DE GRUYTER OPEN

JACSM 2017, Vol. 9, No. 1, pp. 65 - 78
10.1515/jacsm-2017-0004

chR: Dynamic Functional Constraints Checking in R

Konrad Grzanek

IT Institute, University of Social Sciences
9 Sienkiewicza St., 90-113 Lodz, Poland
kgrzanek@spoleczna.pl

Abstract

Dynamic typing of R programming language may issue some quality problems in large scale data-science and machine-learning projects for which the language is used. Following our efforts on providing gradual typing library for Clojure we come with a package chR - a library that offers functionality of run-time type-related checks in R. The solution is not only a dynamic type checker, it also helps to systematize thinking about types in the language, at the same time offering high expressivenes and full adherence to functional programming style.

Keywords: Formal software verification, software quality, dynamic type-checking, functional programming, category theory, R

1 Introduction

Writing software in dynamically typed programming languages requires as much attention with respect to types of expressions as when using a statically and strongly typed one ([2], [1]). One popular and apparently natural approach is to use gradual typing - a process of selectively adding type checks to expressions, mostly to the critical parts of computer programs. With the approach a programmer can decide where to put checks and which parts are so “obvious”, that they do not have to be verified.

R programming language [5] has been growing in use in recent years, together with a growth of computer science and software engineering sub-domains to which it has been targeted: data science and statistical- (more broadly machine-) learning. Unfortunately, it lacks a decent type-checking solution. The great *assertthat* package [7] addresses a slightly different problem: putting generic run-time assertions into R codes. We need a package with the following properties:

– being deeply rooted in functional programming [4] and using notions from the category theory

– overall consistency with the dynamic and in a way Lispy nature of R programming language and adherence to functional programming style with purrr library [6]
– being as fast as possible, using checks that use as low-level elements of the R base (standard library) as possible
– expressiveness, ease of use, and extendability

Our previous work on dependent typing resulted in a *ch* library ([11], [12], [13]) for Clojure programming language ([8], [9]). Following that we decided to create a corresponding package for R. The package is called *chR* [10] and it is a subject of further analysis in this paper.

2 Essentials of the chR Library

The heart of our solution is *ch* procedure. In essence it executes a predicate (*pred*) on an argument *x*. If the predicate returns *false* value (or a value effectively) effectively equal to *false*, an error is raised. Otherwise *x* is returned. This behavior allows a greater composability and support for functional programming style. We may easily put any ch(eck) in a pipeline of data processing procedures, as will be presented further.

Procedure *ch* works also in a predicate-only mode. This mode is necessary when a ch(eck) is used as a sub-component of a larger one. Below we have the *ch* code together with an error generator (*errMessage*):

```
errMessage <- function(x) {
  r <- paste(capture.output(str(x)), collapse = "\n")
  paste0(" ch(eck) failed on\n", r)
}

#' Executes a ch(eck) of pred on x
#' @export
ch <- function(pred, x, asPred = FALSE) {
  r <- pred(x)
  if (asPred) return(r)
  if (!r)     stop(errMessage(x))
  x
}
```

Effective use of *chR* library starts with the following procedure that takes a predicate and returns a corresponding ch(eck). The returned ch(eck) takes an argument *x* and applies *ch* working by default in non-pred mode:

```
#' Returns a ch(eck) based on the pred
#' @export
chP <- function(pred) {
  function(x, asPred = FALSE) ch(pred, x, asPred)
}
```

For classes in R we have a *chInstance* ch(eck)s generator that uses a common *inherits* procedure belonging to R base:

```
#' Returns a \code{inherits(., cls)} ch(eck)
#' @export
chInstance <- function(class) chP(function(x)
  inherits(x, class))
```

An intrinsic property of an expressive language (including an embedded one) is composability. Our ch(eck)s compose. The three composition operators are logic-oriented and they reflect the most basic logical operations. We have negation:

```
#' Returns a ch(eck) that is a negation of the passed ch(eck)
#' @export
chNot <- function(c) chP(function(x)
  !c(x, asPred = TRUE))
```

Also, there is conjunction:

```
#' Returns a ch(eck) that &s all the passed ch(eck)s
#' @export
chAnd <- function(...) {
  chs  <- list(...)
  chP(function(x) {
    for (c in chs) if (!c(x, asPred = TRUE)) return (FALSE)
    TRUE
  })
}
```

and alternative:

```
#' Returns a ch(eck) that |s all the passed ch(eck)s
#' @export
chOr <- function(...) {
  chs <- list(...)
  chP(function(x) {
    for (c in chs) if (c(x, asPred = TRUE)) return (TRUE)
    FALSE
  })
}
```

In the two latter ones we assume a non-restricted number of composed ch(eck)s.

3 Fundamental Ch(eck)s

Many functional programming languages rooted in category theory ([3]), e.g. Haskell, use a unit type and unit value. Although R fully supports functional style of programming, it does not unify notion of no-values. We made an arbitrary decision to treat NULL as unit value. The decision was based upon pragmatics in the technology. A corresponding ch(eck) follows:

```
#' \code{is.null} ch(eck)
#' @export
chUnit <- chP(is.null)

#' \code{!is.null} ch(eck)
#' @export
chSome <- chP(function(x) !is.null(x))
```

The *chSome* ch(eck) is an opposite to *chUnit*, as can be seen above. In R programming language we have both NULL as well as NA values. Thus, a separate ch(eck) for NAs is needed:

```
#' \code{is.na} ch(eck)
#' @export
chNA <- chAnd(chScalar, chP(is.na))
```

For two-element *Discriminated Union Types* we have the following *chEither* ch(eck):

```
#' Either ch(eck) where the left and right types are
#' expressed by checks cl and cr
#' @export
chEither <- function(cl, cr, x, asPred = FALSE)
  chOr(cl, cr)(x, asPred)
```

that used with *chUnit* forms a *Maybe* ch(eck):

```
#' Maybe ch(eck)
#' @export
chMaybe <- function(c, x, asPred = FALSE)
  chEither(chUnit, c, x, asPred)
```

Because R uses only vectorized values (and lists), we need ch(eck)s for scalars, hereby treated as one-element atoms (vectors):

```
#' Scalar \code{is.atomic} & \code{length == 1L}
#' value ch(eck)
#' @export
chScalar <- chP(function(x)
  is.atomic(x) && length(x) == 1L)
```

With the new ch(eck) we can define e.g. a ch(eck) for either a String vector of any length or a single String:

```
#' \code{is.character} ch(eck)
#' @export
chStrings <- chP(is.character)

#' \code{chScalar} & \code{chStrings} ch(eck)
#' @export
chString <- chAnd(chScalar, chStrings)
```

Another essential ch(eck) is for R functions:

```
#' \code{is.function} ch(eck)
#' @export
chFun <- chP(is.function)
```

This shortened presentation ends a section about the most common ch(ecks)s in *chR* library. For more, please read Appendix A.

4 Registry of Ch(eck)s

Additionally the *chR* library (like its ancestor *ch* for Clojure) provides a registry of ch(eck)s, that helps the programmer to understand, what kind of ch(eck)s an object or a collection of objects fulfill. The registry is an associative container that can be used to put a relation between a ch(eck) symbol (name) and the ch(eck):

```
CHSREG <- list()

#' Registeres the ch(eck) using an optional name
#' (ch(eck) name by default)
#' @export
chReg <- function(ch, name = NA) { # BEWARE: THREAD UNSAFE
  if (is.na(name)) name <- as.character(substitute(ch))
  CHSREG[[as.character(name)]] <<- as.function(ch)
  NULL
}
```

After we register selected ch(eck)s like below:

```
chReg(chUnit)
chReg(chScalar)
chReg(chString)
chReg(chStrings)
chReg(chFun)
```

we can ask the library about what kinds of ch(eck)s a given object fulfills:

```
chR::chs(1:10)

[1] "chAtomic"   "chInts"     "chNatInts"
[4] "chNumerics" "chPosInts"  "chSome"
[7] "chVector"
```

5 Supporting C++ Codes (via Rcpp)

Some of the ch(eck)s are better implemented in a low-level programming language. Thankfully R supports easy extensions in C++ written in effective library Rcpp. In *chR* the following procedure is defined to allow evaluation of predicates on vectors of any (presumably numeric) types:

```
template<typename V, typename F>
static inline bool everyInVector(const V xs,
                                 const F&& pred) {
  const int n = xs.size();
  for (int i = 0; i < n; i++)
    if (!pred(xs[i])) return false;

  return true;
}
```

The procedure is used to implement the predicates on vectors of doubles, as presented below:

```
//' Returns true iff all the xs are positive
//' @param xs vector to check
//' @return true or false
//' @export
// [[Rcpp::export]]
bool arePosDoubles(const DoubleVector xs) {
  return everyInVector(xs, [](double d)
    { return d > 0; });
}

//' Returns true iff all the xs are negative
//' @param xs vector to check
//' @return true or false
//' @export
// [[Rcpp::export]]
bool areNegDoubles(const DoubleVector xs) {
  return everyInVector(xs, [](double d)
    { return d < 0; });
}
```

```
//' Returns true iff all the xs are non-negative
//' @param xs vector to check
//' @return true or false
//' @export
// [[Rcpp::export]]
bool areNonNegDoubles(const DoubleVector xs) {
  return everyInVector(xs, [](double d)
    { return d >= 0; });
}
```

Accordingly, there are the preds for *IntegerVector*:

```
//' Returns true iff all the xs are positive
//' @param xs vector to check
//' @return true or false
//' @export
// [[Rcpp::export]]
bool arePosInts(const IntegerVector xs) {
  return everyInVector(xs, [](int n)
    { return n > 0; });
}

//' Returns true iff all the xs are negative
//' @param xs vector to check
//' @return true or false
//' @export
// [[Rcpp::export]]
bool areNegInts(const IntegerVector xs) {
  return everyInVector(xs, [](int n)
    { return n < 0; });
}

//' Returns true iff all the xs are naturals (>= 0)
//' @param xs vector to check
//' @return true or false
//' @export
// [[Rcpp::export]]
bool areNatInts(const IntegerVector xs) {
  return everyInVector(xs, [](int n)
    { return n >= 0; });
}
```

What's interesting is use of C++ lambdas in the procedures above. They are no-cost an highly expressive. Their use is possible in C++11 and above. Current Rcpp implementation supports that language standard.

6 Cases of Use in Production Setting

Our library is currently used in at least three commercial products. The usefulness of ch(eck)s can be seen in the following procedure, whose goal is to read employees' absences information in a business intelligence project:

```
readAbsences <- function(file) chDT({
  chString(file)
  absncs <- fread(file) %>%
    assertDTcolnames(ABSENCES_PROPS)

  for (p in ABSENCES_DATE_PROPS)
    set(absncs, j = p, value =
          parseDates(parse_character(absncs[[p]])))

  absncs %>% setDTcolorder(ABSENCES_PROPS)
  setkey(absncs, "Employee Number")
  absncs
})
```

The argument *file* is intended to be a String (*chString(file)* ch(eck)) and the result of the procedure is a *data.table* object (*chDT(*{...}*)* ch(eck)). Apparently, the system of ch(eck)s not only increases software correctness, but also has a positive impact on readability of codes.

References

1. Pierce B.C., 2002, *Types and Programming Languages, 1st Edition*, MIT Press, ISBN-10: 0262162091, ISBN-13: 978-0262162098

2. Pierce B.C., 2004, *Advanced Topics in Types and Programming Languages*, MIT Press, ISBN-10: 0262162288, ISBN-13: 978-0262162289

3. Awodey S., 2010, *Category Theory, Second Edition*, Oxford University Press

4. Bird R., Wadler R., 1988, *Introduction to Functional Programming*. Series in Computer Science (Editor: C.A.R. Hoare), Prentice Hall International (UK) Ltd

5. R Core Team, 2017, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, https://www.R-project.org/

6. Henry L., Wickham H., 2017, *purrr: Functional Programming Tools*, R package version 0.2.4, https://CRAN.R-project.org/package=purrr

7. Wickham H., 2017, *assertthat: Easy Pre and Post Assertions*, R package version 0.2.0, https://CRAN.R-project.org/package=assertthat

8. Emerick Ch., Carper B., Grand Ch., 2012, *Clojure Programming*, O'Reilly Media Inc., ISBN: 978-1-449-39470-7

9. Fogus M., Houser Ch., 2014, *The Joy of Clojure*, Manning Publications; 2 edition, ISBN-10: 1617291412, ISBN-13: 978-1617291418

10. Grzanek K., 2017, *chR Github Repository*, https://github.com/kongra/chR

11. Grzanek K., 2016, *Low-Cost Dynamic Constraint Checking for the JVM*, Journal of Applied Computer Science Methods, 8(2), pp. 115-136, doi:10.1515/jacsm-2016-0008

12. Grzanek K., 2017, *ch GitHub Repository*, https://github.com/kongra/ch

13. Grzanek K., 2017, *ch Clojars Page*, https://clojars.org/kongra/ch

A Appendix: Selected Pre-defined Ch(eck)s

```
#' \code{is.logical} ch(eck)
#' @export
chBools <- chP(is.logical)

#' \code{chScalar} & \code{chBools} ch(eck)
#' @export
chBool <- chAnd(chScalar, chBools)

#' \code{is.integer} ch(eck)
#' @export
chInts <- chP(is.integer)

#' \code{chScalar} & \code{chInts} ch(eck)
#' @export
chInt <- chAnd(chScalar, chInts)

#' \code{is.double} ch(eck)
#' @export
chDoubles <- chP(is.double)

#' \code{chScalar} & \code{chDoubles} ch(eck)
#' @export
chDouble <- chAnd(chScalar, chDoubles)

#' \code{is.complex} ch(eck)
#' @export
chComplexes <- chP(is.complex)
```

```
#' \code{chScalar} & \code{chComplexes} ch(eck)
#' @export
chComplex <- chAnd(chScalar, chComplexes)

#' \code{is.numeric} ch(eck)
#' @export
chNumerics <- chP(is.numeric)

#' \code{chScalar} & \code{chNumerics} ch(eck)
#' @export
chNumeric  <- chAnd(chScalar, chNumerics)

#' \code{chInts} & > 0 ch(eck)
#' @export
chPosInts <- chAnd(chInts, chP(arePosInts))

#' \code{chInt} & > 0 ch(eck)
#' @export
chPosInt <- chAnd(chInt, chP(arePosInts))

#' \code{chDoubles} & > 0 ch(eck)
#' @export
chPosDoubles <- chAnd(chDoubles, chP(arePosDoubles))

#' \code{chDouble} & > 0 ch(eck)
#' @export
chPosDouble <- chAnd(chDouble, chP(arePosDoubles))

#' \code{chInts} & < 0 ch(eck)
#' @export
chNegInts <- chAnd(chInts, chP(areNegInts))

#' \code{chInt} & < 0 ch(eck)
#' @export
chNegInt <- chAnd(chInt, chP(areNegInts))

#' \code{chDoubles} & < 0 ch(eck)
#' @export
chNegDoubles <- chAnd(chDoubles, chP(areNegDoubles))

#' \code{chDouble} & < 0 ch(eck)
#' @export
chNegDouble <- chAnd(chDouble, chP(areNegDoubles))

#' \code{chInts} & >= 0 ch(eck)
```

```
#' @export
chNatInts <- chAnd(chInts, chP(areNatInts))

#' \code{chInt} & >= 0 ch(eck)
#' @export
chNatInt <- chAnd(chInt, chP(areNatInts))

#' \code{chDoubles} & >= 0 ch(eck)
#' @export
chNonNegDoubles <- chAnd(chDoubles, chP(areNonNegDoubles))

#' \code{chDouble} & >= 0 ch(eck)
#' @export
chNonNegDouble <- chAnd(chDouble, chP(areNonNegDoubles))

#' \code{chInt} & even? check
#' @export
chEvenInt <- chAnd(chInt, chP(function (x) x %% 2L == 0L))

#' \code{chInt} & odd? check
#' @export
chOddInt <- chAnd(chInt, chP(function (x) x %% 2L != 0L))

#' \code{is.list} ch(eck)
#' @export
chList <- chP(is.list)

#' \code{is.vector} ch(eck)
#' @export
chVector <- chP(is.vector)

#' \code{is.factor} ch(eck)
#' @export
chFactor <- chP(is.factor)

#' \code{is.data.frame} ch(eck)
#' @export
chDF <- chP(is.data.frame)

#' \code{data.table::is.data.table} ch(eck)
#' @export
chDT <- chP(data.table::is.data.table)

#' Returns a check for the data.table having exactly
#' n rows
```

```
#' @export
chDTn <- function(n) {
  chNatInt(n)
  chAnd(chDT, chP(function(dt) nrow(dt) == n))
}

#' Returns a check for the data.table having exactly
#' 0 or n rows
#' @export
chDT0n <- function(n) {
  chNatInt(n)
  chAnd(chDT, chP(function(dt) {
    nr <- nrow(dt)
    nr == 0L || nr == n
  }))
}

#' Ch(eck) for a single-row data.table
#' @export
chDT1 <- NULL

#' Ch(eck) for an empty or single-row data.table
#' @export
chDT01 <- NULL

#' \code{ggplot2::is.ggplot} ch(eck)
#' @export
chGgplot <- chP(ggplot2::is.ggplot)

#' \code{tibble::is.tibble} ch(eck)
#' @export
chTibble <- chP(tibble::is.tibble)

#' \code{is.array} ch(eck)
#' @export
chArray <- chP(is.array)

#' \code{is.atomic} ch(eck)
#' @export
chAtomic <- chP(is.atomic)

#' \code{is.recursive} ch(eck)
#' @export
chRecursive <- chP(is.recursive)
```

```
#' \code{is.object} ch(eck)
#' @export
chObject <- chP(is.object)

#' \code{is.matrix} ch(eck)
#' @export
chMatrix <- chP(is.matrix)

#' \code{is.table} ch(eck)
#' @export
chTable <- chP(is.table)

#' \code{is.environment} ch(eck)
#' @export
chEnv <- chP(is.environment)

#' \code{is.call} ch(eck)
#' @export
chCall <- chP(is.call)

#' \code{is.expression} ch(eck)
#' @export
chExpr <- chP(is.expression)

#' \code{is.symbol} ch(eck)
#' @export
chSymbol <- chP(is.symbol)

#' \code{s == ""} ch(eck) for String,
#' deliberately not chReg-ed
#' @export
chEmptyString <- chAnd(chString, chP(function(s) s == ""))

#' \code{s != ""} ch(eck) for String,
#' deliberately not chReg-ed
#' @export
chNonEmptyString <- chNot(chEmptyString)

#' Blank-ness ch(eck) for String,
#' deliberately not chReg-ed
#' @export
chBlank <- chAnd(chString, chP(function(s)
  is.na(readr::parse_character(s))))

#' Non blank-ness ch(eck) for String,
```

```
#' deliberately not chReg-ed
#' @export
chNonBlank <- chNot(chBlank)

#' \code{lubridate::is.Date} ch(eck)
#' @export
chDates <- chP(lubridate::is.Date)

#' \code{chScalar} & \code{chDates} ch(eck)
#' @export
chDate  <- chAnd(chScalar, chDates)
chReg(chDate)
```