# The Functional Programming Language R and the Paradigm of Dynamic Scientific Programming (Position Paper)

Baltasar Trancón y Widemann[1], Carl Friedrich Bolz[2], and Clemens Grelck[3]

[1] Ecological Modelling and Computer Science, Universität Bayreuth, Germany
`Baltasar.Trancon@uni-bayreuth.de`

[2] Software Engineering and Programming Languages, Heinrich-Heine-Universität Düsseldorf, Germany

[3] Computer Systems Architecture, Universiteit van Amsterdam, Netherlands

**Abstract.** R is an environment and functional programming language for statistical data analysis and visualization. Largely unknown to the functional programming community, it is popular and influential in many empirical sciences. Due to its integrated combination of dynamic and reflective scripting on one hand, and array-based numerical computation on the other, R poses unique and challenging implementation problems, at odds with the conservative language technology employed by its developers. We discuss the background of R in historical context, highlight some of the more problematic language features, and discuss the potential for the effective use of state-of-the-art language technology in a future, safe and efficient implementation.

## 1 Introduction

It seems ironic that the functional programming language that is likely to be the only one of its kind for decades featured in the New York Times [26], to account for the most computing hours in scientific labs all over the world, and to have the largest group of academic users outside computer science proper, has a relationship to the functional programming community at large that is best summarized as mutual ignorance. The R system and language [23] is a dynamic functional environment for data analysis and visualization and is widely considered the de-facto standard platform for development of statistical algorithms.

In this paper, we argue that R is a showcase application where functional programming can really shine and be brought to the awareness of a vast scientific and industrial community. We explain why the use of R has outgrown its design, and why more intense contact with the functional programming community could be beneficial to guide future development (sections 2 and 3). We outline which issues in the implementation of R defeat traditional compilation and optimization schemes (section 4), and how modern language technology could be used in novel and interesting ways to tame the beast (section 5).

The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-642-40447-4_12`.

## 2 A Brief History of R

The R system is a free open-source reimplementation of the S system, with minor changes in the language. S was conceived from the late 1970s on by the statistician Chambers [10–13] from Bell Labs and won the 1998 ACM Software System Award [28]. The design of the S system can be understood by analogy to two other systems that were created at approximately the same time: the Emacs text editor and the TEX typesetting system.

The three systems have a variety of properties in common: They provide a framework for automation of complex tasks on top of some basic functionality; text file editing in the case of Emacs, typesetting in the case of TEX, and statistical algorithms given as FORTRAN routines in the case of S. The chosen method of automation is explicit high-level programming rather than pre-packaged scripts collected in menus of a graphical user interface, the dominant theme of rival products such as early WYSIWYG word processors, and statistical packages such as SAS [2] and SPSS [21]. The conscious choice for a programming approach to extensibility has been made in each case because the authors foresaw the active role of users in the development process. Consequently, self-documentation and packaging play major roles in the coding styles.

The respective programming languages S/R, TEX and ELisp are more or less declarative, extremely dynamic and mildly domain-specific, and emphasize user-friendliness in terms of safe memory management and robust error handling. Software engineering concerns, in particular provisions for programming in the large such as formalized interface descriptions and means for encapsulation, are not reflected in the basic designs. The disregard for software engineering by the designers does not imply that the systems are ill-designed. To the contrary, the quickness and ease with which they could be employed for practical tasks has contributed significantly to their broad success. But, because of that success, nowadays all of the systems have by far exceeded the lifetime, variety of platforms, number of contributors and complexity of programming layers their inventors could possibly have foreseen, with literally thousands of contributed packages available from central repositories. It is therefore no surprise that issues of versioning, scalability, packaging and feature interaction have become a continual nuisance to the developer communities.

S ran on the obscure Honeywell GCOS platform and on top of FORTRAN routines at first, but major technologies such as Unix and C had been adopted before development activity ceased around 1990. R is a fairly faithful reimplementation of the C flavour of S, begun in 1993 by Ihaka and Gentleman from the University of Auckland, adding aspects of the GNU philosophy to the system, most notably “copyleft” licensing and an open-source community-driven development process. It is administered by the non-profit R Foundation. From the theoretical point of view, the only significant change in the language is the switch from dynamic to lexical scoping of variables [15], although the concept is somewhat compromised (see section 4.2 below). While R was the “illegitimate” rival of the officially licensed commercial S offspring S-PLUS at first, it has been endorsed by Chambers by his joining the development core team.

## 3 Using R

The basic R system supports both interaction via a command-line interpreter loop, and offline processing via numerous batch commands and options. Diverse window-based user interfaces and integrated environments for all major platforms exist. Graphical output in high quality and a variety of formats, both interactively and file-based, is supported by basic libraries. The R interpreter can also be embedded in applications written in many mainstream languages.

The functionality of R comprises a base library of statistical and I/O code, and a huge user-contributed repository [25] of currently about 5 000 packages for virtually all conceivable sorts of data analysis. R packages typically come with extensive documentation of function signatures and usage examples. Documentation quality varies; especially structured data objects returned by complex analyses (containing, for instance, the parameters, fitted values, residuals and goodness-of-fit scores of a statistical model) are often under-specified and require a fair deal of trial-and-error and reverse-engineering of encodings to be practically usable. Not one but two concepts of object-orientation exist in R to address these issues, but they are used inconsistently, and the expressiveness of the type system for object attributes is rather poor: For instance, the only homogeneous aggregate types are arrays, of a fixed small set of base types, with unspecified dimension; lists are always heterogeneous, while both primitive scalar types and proper records are nonexistent.

The current state and interaction history of a running R system can be stored and retrieved, in order to make sessions persistent and accountable. R code units come in two flavours: *scripts* containing arbitrary sequences of commands, and *packages* containing definitions in a certain namespace. Note that there is no special declaration level of the R language: a definition is merely the assignment of an anonymous function expression to a variable. Consequently, local definitions, let-bindings and function literals have the same expressive power as global definitions.

The function abstraction mechanism of R is very powerful indeed, featuring named and positional parameters, lexical scoping, variadic functions, implicit laziness, and mutually recursive default values for every lambda term (see section 4.3 below). While this mechanism is adequate for complex and highly customizable interface functions of packages, the interpretative overhead for a massively recursive programming style is prohibitively high, easily reaching three orders of magnitude relative to efficient compiled code. Functional programming idioms in R include a limited range of higher-order functions, most notably a family of *map/reduce* operations on multi-dimensional arrays, unhelpfully all called **apply** with an optional initial consonant. Additionally there is a tendency to avoid FORTRAN-style loops in favour of point-free operations (see section 4.4 below). Besides concerns for elegance, the main reason is that the R interpreter has no special notion of index variables, increment operations and single-point array access. Consequently, the interpretative overhead of treating them as dynamic degenerate cases of high-level operations is enormous with respect to the use of dedicated machine registers and instructions in compiled code.

From the academic viewpoint, R is a valuable addition to any curriculum in empirical sciences. The language is easy and fun to teach and to learn, and particularly suitable for students without a computer science background. Even though advanced functional programming skills are rare among R programmers, the fundamentals are well supported by the language and the corpus of example programs. Everything from ad-hoc data exploration and manipulation to publication-quality statistic analyses and diagrams can be achieved in the form of a short R script, often incrementally derived from freely available code. R scripts are also a useful, executable complement to term papers and theses in data-oriented courses. Program source code and results can be spliced into documents by means of the Knuth-style literate programming tool `SWeave`.

## 4 Under the Hood of R

The R system is a classical command-line interpreter with a core written in C. A major part of the higher-level functionality is written in the R language itself. Code is represented for interpretation and meta-programming purposes as an abstract syntax tree. An experimental bytecode format is supported since version 2.13, released in 2011. Native code is not generated, but foreign function interfaces for various languages such as FORTRAN, C and C++ exist. Many computationally expensive tasks are currently solved using these, because pure R is known to be quite slow and unsuited for tight loops or deep recursion.

Memory management is automatic by a generational garbage collector with additional reference counting for arrays, enabling the dynamic scheduling of destructive updates that is absolutely necessary for efficient, referentially transparent array programming. The system is extremely reflective: code objects, environments and the call stack can be inspected and manipulated freely at runtime. Persistence is a major topic: array data in various formats, command histories and complete snapshots of system state can be read and written. Graphical output, both online and in diverse file formats, is supported natively as well.

The type system is completely dynamic and rather complicated due to the multitude of historically accumulated, incongruent layers. Some type information, such as syntactic categories and array types, is encoded as magic numbers in memory cell headers. Array base types include three-valued logicals, integers, real and complex floating point numbers and strings (but not single characters). Two kinds of generic lists, with array- and pair-based structure, respectively, exist. Other datatypes are characterized by particular values of predefined metadata attributes, most notably **class**: two generic rivalling object/class mechanisms, domain-specific types such as *factors* (arrays of finite enumerated base type, optionally ordered), *data frames* (rectangular tables with heterogeneous columns) and *time series*, etc. The type system is further confounded by a collection of four dynamic type identification functions, namely **typeof**, **mode**, **storage**.**mode** and **class** with redundant but subtly different results.

Atomic data such as a single integer or truth value are not supported directly, but only in boxed form as singleton arrays; even in places where nothing but a single value makes sense, such as if-then-else conditions. Despite the ubiquity of function parameters that are expected to contain a single number or truth value switch, neither documentation nor semantics are consistent throughout the R base system: Attempts to pass vectors of length other than unity as such parameters variously cause the additional elements to be ignored silently, or one of a variety of warnings and errors to be raised.

## 4.1 Fundamental Pragmatism

R inherits from S the fundamental design focus on immediate practical domain-specific usability. Chambers describes the position retrospectively [9]:

> *There was also interest in different approaches or theories of computing, and much more so in later versions. However, there seemed always to be an unquestioned assumption that the essential criterion was a system that people would use and in particular one that provided the techniques considered essential. Much of the early discussion was therefore about which techniques we most needed to supply, and how to do it.* [. . . ]
>
> *From a general computing view, the philosophy tries to combine aspects of functional and object-oriented (i.e., method-centered) approaches. But as in previous stages of the evolution of S, adherence to a formal approach tended to be compromised when it conflicted with what we saw users as needing.*

A formally trained programming language expert might contend that mere interest in theory is not nearly enough to guarantee any benefits from novel language aspects, and that the pragmatic blessing may well become a semantic curse, when interferences of compromised aspects get out of control. As a first hint at the kind of problems to be expected, consider the following quote from the official R language definition [23], emphasis added:

> *Whether attributes* ***should*** *be copied when an object is altered is a* ***complex*** *area, but there are some* ***general*** *rules* [. . . ]*: Scalar functions (those which operate element-by-element on a vector and whose output is* ***similar*** *to the input)* ***should*** *preserve attributes (except* ***perhaps*** ***class****). Binary operations* ***normally*** *copy* ***most*** *attributes from the longer argument (and if they are of the same length from both,* ***preferring*** *the values on the first). Here 'most' means all except the* ***names****,* ***dim*** *and* ***dimnames*** *which are set* ***appropriately*** *by the code for the operator.* [. . . ]

We have highlighted words that are unexpected in a language definition, which is the definitive source of semantics after all.

## 4.2 Variable Scoping and Evaluation Strategy

Somewhat bizarrely, R's data model is *almost* purely functional: even large arrays are persistent, and efficient updates rely on reference-counting to detect singleton references and switch to destructive updates dynamically and transparently. The only mutable data structures, however, are those that have the greatest impact on semantics, namely *environments*. Variable bindings and even references to parent environments can be overwritten ad libitum, and bizarre applications have been found and posted on R mailing lists. Thus variable scoping in R has, possibly uniquely, the properties of being *lexical* and generally *lazy* but *late* and *not referentially transparent*. The following example illustrates the peculiar effects.

```
foo ← function(x = y + 1) {y ← 2; x}
```

This statement binds the variable foo to a function of a formal parameter x with a default expression. The default, to be used whenever no actual parameter is given, is a lazy promise or closure, to be evaluated in the environment of the function body. The function body binds the variable y, then returns x. Evaluating the pair of statements

```
y ← 1; c(foo(), y)
```

in the same environment where foo has been defined, assuming **c** has its predefined meaning as the free array constructor, yields the array $(3, 1)$. The assignment to y in the function body is local and has no effect on the outer binding, but shadows it before the promise bound to x is forced. If the function body is changed to {x; y ← 2; x}, then the result becomes $(2, 1)$, because the promise is forced earlier and the reference to y falls through to the outer binding.

In theory, the mutable environments of R can make static analysis arbitrarily hard. The fact has been realized by R developers, and used to rationalize poor performance under the motto "too flexible to be fast" [22]. But fortunately, and quite understandably, destructive updates to environments are used only in very controlled ways in practice. The apparently dominant usage pattern in typical, reasonable user code is repeated assignment to the same local variable, as in loop counter increments or array updates. We propose that a clever combination of static analysis and dynamic guards should be able to wring enough meaning from variable bindings to improve both performance and safety significantly.

The following two example topics illustrate costly dynamic problems faced by an execution engine for R. Cheaper, wholly or partly static solutions rely on estimates of the call graph, and hence on some knowledge about the bindings of *function* variables, since there are no static function calls in R.

### 4.3 Function Call Rules

The rules depicted in Fig. 1 are a succinct paraphrase of the specification of function application in the R language definition. Since the details are quite complex, the rules depicted in Fig. 2 summarize the well-formed parametrizations from the caller's perspective.

The parameter-matching steps 2–9 are required for all function calls, except when the function expression evaluates to either of two kinds of builtins: for *primitive* functions the parameters are evaluated eagerly, sidestepping the

1. In an expression of the form A(B) the function part A is evaluated first.
   - It is an error if the result is neither a builtin nor a lambda abstraction.
2. From B a list of **actual parameters** is formed: pairs of *name* literal (optional, followed by = if present) and *value* expression (optional).
3. From the function head a list of **formal parameters** is formed: pairs of *name* literal (mandatory) and *default* expression (optional, preceded by = if present).
4. Named actual parameters are matched with their eponymous formal counterparts.
5. Remaining named actual parameters are matched with formal parameters if the name extends uniquely.
   - It is an error if an actual name is not exact and has no unique formal extension.
6. Unnamed actual parameters are matched to yet unmatched formal parameters in textual order.
7. Valueless formal parameters (unmatched or matched by an actual parameter without value) are matched with their defaults.
   - Valueless formal parameters without defaults are matched implicitly with *missing* value expressions.
8. Remaining actual parameters are matched with the formal catch-all parameter ‘ ... ’, which may occur at any position and declares the function variadic.
9. An environment with the formal parameters bound to *promises* of the matching expressions is created.
   - Its parent is the lexical environment of the function definition.
   - Missing values are implemented as promises of an error.
10. The function body is evaluated in the environment.
    - Evaluating a reference to a formal parameters forces the matching promise, except for a few special primitive operations (**substitute**).
    - Assignments within the function body modify the environment.
11. The environment is discarded.
12. The value of the last body statement becomes the function result.

**Fig. 1:** Function evaluation, operational rules

†1. A function call is well-formed if each formal parameter has a corresponding actual parameter, explicitly named and in the same textual order.

†2. From a call well-formed by †1, an actual parameter name may be omitted, except if its formal counterpart is declared after ‘ ... ’.

†3. From a call well-formed by †1–2, a pair of adjacent actual parameters may be transposed, if one of the pair is named.

†4. From a call well-formed by †1–4,
   (a) an actual parameter name may be shortened to a unique prefix,
   (b) an actual parameter value may be omitted, if the formal parameter has a default, is not used, or explicitly allows omission,
   (c) an actual parameter may be omitted altogether, if the formal parameter has a default and no unnamed actual parameters follow,
   (d) a named actual parameter may be added at any position, if the name does not match any formal parameter, not even as a prefix, and the function is variadic,
   (e) an unnamed actual parameter may be added at any position, if all formal parameters are matched and the function is variadic.

**Fig. 2:** Function parameter matching, well-formedness rules

promise mechanism, and passed in textual order; whereas for *special forms* the parameters are passed unevaluated in textual order.

With the information readily available to the R interpreter, the function to be called is predictable only in rare cases, namely for function literals (explicitly named primitive or lambda term). Function variables, on the other hand, can be bound to functions of any kind and signature. Hence matching of parameters and even the well-formedness of a call are conceptually dynamic problems.

Formal parameter names in a lambda abstraction double in the roles of local variables and actual parameter keywords. Since the caller is entirely free to choose whether to address a parameter by position or by name, alpha equivalence does not hold for any R function expressions.

Missing parameter values (no default and no matching actual parameter or value omitted) are implemented as promises that raise an error when, and only when, forced. Hence it is not generally an error to call a function with too few parameter values, but only if the current function call attempts to use the parameter in a forcing way. Non-forcing uses such as **substitute** (see below) or the **missing** check do not raise errors.

The operation **substitute** is part of the R reflection toolkit. It is able to extract the actual syntax tree and environment from a promise. It is used in some R analyses to record queries in the result objects, and in visualizations to generate titles and axis labels automatically. Apart from these apparently innocuous uses, however, it has the semantically unpleasant property of violating the Church–Rosser principle: reduction of function arguments does not commute with their substitution into the function body. As a consequence, all program optimization by expansion or reduction, most notably common subexpression elimination and partial evaluation, respectively, in nested expressions (every operator is a function call too) becomes unsound and requires nontrivial safeguards in the presence of reflection.

## 4.4 High-Level Array Operations

The R language, in marked contrast to low-level languages such as C and FORTRAN, and more stringently than its more imperative competitor MATLAB, supports and encourages a point-free style of array programming. Again, the design is extremely pragmatic. Rather than providing an explicit, semantically well-understood basis of higher-order operations such as *map* and *reduce*, or even APL hieroglyphs, individual standard operations are *vectorized*: their global behaviour on arrays is defined in terms of local rules for individual elements, in an ad-hoc fashion governed by the most frequent usage patterns. Hence the resulting coding style is often operationally imprecise, but it can be read and written very effectively and with little room for nontrivial errors.

The following code fragment, excerpted from a package developed by one of the authors, illustrates some of the typical techniques.

```
x ← x[x != 0L]
f ← function (p) −sum(p * log(p, base = 2))
f(x / sum(x))
```

Here the binary entropy of an integer array x, assumed to contain absolute frequencies of a population of items, is computed.

The first line removes zero entries from the array: x is compared with an array containing a single (integer) zero. Like most binary operations, inequality is vectorized to act simultaneously pointwise on its arguments (*zip* in standard functional programming jargon). The arguments do not have the same length. This causes the shorter, second argument to be *recycled*, resulting effectively in an array of zeroes of the same length as x. The result of the comparison is an array of just as many truth values, with TRUE in positions where x is nonzero. Indexing x with this array is vectorized to act as a *filter*, retaining only the elements flagged as TRUE.

The second line binds the familiar entropy formula to the variable f, except that the probability distribution p need not subscripted with a loop variable, by virtue of vectorization.

The third line scales down from absolute to relative frequencies. The division operator is again vectorized to *zip* its arguments, where the latter is a single integer and recycled accordingly. Ordinary division also implies coercion from integers to floating-point numbers. The function bound to f is applied to the resulting array. It inherits vectorization from its constituent operations: Logarithm acts as *map* on its first argument, multiplication as *zip*. Unary minus acts trivially vectorized, negating the single element of the array resulting from **sum**.

Even though point-free style is used to great effect for the human reader and writer, its potential for optimized execution is not currently leveraged in R. Several intermediate arrays are created by this expression, namely as the results of the operations **!=**, [ ], **log**, $*$ and /, respectively, as well as degenerate arrays that box a single number each, namely the results of the literal 0L and the operations **sum** (bis) and $-$. Neither loop-fusion techniques that would reduce the amount of intermediate data, nor parallelization of *map*- or *zip*-vectorized operations can be applied in the default R system, because each of the involved operations could be redefined dynamically, altering the algebraic and vectorization properties that underly such optimizations.

## 5 Technological Suggestions

We suggest a three-pronged strategy to make the execution of R programs more efficient. Firstly, standard optimization techniques require a lot of information that is not declared explicitly in R and hence needs to be inferred, preferably by static analysis. The usual suspects are: type and shape analysis for arrays to elide runtime type and bound checks and coercions, strictness analysis to support the unboxing of function parameters, and static binding analysis to enable algebraic laws and inlining of known functions.

Secondly, we address the strict isolation of low-level, statically compiled core functionality on the one hand, and inefficient, dynamically interpreted scripting on the other hand, at the granularity level of individual function bodies and with no exchange of information except function calls proper. We feel it needs to be lifted in favour of an integrated, flexible, code generator-centric approach. This probably requires a radical departure from the existing implementation, since the distinction is deeply ingrained in the employed methodologies and technologies.

No obvious candidate for a backend platform exists that supports the dynamic and number crunching aspects of R equally. But fortunately, even though their unification in the R *language* is most desirable from the user and software engineering perspectives, the separation of scripting and numerics layer in R *programs* appears feasible. Hence we propose the liberal solution to target not one but two platforms, specializing on either aspect, and to guide the choice and/or combination by user preference and analysis results.

## 5.1 Static Analysis and Beyond

In a language where programs are short and run typically for a long time on huge amounts of data, static analysis and transformation easily pays off. Unfortunately, virtually all ahead-of-time transformation of R code is either impossible or semantically unsafe, unless strong assumptions about the program can be made and verified: The dynamic, intransparent behaviour of environments makes prediction of variable values theoretically difficult, and foils optimizations such as constant propagation and algebraic simplifications. The powerful reflection mechanism allows the behaviour of code to depend on its literal form, and foils optimizations such as common subexpression elimination, function inlining, lambda lifting and substitutive optimizations in general, most notably specializations relying on "the trick" of partial evaluation. The lack of explicit type and shape information for arrays poses the same problems regarding loop organization and bounds checking as in other dynamic languages.

The last, type-related category of issues can also be handled with local speculative techniques such as *quickening*, even when retaining an interpreter [8], and does not necessarily call for static analysis. But the former two involve long-range dependencies that are not easily dealt with, unless it can be assumed that the problematic features are not actually used. An empirical survey [20] (see section 6.1 below) of a corpus of R code indicates that the majority of practical program fragments is actually reasonably safe. Hence we expect that a static approximation, even if a little coarse, should be able to leverage many well-understood and effective optimization techniques. It may even be worthwhile to investigate whether potentially unsafe transformations actually break a given package, relying on R regression testing for heuristic validation.

For the sake of modularity, as well as to allow the user to stand in where static analysis fails, a general annotation format for static information in R code would be useful to communicate properties at interfaces. In this aspect, the reflective power of R can be turned to advantage: Program terms are ordinary data structures with an extensive query and manipulation interface, and arbitrary structured metadata can be attached to any data via attributes. Hence annotations can be added simply by establishing a metadata format, with no need for any invasive extensions of the language proper.

It remains to see whether users of R can be convinced to use the mechanism and add declarations to their programs. We expect that ideological arguments from semantic theory or software engineering methodology would not be well-received. But practical annotation-based tools, for instance providing diagnosis of inconsistencies, automatic instrumentation with assertions, automatic documentation of interfaces and/or automatic generation of test cases, might well gain some recognition in the community and increase awareness for the issues.

## 5.2 Dynamic Compilation

A recent line of techniques that have been successfully used for implementing other complex dynamic languages without too much effort are *tracing* JIT compilers [14]. Those are JIT compilers that take as their compilation unit not a function but a commonly executed code path (trace) within the program. Quite often these traces correspond to loops within the original program, ending with a jump to their beginning.

Tracing JITs have been used successfully for dynamic imperative programming languages [14], but also some first experiments for declarative programming languages have been done, for example for Prolog [7].

The traces in a tracing JIT are formed by observing the execution paths through the program as it executes after some profiling. Thus all the traces correspond to control flow paths that have been taken a few times already.

This approach has many advantages. On the one hand it makes many components of the JIT compiler much simpler to write. Both the optimizers [4] and the backends can be very simple, because they only need to deal with linear pieces of code. This also allows the optimizer to perform aggressive type specialization and optimize away the potential for dynamic behaviour that is not used in the current code path. This can lead to very efficient machine code which removes most of the overhead of dynamic typing.

Dynamic compilation and trace compilation in particular is extremely effective in optimizing the overhead of dynamic typing on the local level. However, due to the limited scope of compilation no global information can be exploited. This is why we think that a combination with a static analysis phase which feeds back some globally established properties of the program into the runtime compiler could improve the efficiency of dynamic compilation even more.

Most tracing JIT compilers have been written for one specific VM and thus for one specific language. A few projects have emerged that tried to write reusable tracing JIT compilers. This approach is called "meta-tracing", because the tracers do not operate on the level of the program, but on the level of the interpreter for the program. Examples are the SPUR project [3] and the PyPy/RPython project [6].

RPython [1] ("Restricted Python", there is no relationship to R) was developed for the PyPy project [24]. It is a programming language designed for implementing interpreters for dynamic programming languages. The RPython subset of Python is restricted in such a way as to make compilation of the interpreter to C possible. The interpreter written in RPython can thus be translated into a VM in C. During this process, various aspects of the final VM are woven into the interpreter. Examples for this are garbage collection and a tracing JIT compiler [6].

This weaving of low-level aspects into the final VM means that the language implementation in RPython stays independent of low-level details. The meta-tracing JIT is one such orthogonal aspect. It is woven into the final VM, guided by a few hints that the interpreter author inserts into the interpreter [5].

We plan to utilize the RPython framework for the implementation of our R system, specifically the execution of R code. We feel that a meta-approach to tracing JIT construction is the only sensible way to efficiently implement a language as complex as R efficiently as a whole. We hope to be able to integrate with this jitting implementation the results from our static analyses to make the JIT generate even better code.

### 5.3 High-Performance Functional Backends

It hardly requires visionary power to understand that fairly small compute-intensive numerical kernels, as shown in Section 4.4, are the kind of R code that dominates the execution times of entire programs while at the same time the performance difference between R and low-level compiled languages, say C or Fortran, is the greatest. Consequently, such definitions also provide the most attractive opportunities to speed up the execution of R code, potentially by orders of magnitude. The on-going trend in commodity hardware towards multi-core designs and the proliferation of many-core graphics accelerators in the mass market are both a blessing and a curse for languages like R: a curse as they will widen the performance gap between R and more low-level approaches; a potential blessing if R could implicitly utilise parallel computing power without the notorious hassle incurred by low-level parallel programming.

Compilation of R array kernels into efficient code, not to mention decent support for a variety of multi- and many-core architectures, is a major research and engineering challenge. Therefore, it is attractive not to go all the way from R down to C or Fortran, but to leverage a language whose design is somewhat half way between R and C as an intermediate compilation target and to leverage existing compilation technology.

Such a language is SaC (Single Assignment C, [17]). SaC is a compiled array programming language with a syntax that, as the name suggests, resembles that of C proper, but that at the same time comes with a purely functional semantics. SaC features stateless multi-dimensional arrays similar to APL or MatLab, call-by-value parameter passing for arrays, automatic memory management, etc. Such features of SaC considerably reduce the semantic gap in compiling R.

We illustrate this in Fig. 3 by means of a SaC code fragment implementing the example introduced in Section 4.4. Apart from the C-style syntax of function definitions and applications, fragments of the code are almost identical to R. Some issues are minor: for example, prior to the division in function `g` we need to explicitly convert integers into floating point numbers. Other issues are more relevant: SaC is monomorphic with respect to scalar types. A big plus on

```
double[*] log2( double[*] p)
{
  return { iv -> p[iv] == 0.0 ? 0.0 : log2(p[iv]) };
}

double f( double[*] p)
{
  return -sum( p * log2(p));
}

double g( int[*] x)
{
  return f( tod(x) / tod(sum(x)));
}
```

**Fig. 3:** Computing entropy in SaC

the other side is SaC's support for rank-generic programming: the type `int[*]` refers to integer arrays of any number of dimensions and any extent along these dimensions. One aspect we cannot directly mimic from the R code is the elimination of zero elements in the argument array. Apart from reducing the size of the array and, thus, reducing the computational effort of subsequent computations, the main reason is to avoid computing the logarithm of zero. We achieve at least the latter by overloading the `log2` function for arrays such that it yields zero in case, which does not affect the rest of the computation.

Despite a fairly similar programming style, the semantics of SaC code is much more stringently defined than that of R. This supports a highly optimising compiler technology that achieves competitive runtime performance for compute-intensive applications [27] and fully compiler-directed parallelisation for multi-core multi-processor systems [16] and for CUDA-enabled graphics cards [18].

## 6 Related Work

### 6.1 Evaluation of the Design of R

In a paper to appear at ECOOP 2012, Morandat et. al. [20] comprehensively analyze the properties of the R language as well, albeit from an object-oriented rather than functional perspective, reaching similar conclusion as this paper. They offer several interesting contributions, among them a formal semantics for some core parts of the language as well as a careful analysis of which features are used *in practice* by R programmers. The latter is done by collecting, running and analyzing a large corpus of R programs. One of the results of running benchmarks of R programs compared to equivalent programs written in Python is that R is significantly slower even than other dynamic programming languages like Python. Another result of the corpus analysis of R programs is that the older of the two competing R object systems is still much more widely used.

### 6.2 The Ra Extension

Ra [19] is an extension to the R system that interprets a proprietary bytecode format rather than R syntax trees. The bytecode is produced by a companion JIT library that specializes on arithmetic loops, using runtime type and shape information for arrays, and thus eliminating many dynamic type case distinctions, allocation and variable lookup operations. The label “JIT compiler” on the approach is not consistent with the conventional use of the term for dynamic languages, because no machine code is produced at runtime.

Unfortunately, there are severe additional caveats: Firstly, function calls and local control flow other than **for** loops are not processed in any way. Secondly, the transformation is not semantically conservative; more dynamic uses of array variables than supported by the compilation scheme, even such as resizing, will not just revert to the slow original, but raise an error. Lastly, the loop-oriented programming pattern that is sped up by Ra is deprecated in R in favour of vectorized operations anyway. In summary, despite good ideas, Ra cannot be considered a significantly more efficient implementation of R.

## 7 Conclusion

Scientific computing is nowadays more than just number crunching. The increasing *logical* complexity of data processing methods, as opposed to their *computational* complexity, requires other approaches than the extreme high-performance low-elegance style offered by the classics, most notoriously FORTRAN.

Scientists have reacted by adding high-level scripting layers, written in dynamic languages such as Python, to their software. A unified approach such as the one offered by R has numerous advantages:

Scientists, often without proper formal training in programming skills, are required to master only one language and one environment that apply to all levels of their data processing software. The R language is, semantic peculiarities aside, well-documented and forgiving, and encourages abstract, elegant and reusable functional style. The R environment is highly interactive and comes with extremely powerful tools for data exploration and visualization.

The functional structure of R programs makes them theoretically amenable to automatic optimization and parallelization of array processing code, thus compensating for a significant part of the efficiency loss with respect to low-level high-performance code, without sacrificing the gains in flexibility, both with respect to application parameters and hardware capabilities, and ease of development and maintenance.

On the practical side, the existing R system has deficiencies with respect to diagnostics and performance of pure R code. The problem is currently being side-stepped by R developers by recommending the use of foreign function interfaces for performance-critical code fragments, thus compromising many benefits of the high-level approach, most notably datatype dynamicity, platform independence, memory safety and interactive responsiveness.

The concentration of the dynamic and numeric perspectives in a single language is a unique and challenging situation. We believe that the traditional language technology underlying the current R implementation is fundamentally ill-suited for the problem, but that modern technologies developed for other dynamic languages on the one hand, and for other functional languages on the other hand, can be used to great effect. The seemingly active hostility of R towards optimization should be understood as a motivating challenge to compiler constructors, and may well serve as a realistic test case to evaluate the techniques applied to it. R has already gained considerable recognition from academic users for being *usable* and *flexible*. If *fast* and *safe* could be added to the list, it would make a very convincing case for functional programming.

## Acknowledgments

A research proposal for an optimizing implementation of R using the techniques outlined in this article is currently under consideration for funding by the *Deutsche Forschungsgemeinschaft* under application number LE 2377/2-1.

## References

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. “RPython: a step towards reconciling dynamically and statically typed OO languages”. In: *DLS*. Montreal, Quebec, Canada: ACM, 2007. ISBN: 978-1-59593-868-8. DOI: 10.1145/1297081.1297091.

[2] A. J. Barr and J. H. Goodnight. *SAS, Statistical Analysis System*. North Carolina State University. 1971.

[3] M. Bebenita et al. “SPUR: a trace-based JIT compiler for CIL”. In: *OOPSLA*. Reno/Tahoe, Nevada, USA: ACM, 2010. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869517.

[4] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. “Allocation removal by partial evaluation in a tracing JIT”. In: *Partial Evaluation and Program Manipulation (PEPM 2011)*. Ed. by S.-C. Khoo and J. G. Siek. Austin, Texas, USA: ACM, 2011. ISBN: 978-1-4503-0485-6.

[5] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, A. Rigo, and S. Pedroni. “Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages”. In: *ICOOOLPS*. Lancaster, UK: ACM, 2011.

[6] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. “Tracing the meta-level: PyPy’s tracing JIT compiler”. In: *ICOOOLPS*. Genova, Italy: ACM, 2009, pp. 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827.

[7] C. F. Bolz, M. Leuschel, and D. Schneider. “Towards a Jitting VM for Prolog execution”. In: *PPDP*. Hagenberg, Austria: ACM, 2010. ISBN: 978-1-4503-0132-9. DOI: 10.1145/1836089.1836102.

[8] S. Brunthaler. “Efficient interpretation using quickening”. In: *SIGPLAN Not.* 45.12 (Oct. 2010), pp. 1–14. ISSN: 0362-1340. DOI: 10.1145/1899661.1869633. URL: http://doi.acm.org/10.1145/1899661.1869633.

[9] J. M. Chambers. *Stages in the Evolution of S*. Bell Labs. 2000. URL: http://cm.bell-labs.com/cm/ms/departments/sia/S/history.html.

[10] J. M. Chambers and R. A. Becker. *S: An Interactive Environment for Data Analysis and Graphics*. Wadsworth & Brooks/Cole, 1984. ISBN: 0-534-03313-X.

[11] J. M. Chambers and R. A. Becker. *Extending the S System*. Wadsworth & Brooks/Cole, 1985. ISBN: 0-534-05016-6.

[12] J. M. Chambers and R. A. Becker. *The New S Language: A Programming Environment for Data Analysis and Graphics*. Wadsworth & Brooks/Cole, 1988. ISBN: 0-534-09192-X.

[13] J. M. Chambers and T. Hastie. *Statistical Models in S*. Wadsworth & Brooks/-Cole, 1991. ISBN: 0-412-05291-1.

[14] A. Gal et al. "Trace-based just-in-time type specialization for dynamic languages". In: PLDI '09. ACM ID: 1542528. Dublin, Ireland: ACM, 2009. ISBN: 978-1-60558-392-1. DOI: `10.1145/1542476.1542528`.

[15] R. Gentleman and R. Ihaka. "Lexical Scope and Statistical Computing". In: *Journal of Computational and Graphical Statistics* 9 (2000), pp. 491–508.

[16] C. Grelck. "Shared memory multiprocessor support for functional array processing in SAC". In: *Journal of Functional Programming* 15.3 (2005), pp. 353–401.

[17] C. Grelck and S.-B. Scholz. "SAC: A Functional Array Language for Efficient Multithreaded Execution". In: *International Journal of Parallel Programming* 34.4 (2006), pp. 383–427.

[18] J. Guo, J. Thiyagalingam, and S.-B. Scholz. "Breaking the GPU programming barrier with the auto-parallelising SAC compiler". In: *Annual Symposium on Principles of Programming Languages, 6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, Texas, USA*. ACM Press, 2011, pp. 15–24.

[19] S. Milborrow. *The Ra Extension to R*. 2011. URL: `http://www.milbo.users.sonic.net/ra/`.

[20] F. Morandat, B. Hill, L. Osvald, and J. Vitek. "Evaluating the Design of the R Language". In: ECOOP'12. to appear. 2012.

[21] N. Nie, D. H. Bent, and C. H. Hull. *SPSS: statistical package for the social sciences*. McGraw–Hill, 1970. ISBN: 0-070-46530-4.

[22] R-bloggers. *You can scrap it and write something better but let me keep R ;)* 2011. URL: `http://www.r-bloggers.com/you-can-scrap-it-and-write-something-better-but-let-me-keep-r/` (visited on 08/21/2011).

[23] *R Language Definition*. R Development Core Team. 2010. ISBN: 3-900051-13-5.

[24] A. Rigo and S. Pedroni. "PyPy's approach to virtual machine construction". In: *DLS*. Portland, Oregon, USA: ACM, 2006. ISBN: 1-59593-491-X. DOI: `10.1145/1176617.1176753`.

[25] *The Comprehensive R Archive Network*. URL: `http://cran.r-project.org`.

[26] A. Vance. "Data Analysts Captivated by R's Power". In: *New York Times* (Jan. 9, 2009): *Business Computing*.

[27] V. Wieser, C. Grelck, P. Haslinger, J. Guo, et al. "Combining High Productivity and High Performance in Image Processing Using Single Assignment C on Multi-core CPUs and Many-core GPUs". In: *J. Electronic Imaging* (to appear).

[28] A. Wilson. "ACM honors Dr. John M. Chambers of Bell Labs with the 1998 ACM Software System Award for creating 'S System' software". In: *ACM Press Release* (Mar. 1999).