checkmate: Fast Argument Checks for Defensive R Programming

by Michel Lang

Abstract Dynamically typed programming languages like R allow programmers to write generic, flexible and concise code and to interact with the language using an interactive Read-eval-print-loop (REPL). However, this flexibility has its price: As the R interpreter has no information about the expected variable type, many base functions automatically convert the input instead of raising an exception. Unfortunately, this frequently leads to runtime errors deeper down the call stack which obfuscates the original problem and renders debugging challenging. Even worse, unwanted conversions can remain undetected and skew or invalidate the results of a statistical analysis. As a resort, assertions can be employed to detect unexpected input during runtime and to signal understandable and traceable errors. The package **checkmate** provides a plethora of functions to check the type and related properties of the most frequently used R objects and variable types. The package is mostly written in C to avoid any unnecessary performance overhead. Thus, the programmer can conveniently write concise, well-tested assertions which outperforms custom R code for many applications. Furthermore, **checkmate** simplifies writing unit tests using the framework **testthat** (Wickham, 2011) by extending it with plenty of additional expectation functions, and registered C routines are available for package developers to perform assertions on arbitrary SEXPs (internal data structure for R objects implemented as struct in C) in compiled code.

Defensive programming in R

Most dynamic languages utilize a weak type system where the type of variable must not be declared, and R is no exception in this regard. On the one hand, a weak type system generally reduces the code base and encourages rapid prototyping of functions. On the other hand, in comparison to strongly typed languages like C/C++, errors in the program flow are much harder to detect. Without the type information, the R interpreter just relies on the called functions to handle their input in a meaningful way. Unfortunately, many of R's base functions are implemented with the REPL in mind. Thus, instead of raising an exception, many functions silently try to auto-convert the input. E.g., instead of assuming that the input `NULL` does not make sense for the function `mean()`, the value `NA` of type numeric is returned and additionally a warning message is signaled. While this behaviour is acceptable for interactive REPL usage where the user can directly react to the warning, it is highly unfavorable in packages or non-interactively executed scripts. As the generated missing value is passed to other functions deeper down the call stack, it will eventually raise an error. However, the error will be reported in a different context and associated with different functions and variable names. The link to origin of the problem is missing and debugging becomes much more challenging. Furthermore, the investigation of the call stack with tools like `traceback()` or `browser()` can result in an overwhelming number of steps and functions. As the auto-conversions cascade nearly unpredictably (as illustrated in Table 1), this may lead to undetected errors and thus to misinterpretation of the reported results.

Input	Return value of `mean(x)`	`median(x)`	`sin(x)`	`min(x)`
`numeric(0)`	`NaN`	`NA`	`numeric(0)`	`Inf` (w)
`character(0)`	`NA_real_` (w)	`NA_character_`	[exception]	`NA_character_` (w)
`NA`	`NA_real_`	`NA`	`NA_real_`	`NA_integer_`
`NA_character_`	`NA_real_` (w)	`NA_character_`	[exception]	`NA_character_`
`NaN`	`NaN`	`NA`	`NaN`	`NaN`
`NULL`	`NA` (w)	`NULL` (w)	[exception]	`Inf` (w)

Table 1: Input and output for some simple mathematical functions from the **base** package (R-3.4.0). Outputs marked with "(w)" have issued a warning message.

As a final motivating example, consider the following function which uses the base functions `diff()` and `range()` to calculate the range $r = x_{\max} - x_{\min}$ of a numerical input vector `x`:

```
myrange <- function(x) {
  diff(range(x))
}
```

This function is expected to return a single numerical value, e.g. `myrange(1:10)` returns `9`. However, the calls `myrange(NULL)` and `myrange(integer(0))` also return such an unsuspicious, single numeric value: `-Inf`. Both calls signal warnings, but warnings can rather easily be overlooked and are hard to track down. It would arguably be much better to directly get an error message informing the user that something unintended is happening. In the worst case the warnings do not surface and – if this piece of code is embedded in a larger analysis – wrong conclusions are drawn.

The described problems lead to a concept called "defensive programming" where the programmer is responsible for manually checking function arguments. Reacting to unexpected input as soon as possible by signaling errors instantaneously with a helpful error message is the key aspect of this programming paradigm. A similar concept is called "design by contract" which demands the definition of formal, precise and verifiable input and in return guarantees a sane program flow if all preconditions hold. For `myrange()`, it would be necessary to insist that `x` is a numeric vector with at least one element to ensure a meaningful result. Additionally, missing values must be dealt with, either by completely prohibit them or by ensuring a meaningful return value. The package **checkmate** assists the programmer in writing such assertions in a concise way for the most important R variable types and objects. For `myrange()`, adding the line

```
assertNumeric(x, min.len = 1, any.missing = FALSE)
```

would be sufficient to ensure a sane program flow.

Related work

Many packages contain custom code to perform argument checks. These either rely on (a) the base function `stopifnot()` or (b) hand-written cascades of `if-else` blocks containing calls to `stop()`. Option (a) can be considered a quick hack because the raised error messages lack helpful details or instructions for the user. Option (b) is the natural way of doing argument checks in R but quickly becomes tedious. For this reason many packages have their own functions included, but there are also some packages on CRAN whose sole purpose are argument checks.

The package **assertthat** (Wickham, 2017) provides the "drop-in replacement" `assert_that()` for R's `stopifnot()` while generating more informative help messages. This is achieved by evaluating the expression passed to the function `assert_that()` in an environment where functions and operators from the base package (e.g. `as.numeric()` or `` `==` ``) are overloaded by more verbose counterparts. E.g., to check a variable to be suitable to pass to the `log()` function, one would require a numeric vector with all positive elements and no missing values:

```
assert_that(is.numeric(x), length(x) > 0, all(!is.na(x)), all(x >= 0))
```

For example, the output of the above statement for the input `c(1,NA,3)` reads `"Error: Elements 2 of !is.na(x) are not true"`. Additionally, **assertthat** offers some additional convenience functions like `is.flag()` to check for single logical values or `has_name()` to check for presence of specific names. These functions also prove useful if used with `see_if()` instead of `assert_that()` which turns the passed expression into a predicate function returning a logical value.

The package **assertive** (Cotton, 2016) is another popular package for argument checks. Its functionality is split over 16 packages containing over 400 functions, each specialized for a specific class of assertions: For instance, **assertive.numbers** specializes on checks of numbers and **assertive.sets** offers functions to work with sets. The functions are grouped into functions starting with `is_` for predicate functions and functions starting with `assert_` to perform `stopifnot()`-equivalent operations. The author provides a "checklist of checks" as package vignette to assist the user in picking the right functions for common situations like checks for numeric vectors or for working with files. Picking up the `log()` example again, the input check with **assertive** translates to:

```
assert_is_numeric(x)
assert_is_non_empty(x)
assert_all_are_not_na(x)
assert_all_are_greater_than_or_equal_to(x, 0)
```

The error message thrown by the first failing assertion for the input `c(1,NA,3)` reads `"is_not_na : The values of x are sometimes NA."`. Additionally, a `data.frame` is printed giving detailed information about "bad values":

```
  There was 1 failure:
  Position Value   Cause
1        2    NA missing
```

Moreover, the package **assertr** (Fischetti, 2016) focuses on assertions for **magrittr** (Bache and Wickham, 2014) pipelines and data frame operations in **dplyr** (Wickham and Francois, 2016), but is not intended for generic runtime assertions.

The checkmate package

Design goals

The package has been implemented with the following goals in mind:

Runtime To minimize any concern about the extra computation time required for assertions, most functions directly jump into compiled code to perform the assertions directly on the SEXPs. The functions also have been extensively optimized to first perform inexpensive checks in order to be able to skip the more expensive ones.

Memory In many domains the user input can be rather large, e.g. long vectors and high dimensional matrices are common in text mining and bioinformatics. Basic checks, e.g. for missingness, are already quite time consuming, but if intermediate objects of the same dimension have to be created, runtimes easily get out of hand. For example, `any(x < 0)` with `x` being a large numeric matrix internally first allocates a logical matrix `tmp` with the same dimensions as `x`. The matrix `tmp` is then passed in a second step to `any()` which aggregates the logical matrix to a single logical value and `tmp` is marked to be garbage collected. Besides a possible shortage of available memory, which may cause the machine to swap or the R interpreter to terminate, runtime is wasted with unnecessary memory management. **checkmate** solves this problem by looping directly over the elements and thereby avoiding any intermediate objects.

Code completion The package aims to provide a single function for all frequently used R objects and their respective characteristics and attributes. This way, the built-in code completion of advanced R editors assist in finding the suitable further restrictions. For example, after typing the function name and providing the object to check (`"assertNumeric(x,"`), many editors look up the function and suggest additional function arguments via code completion. In this example, checks to restrict the length, control the missingness or setting lower and upper bounds are suggested. These suggestions are restrictions on `x` specific for the respective base type (numeric). Such context-sensitive assistance helps writing more concise assertions.

The focus on runtime and memory comes at the price of error messages being less informative in comparison to **assertthat** or **assertive**. Picking up the previous example with input `c(1,NA,3)`, **checkmate**'s `assertNumeric(x,any.missing = FALSE,lower = 0)` immediately raises an exception as soon as the `NA` is discovered at position 2. The package refrains from reporting an incomplete list of positions of missing values, instead the error message just reads `"Assertion on 'x' failed: Contains missing values."`. Thus, the implementations in **assertive** or **assertr** are better suited to find bad values in data and especially in data frames. **checkmate** is designed to amend functions with quick assertions in order to ensure a sane program flow. Nevertheless, the provided information is sufficient to quickly locate the error and start investigations with the debugging tools provided by R.

Naming scheme

The core functions of the package follow a specific naming scheme: The first part (prefix) of a function name determines the action to perform w.r.t. the outcome of the respective check while the second part of a function name (suffix) determines the base type of the object to check. The first argument of all functions is always the object `x` to check and further arguments specify additional restrictions on `x`.

Prefixes

There are currently four families of functions, grouped by their prefix, implemented in **checkmate**:

assert* Functions prefixed with "assert" throw an exception if the corresponding check fails and the checked object is returned invisibly on success. This family of functions is suitable for many different tasks. Besides argument checks of user input, this family of functions can also be used as a drop-in replacement for `stopifnot()` in unit tests using the internal test mechanism of R as described in Writing R Extensions (R Core Team, 2016), Subsection 1.1.5. Furthermore, as the object to check is returned invisibly, the functions can also be used inside **magrittr** pipelines.

test* Functions prefixed with "test" are predicate functions which return `TRUE` if the respective check is successful and `FALSE` otherwise. This family of functions is best utilized if different checks must be combined in a non-trivial manner or custom error messages are required.

expect* Functions prefixed with "expect" are intended to be used together with **testthat** (Wickham, 2011): the check is translated to an expectation which is then forwarded to the active **testthat** reporter. This way, **checkmate** extends the facilities of **testthat** with dozens of powerful helper functions to write efficient and comprehensive unit tests. Note that **testthat** is an optional dependency and the `expect`-functions only work if **testthat** is installed. Thus, to use **checkmate** as an **testthat** extension, **checkmate** must be listed in `Suggests` or `Imports` of a package.

check* Functions prefixed with "check" return the error message as a string if the respective check fails, and `TRUE` otherwise. Functions with this prefix are the workhorses called by the "assert", "test" and "expect" families of functions and prove especially useful to implement custom assertions. They can also be used to collect error messages in order to generate reports of multiple check violations at once.

The prefix and the suffix can be combined in both "camelBack" and "underscore_case" fashion. In other words, checkmate offers all functions with the "assert", "test" and "check" prefix in both programming style flavors: `assert_numeric()` is a synonym for `assertNumeric()` the same way `testDataFrame()` can be used instead of `test_data_frame()`. By supporting the two most predominant coding styles for R, most programmers can stick to their favorite style while implementing runtime assertions in their packages.

Suffixes

While the prefix determines the action to perform on a successful or failed check, the second part of each function name defines the base type of the first argument `x`, e.g. `integer`, `character` or `matrix`. Additional function arguments restrict the object to fulfill further properties or attributes.

Atomics and Vectors The most important built-in atomics are supported via the suffixes `*Logical`, `*Numeric`, `*Integer`, `*Complex`, `*Character`, `*Factor`, and `*List` (strictly speaking, "numeric" is not an atomic type but a naming convention for objects of type `integer` or `double`). Although most operations that work on real values also are applicable to natural numbers, the contrary is often not true. Therefore numeric values frequently need to be converted to integer, and `*Integerish` ensures a conversion without surprises by checking double values to be "nearby" an integer w.r.t. a machine-dependent tolerance. Furthermore, the object can be checked to be a `vector`, an `atomic` or an atomic vector (a `vector`, but not `NULL`).

All functions can optionally test for missing values (any or all missing), length (exact, minimum and maximum length) as well as names being (a) not present, (b) present and not `NA`/empty, (c) present, not `NA`/empty and unique, or (d) present, not `NA`/empty, unique and additionally complying to R's variable naming scheme. There are more type-specific checks, e.g. bound checks for numerics or regular expression matching for characters. These are documented in full detail in the manual.

Scalars Atomics of length one are called scalars. Although R does not differentiate between scalars and vectors internally, scalars deserve particular attention in assertions as arguably most function arguments are expected to be scalar. Although scalars can also be checked with the functions that work on atomic vectors and additionally restricting to length 1 via argument `len`, **checkmate** provides some useful abbreviations: `*Flag` for logical scalars, `*Int` for an integerish value, `*Count` for a non-negative integerish values, `*Number` for numeric scalars and `*String` for scalar character vectors. Missing values are prohibited for all scalar values by default as scalars are usually not meant to hold data where missingness occurs naturally (but can be allowed explicitly via argument `na.ok`). Again, additional type-specific checks are available which are described in the manual.

Compound types The most important compound types are matrices/arrays (vectors of type logical, numeric or character with attribute `dim`) and data frames (lists with attribute `row.names` and class `data.frame` storing atomic vectors of same length). The package also includes checks for the popular `data.frame` alternatives **data.table** (Dowle et al., 2017) and **tibble** (Wickham et al., 2017). Some checkable characteristics conclude the internal type(s), missingness, dimensions or dimension names.

Miscellaneous On top of the already described checks, there are functions to work with sets (`*Subset`, `*Choice` and `*SetEqual`), environments (`*Environment`) and objects of class "Date" (`*Date`). The `*Function` family checks R functions and its arguments and `*OS` allows to check if R is running on a specific operating system. The functions `*File` and `*Directory` test for existence and access rights of files and directories, respectively. The function `*PathForOutput` allows to check whether a directory can be used to store files in it. Furthermore, **checkmate** provides functions to check the class or names of arbitrary R objects with `*Class` and `*Names`.

Custom checks Extensions are possible by writing a `check*` function which returns `TRUE` on success and an informative error message otherwise. The exported functionals `makeAssertionFunction()`, `makeTestFunction()` and `makeExpectationFunction()` can wrap this custom check function to create the required counterparts in such a way that they seamlessly fit into the package. The vignette demonstrates this with a check function for square matrices.

DSL for argument checks

Most basic checks can alternatively be performed using an implemented Domain Specific Language (DSL) via the functions `qassert()`, `qtest()` or `qexpect()`. All three functions have two arguments: The arbitrary object `x` to check and a "rule" which determines the checks to perform provided as a single string. Each rules consist of up to three parts:

1. The first character determines the expected class of `x`, e.g. "n" for numeric, "b" for boolean, "f" for a factor or "s" for a string (more can be looked up in the manual). By using a lowercase letter, missing values are permitted while an uppercase letter disallows missingness.
2. The second part is the length definition. Supported are "?" for length 0 or length 1, "+" for length ≥ 1 as well as arbitrary length specifications like "1"/"==1" for exact length 1 or "<10" for length < 10.
3. The third part triggers a range check, if applicable, in interval notation (e.g., "$[0,1)$" for values $0 \leq x < 1$). If the boundary value on an open side of the interval is missing, all values of `x` will be checked for being $> -\infty$ or $< \infty$, respectively.

Although this syntax requires some time to familiarize with, it allows to write extensive argument checks with very few keystrokes. For example, the previous check for the input of `log()` translates to the rule `"N+[0,]"`. More examples can be found in Table 2

Base R	DSL
Single string `is.character(x) && length(x) == 1`	`"s1"`
Factor with minimum length 1, no missing values `is.factor(x) && length(x) >= 1 && all(!is.na(x))`	`"F+"`
List with no missing elements (NULL interpreted as missing for lists) `is.list(x) && !any(sapply(x,is.null))`	`"L"`
Integer of length 3 with positive elements `is.integer(x) && length(x) == 3 && all(x >= 0)`	`"i3[0]"`
Single number representing a proportion ($x \in [0,1]$) `is.numeric(x) && length(x) == 1 && !is.na(x) && x >= 0 && x <= 1`	`"N1[0,1]"`
Numeric vector with non-missing, positive, finite elements `is.numeric(x) && all(!is.na(x) & x >= 0 & is.finite(x))`	`"N[0,)"`
NULL or a single string with at least one character `is.null(x) || (is.character(x) && length(x) == 1 && nzchar(x))`	`"0"`, `"s1[1]"`

Table 2: Exemplary checks using base R and the abbreviations implemented in the DSL.

As the function signature is really simplistic, it is perfectly suited to be used from compiled code written in C/C++ to check arbitrary `SEXP`s. For this reason **checkmate** provides header files which third-party packages can link against. Instructions can be found in the package vignette.

Benchmarks

This small benchmark study picks up the log() example once again: testing a vector to be numeric with only positive, non-missing values.

Implementations

Now we compare **checkmate**'s `assertNumeric()` and `qassert()` (as briefly described in the previous Section DSL for argument checks) with counterparts written with R's `stopifnot()`, **assertthat**'s `assert_that()` and a series of **assertive**'s `assert_*()` functions:

```
checkmate <- function(x) { assertNumeric(x, any.missing = FALSE, lower = 0) }
qcheckmate <- function(x) { qassert(x, "N[0,]") }
R <- function(x) { stopifnot(is.numeric(x), all(!is.na(x)), all(x >= 0)) }
assertthat <- function(x) { assert_that(is.numeric(x), all(!is.na(x)), all(x >= 0)) }
assertive <- function(x) { assert_is_numeric(x); assert_all_are_not_na(x);
  assert_all_are_greater_than_or_equal_to(x, 0) }
```

To allow measurement of failed assertions, the wrappers are additionally wrapped into a `try()` statement. Note that all functions perform the checks in the same order: First they check for type numeric, then for missing values and finally for all elements being non-negative. The source code for this benchmark study is hosted on **checkmate**'s project page in the directory `inst/benchmarks`.

Setup

The benchmark was performed on an Intel i5-6600 with 16 GB running R-3.4.0 on a 64bit Arch Linux installation using a 4.10.11 kernel. The package versions are 1.8.2 for **checkmate**, 0.2 for **assertthat** and 0.3.5 for **assertive**. R, the linked OpenBLAS and all packages have been compiled with the GNU Compiler Collection (GCC) in version 6.3.1 and tuned with `march=native` on optimization level `-O2`. To compare runtime differences, **microbenchmark** (Mersmann, 2015) is set to do 100 replications. The implemented wrappers have also been compared to their byte-compiled version (using `compiler::cmpfun`) with no notable difference in performance. The just-in-time compiler which is enabled per default as of R-3.4.0 causes a small but non-crucial decrease in performance for all implementations. The presented results are extracted from the uncompiled versions of these wrappers, with the JIT enabled on the default level 3.

Memory consumption is measured with the `benchexec` (Beyer et al., 2015) framework, version 1.10. Unlike R's `gc()` which only keeps track of allocations of SEXPs, `benchexec` measures all allocations (e.g., using C's `malloc`) and also works well with threads and child processes. Note that setting an upper memory limit is mandatory to ensure comparable measurements for memory consumption. Here, all processes are started with an upper limit of 2 GB.

Results

The benchmark is performed on four different inputs and the resulting timings are presented in Figure 1. Note that the runtimes on the x-axis are on $\log_{10}$-scale and use different units of measurement.

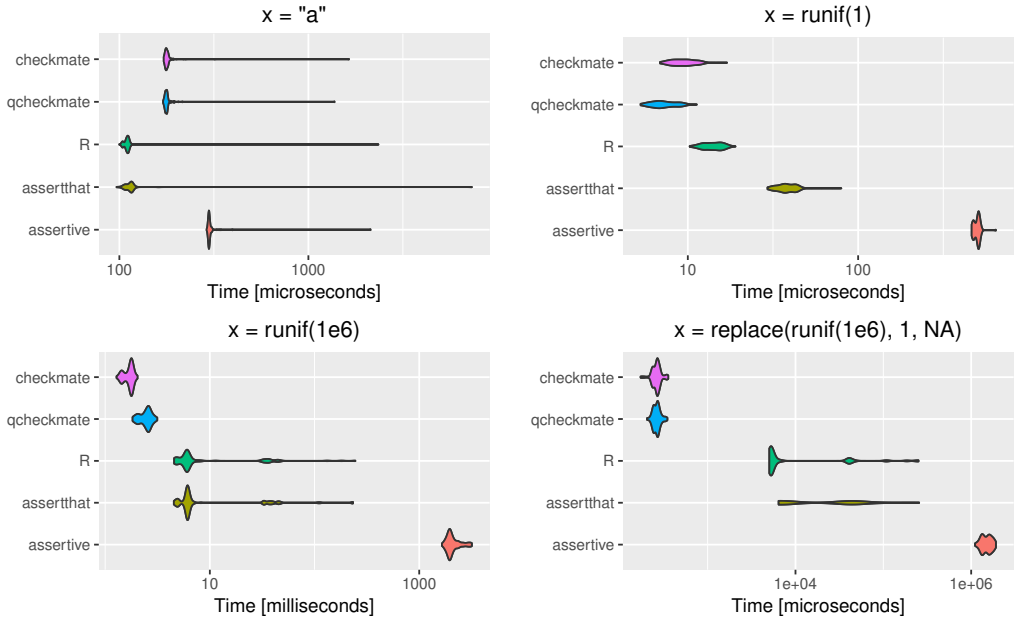

Figure 1: Violin plots of the runtimes on $\log_{10}$-scale of the assertion "x must be a numeric vector with all elements positive and no missing values" on different input `x`.

top left Input `x` is a scalar character value, i.e. of wrong type. This benchmark serves as a measurement of overhead: the first performed (and cheapest) assertion on the type of `x` directly fails. In fact, all

assertion frameworks only require microseconds to terminate. R directly jumps into compiled code via a `Primitive` and therefore has the least overhead. **checkmate** on the other hand has to jump into the compiled code via the `.Call` interface which is comparably slower. The implementation in **assertthat** is faster than **checkmate** (as it also primarily calls primitives) but slightly slower than `stopifnot()`. The implementation in **assertive** is the slowest. However, in case of assertions (in comparison to tests returning logical values), the runtimes for a successful check are arguably more important than for a failed check because the latter raises an exception which usually is a rare event in the program flow and thus not time-critical. Therefore, the next benchmark might be more relevant for many applications.

top right Here, the input `x` is a valid numeric value. The implementations now additionally check for both missingness and negative values without raising an exception. The DSL variant `qassert()` is the fastest implementation, followed by **checkmate**'s `assertNumeric()`. Although `qassert()` and `assertNumeric()` basically call the same code internally, `qassert()` has less overhead due to its minimalist interface. R's `stopifnot()` is a tad slower (comparing the median runtimes) but still faster than **assertthat** (5x slowdown in comparison to `qassert()`). **assertive** is >70x slower than `qassert()`.

bottom left Input `x` is now a long vector with 10^6 numeric elements. This vector is generated using `runif()` and thus all values are valid (no negative or missing values). **checkmate** has the fastest versions with a speedup of approximately 3.5x compared to R's `stopifnot()` and `assert_that()`. In comparison to its alternatives, **checkmate** avoids intermediate objects as described in Design goals: Instead of allocating a `logical(1e6)` vector first to aggregate it in a second step, **checkmate** directly operates on the numeric input. That is also the reason why `stopifnot()` and `assertthat()` have high variance in their runtimes: The garbage collector occasionally gets triggered to free memory which requires a substantial amount of time.

assertive is orders of magnitude slower for this input (>1200x) because it follows a completely different philosophy: Instead of focusing on speed, **assertive** gathers detailed information while performing the assertion. This yields report-like error messages (e.g., the index and reason why an assertion failed, for each element of the vector) but is comparably slow.

bottom right Input `x` is again a large vector, but the first element is a missing value. Here, all implementations first successfully check the type of `x` and then throw an error about the missing value at position 1. Again, **checkmate** avoids allocating intermediate objects which in this case yields an even bigger speedup: While the other packages first check 10^6 elements for missingness to create a `logical(1e6)` vector which is then passed to `any()`, **checkmate** directly stops after analyzing the first element of `x`. This obvious optimization yields a speedup of 25x in comparison to R and **assertthat** and a 7000x speedup in comparison to **assertive**.

Note that this is the best case scenario for early stopping to demonstrate the possible effect memory allocation can have on the runtime. Triggering the assertion on the last element of the vector results in runtimes roughly equivalent to the previous benchmark displayed at bottom left. A randomly placed bad value yields runtimes in the range of these two extremes.

Memory has been measured using `x = runif(1e7)` as input (similar to the setup of the benchmark shown in the bottom left of Figure 1). Measurements are repeated 100 times in independent calls of `Rscript` via the command line tool `runexec` and summarized by the mean and standard deviation. A no-operation (startup of R, creating the vector `x` and terminating) requires 105.0 ± 0.4 MB. The same script which additionally loads the **checkmate** package and performs the assertion with the above defined wrapper `checkmate()` does not increase the memory footprint (105.1 ± 0.2 MB) notably. Same for the previously defined wrapper `qcheckmate()` (105.0 ± 0.1 MB). The equivalent assertion using base R requires 185.1 ± 0.1 MB, about the same as the implementation in **assertthat** (185.1 ± 0.1 MB). Using **assertive**, 1601.8 ± 0.6 MB are required to run the script.

Summed up, **checkmate** is the fastest option to perform expensive checks and only causes a small decrease in performance for trivial, inexpensive checks which fail quickly (top left). Although the runtime differences seem insignificant for small input (top right), the saved microseconds can easily sum up to minutes or hours if the respective assertion is located in a hot spot of the program and therefore is called millions of times. By avoiding intermediate objects, assertions have virtually no memory overhead. This saves runtime in the garbage collection, and even becomes much more important as data grows bigger.

Conclusion

Runtime assertions are a necessity in R to ensure a sane program flow, but R itself offers very limited capabilities to perform these kind of checks. **checkmate** allows programmers and package developers to write assertions in a concise way without unnecessarily sacrificing runtime performance nor

increasing the memory footprint. Compared to the presented alternatives, assertions with **checkmate** are faster, tailored for bigger data and (with the help of code completion) more convenient to write. They generate helpful error messages, are extensively tested for correctness and suitable for large and extensive software projects (**mlr** (Bischl et al., 2016), **BatchJobs** (Bischl et al., 2015) and **batchtools** (Lang et al., 2017) already make heavy use of **checkmate**). Furthermore, **checkmate** offers capabilities to perform assertions on SEXPs in compiled code via a domain specific language and extends the popular unit testing framework **testthat** with many helpful expectation functions.

Acknowledgments

Part of the work on this paper has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A3 (http://sfb876.tu-dortmund.de).

Bibliography

S. M. Bache and H. Wickham. Magrittr: A Forward-Pipe Operator for R, 2014. URL https://cran.r-project.org/package=magrittr. [p439]

D. Beyer, S. Löwe, and P. Wendler. Benchmarking and Resource Measurement. In *Model Checking Software*, pages 160–178. Springer-Verlag, 2015. URL https://doi.org/10.1007/978-3-319-23404-5_12. [p442]

B. Bischl, M. Lang, O. Mersmann, J. Rahnenführer, and C. Weihs. BatchJobs and BatchExperiments: Abstraction Mechanisms for Using R in Batch Environments. *Journal of Statistical Software*, 64(11): 1–25, 2015. ISSN 1548-7660. URL https://doi.org/10.18637/jss.v064.i11. [p444]

B. Bischl, M. Lang, L. Kotthoff, J. Schiffner, J. Richter, E. Studerus, G. Casalicchio, and Z. M. Jones. Mlr: Machine Learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016. URL http://www.jmlr.org/papers/v17/15-066.html. [p444]

R. Cotton. Assertive: Readable Check Functions to Ensure Code Integrity, 2016. URL https://cran.r-project.org/package=assertive. R package version 0.3-5. [p438]

M. Dowle, A. Srinivasan, J. Gorecki, T. Short, S. Lianoglou, and E. Antonyan. Data.table: Extension of 'data.frame', 2017. URL https://cran.r-project.org/package=data.table. [p440]

T. Fischetti. Assertive Programming for R Analysis Pipelines, 2016. URL https://cran.r-project.org/package=assertr. R package version 1.0.2. [p439]

M. Lang, B. Bischl, and D. Surmann. Batchtools: Tools for R to work on batch systems. *The Journal of Open Source Software*, 2(10), 2017. ISSN 2475-9066. URL https://doi.org/10.21105/joss.00135. [p444]

O. Mersmann. Microbenchmark: Accurate Timing Functions, 2015. URL https://cran.r-project.org/package=microbenchmark. R package version 1.4-2.1. [p442]

R Core Team. Writing R Extensions, 2016. URL https://cran.r-project.org/doc/manuals/r-release/R-exts.html. [p439]

H. Wickham. Testthat: Get Started with Testing. *The R Journal*, 3(1):5–10, 2011. URL https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf. [p437, 440]

H. Wickham. Assertthat: Easy pre and post assertions, 2017. URL https://cran.r-project.org/package=assertthat. R package version 0.2. [p438]

H. Wickham and R. Francois. Dplyr: A Grammar of Data Manipulation, 2016. URL https://cran.r-project.org/package=dplyr. R package version 0.5.0. [p439]

H. Wickham, R. Francois, K. Müller, and RStudio. Tibble: Simple Data Frames, 2017. URL https://cran.r-project.org/package=tibble. [p440]

Michel Lang
TU Dortmund University, Faculty of Statistics
Vogelpothsweg 87, 44227 Dortmund

Germany
ORCiD: 0000-0001-9754-0393
lang@statistik.tu-dortmund.de