     8876
```

Using compressed pointers increases time for the test programs by about 30%. This is probably about the worst-case penalty.

Time for GC is smaller, now less than half that for pqR with the old GC.

Memory usage is about half that with uncompressed pointers, since memory is largely occupied by language objects that consist mostly of pointers.

Symbol Lookup in Previous Versions of pqR

To find a symbol binding, R looks in a sequence of environments. Some are searched linearly; some have hash tables. Even for a hashed environment, a lookup takes some time (and accesses some cache lines).

Fast lookup of operators and other common base functions is crucial.

A method introduced in pqR (later adopted in R Core versions) quickly skips some environments when looking for a "special" symbol, defined in base, but unlikely to be defined elsewhere (eg, `+`, `` `for` ``). Special symbols are identified by a bit in their header. Environments that have never contained special symbols are also identified by a header bit.

pqR also records in each symbol the binding in the last non-hashed environment where it was found — avoiding a search for this binding.

Up until now, pqR also recorded the last non-hashed environment in which the symbol was *not* found, allowing that environment to be quickly skipped.

The Symbits Idea

In the current development version of pqR (not the latest release), the idea of using one bit to identify an environment with no "special" symbols is generalized to using a larger set of "symbits" (eg, 64) that can represent the absence of many symbols.

Each symbol is assigned (when it is installed) a symbits value in which only a few (eg, 3) bits are 1s. Environments record the logical-OR of the symbit values for all symbols they contain (or once contained).

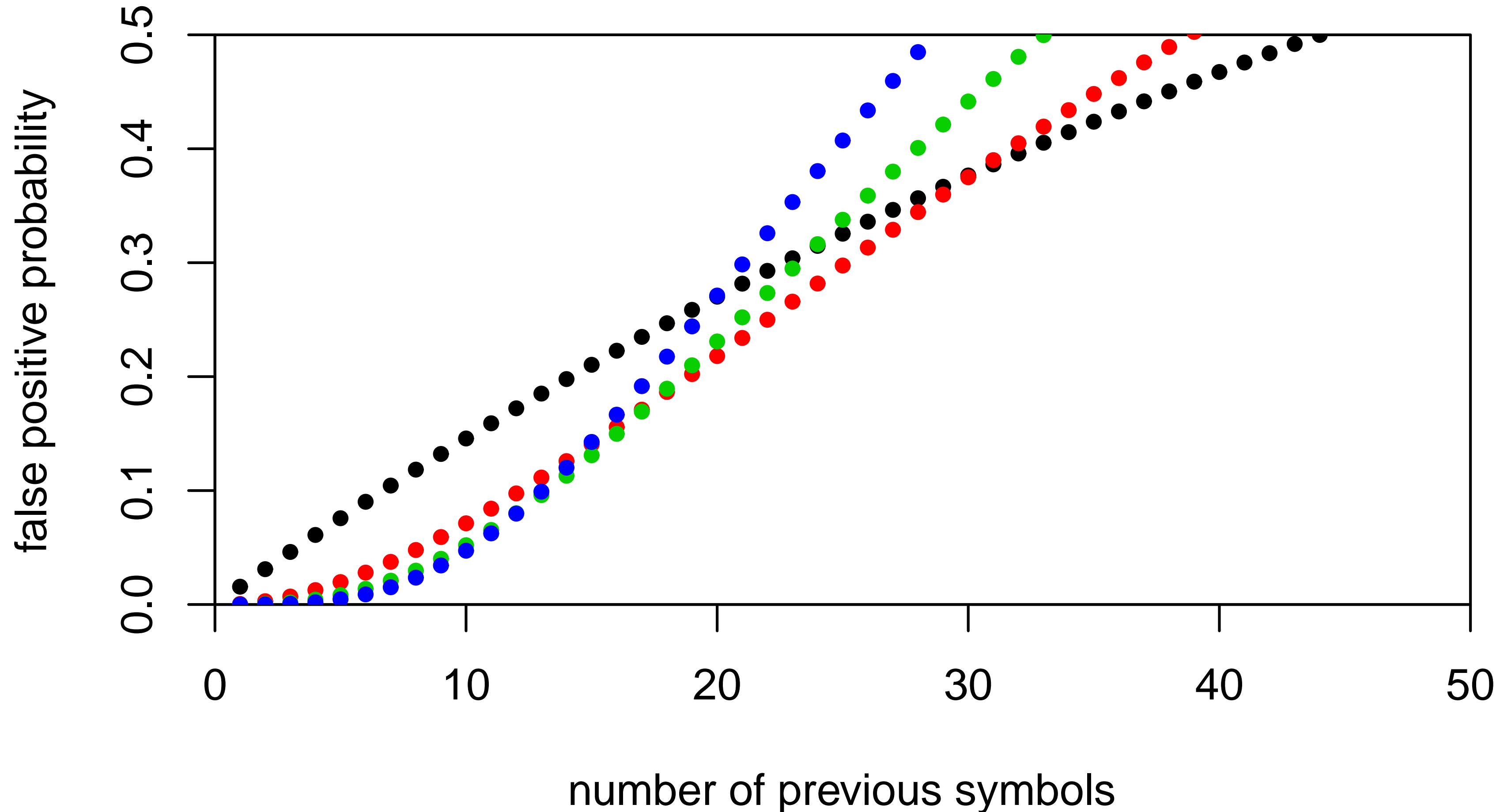
If `S` is the symbits for a symbol being searched for, and `E` the symbits for an environment, we know that the symbol is not in that environment if the logical-AND of `S` and `E` does not equal `S`.

The hope is that this can replace both the "special" symbol trick, and the "last environment not found in" trick. There is flexibility to fiddle the idea a bit to make it work especially well for "special" symbols.

Note: After the talk, Lukas Stadler pointed out to me that this idea is known as a "Bloom filter" — eg, see `https://blogs.oracle.com/ali/gnu-hash-elf-sections`

How Number of Bits Set Affects Error Probability

Plot of the probability of mistakenly thinking a symbol might be in an environment, as number of symbols it contains various, when symbols have 1 to 4 symbits set in a 64-bit word. Assumes perfect random bit assignment.



How Well do Symbits Work?

I've only just started playing with it. Preliminary impressions:

- Performs as expected, with quite low overhead.
- Allows special symbols to be found quickly even if a few specials have been redefined locally — hence worthwhile for large hashed environments (eg, package imports), and appropriate for a larger set of special symbols (eg, include `length`).
- May not completely replace the "last environment not found in" approach, because some local environments have too many symbols.

  But no reason can't have both (at cost of slightly greater overhead). Using symbits should make "last environment not found in" work better, by eliminating some enviornments from consideration.

Is Byte Compilation Desirable?

Advantages of byte compiling:

- Stepping through byte-code instructions may be faster than traversing an abstract syntax tree (AST).
- With mild restrictions, some operations may be doable at compile time — eg, constant folding, resolving which environment a symbol is in.

Disadvantages of byte compiling:

- Doubles the maintainence burden — all language operations are implemented twice (assuming an interpreter is still needed).
- Doesn't work well when code is dynamically created, environment enclosures changed, etc.
- It may be hard for compiled code to adapt to the context of an operation.

More on Byte Compilation...

Here's an example where context is available to the interpreter, using pqR's "variant result" mechanism:

```
f <- function (x) x>0
v <- seq(0,1,length=1000000)
print(any(f(v)))
```

When this is interpreted, pqR will not create a temporary object for `x>0`, and will stop comparing after finding that `x[2]>0`.

Could this be done with byte compilation? Not done in pqR, but perhaps it's possible.

Byte compilation becomes relatively less desirable when the CONS cells making up the AST are smaller, and when general symbol lookup is quicker, since the interpreter is then faster.