A Fast Abstract Syntax Tree Interpreter for R

Tomas Kalibera [†]Petr Maj [‡]Floreal Morandat Jan Vitek

Purdue University [†]ReactorLabs [‡]University of Bordeaux

Abstract

Dynamic languages have been gaining popularity to the point that their performance is starting to matter. The effort required to develop a production-quality, high-performance runtime is, however, staggering and the expertise required to do so is often out of reach of the community maintaining a particular language. Many domain specific languages remain stuck with naive implementations, as they are easy to write and simple to maintain for domain scientists. In this paper, we try to see how far one can push a naive implementation while remaining portable and not requiring expertise in compilers and runtime systems. We choose the R language, a dynamic language used in statistics, as the target of our experiment and adopt the simplest possible implementation strategy, one based on evaluation of abstract syntax trees. We build our interpreter on top of a Java virtual machine and use only facilities available to all Java programmers. We compare our results to other implementations of R.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—interpreters, optimization; G.3 [*Probability and Statistics*]: statistical computing

Keywords R language; specialization; lazy evaluation

1. Introduction

Dynamic languages are gaining in popularity in many areas of science. Octave and R are perfect examples of widely adopted domain specific languages that were developed by scientists, chemical engineers and statisticians respectively. They are appealing because of their extensive libraries and support for exploratory programming. Yet, both are painfully slow and memory hungry; R programs can run hundreds of times slower than equivalent C code [10]. Inefficiencies sometime force end-users to rewrite their applications in more performant languages. This is clearly undesirable and could be mitigated by better language implementations. Unfortunately, as is often the case for community-supported languages, domain scientists lack the manpower to build a high-performance runtime and often also the skills to do so. Their expertise lies elsewhere, chemistry or statistics, they are language implementers by necessity, not by choice. Also, even if a high-performance runtime were to be handed to them, maintenance would likely prove to be a stumbling block.

We explore how far one can push simple implementation techniques – techniques that are portable and leverage widely deployed technologies – to obtain a performing language implementation. We aim to show that a relatively performant interpreter and runtime system can be obtained without requiring deep knowledge of compiler techniques. For concreteness, we have chosen to implement a subset of the R language and to restrict ourselves to an abstract syntax tree (AST) interpreter built on top of an off-the-shelf Java virtual machine (JVM). Choosing Java as an implementation language simplifies maintenance as it is type safe and provides a high-quality runtime that includes a garbage collector and a threading system. Relying on the JVM gives us portability across all supported architectures and operating systems, as well as some basic security guarantees. Of course this comes at a cost, writing an interpreter in a managed language is likely to be less efficient than in C as we only have limited access to memory and pay for Java's runtime safety checks. Furthermore, implementing R data types on top of Java objects can lead to less than optimal memory usage. These costs and benefits have to be balanced when evaluating the viability of the approach.

R was designed by Ihaka and Gentleman [8] based on the S language [2]. GNU-R is maintained by a core group of statisticians and is available under the GPL license [12]. R is extensible and widely extended, currently there are nearly 6,000 packages available from the CRAN[1] and Bioconductor[2] repositories. R is heavily used for data analysis, visualization, data mining, or machine learning in fields including biology, environmental research, economics and marketing. R has an estimated 2 million installed base [13].

VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2764-0/14/03…$15.00.
http://dx.doi.org/10.1145/2576195.2576205

[1] `http://cran.r-project.org` [2] `http://www.bioconductor.org`

For its first fifteen years or so, R was implemented as an AST interpreter. This was likely due to the fact that an AST interpreter is simple to write, portable, easy to maintain. In 2011, Luke Tierney added a bytecode interpreter to improve performance. For compatibility reasons, the AST interpreter was retained and users can switch between the two engines freely. Both interpreters (we will refer to them as GNUR-AST and GNUR-BC) are written in C. We will use the AST as our baseline for performance comparisons. With R programs that spend mostly time out of numerical libraries, bytecode is about 2x faster than the AST. For context, we also consider Renjin, a rewrite of the GNUR-AST in Java. Renjin is roughly 2x slower than the GNUR-AST. So, if speedups compose, one could expect that a Java AST interpreter should be roughly 4x slower than a hand-tuned C bytecode interpreter.

This paper introduces FastR v0.168, an AST interpreter for the R language written in Java and capable of running on any off-the-shelf JVM. FastR leverages the ANTLR parser generator and the Java runtime for garbage collection and runtime code generation; native code is invoked via the Java Native Interface. What makes the implementation stand out is the extensive use of runtime feedback to perform AST-specialization [17], data specialization, as well as data and code co-specialization. These techniques enable the program to optimize itself by in-place rewriting of AST-nodes. To this we added a number of interpreter optimization tricks. The result is an interpreter that runs roughly **5x** faster than GNUR-BC and about **8x** faster than GNUR-AST. The remainder of the paper will describe our implementation techniques and argue that they remain simple enough to be maintained and extended by domain experts.

2. The R language and its implementation

R is a dynamically-typed, lazy functional language with limited side-effects and support for computational reflection. The authoritative description of R is its source code [12].

Data types. R has few primitive data types, namely, `raw` (unsigned byte), `logical` (three-valued booleans), `integer` (signed 32-bit), `double`, `complex`, and `string`. Missing observations are denoted by `NA`. For integers, `NA` is the smallest integer representable, for doubles, one of the IEEE NaNs is used, and for logicals, `NA` is a special value. Integer overflow results in an `NA`. All values are vectors of zero or more data points of a base type. Values can have *attributes*, which are lists of name-value pairs that can be manipulated programmatically. Built-in attributes define dimensions, names, classes, etc. Operations are vectorized; they perform type conversions when values are not of the same type, and for vectors of different lengths, re-use elements of shorter vectors. R has several other data types including lists – polymorphic vectors capable of holding values of different types – closures, built-ins, language objects, and environments.

Functions. Functions are first class. They nest and have read/write access to their lexical environment. R also supports dynamic scoping with a global environment and package environments. Copy semantics provides the illusion of referential transparency; each assignment semantically creates a deep copy of the original value. An exception is environments, which are passed by reference. Functions may have formal arguments with default expressions. Arguments are matched by name, by unique prefix match on name, and as a last resort, by position. Arguments are evaluated lazily, using *promises*. Default expressions are packed in promises that evaluate in the called function while accessing data from the caller scope. Promises cache their results to avoid wasting computational resources and performing any included side-effects multiple times.

Meta-programming. Environments can be created synthetically and attached to the current variable search path or to closures. Reflection allows to change variable values, add new variables, or even remove variables from any environment unless the variables are locked. Code can be stored as a "language object", passed around and evaluated using `eval` in any environment. Language objects can also be created dynamically, e.g. by parsing a string.

Environment. R runs in a read-eval-print loop. R has about 700 built-in functions. R supports calling into native code, particularly with focus on C and Fortran. It interfaces with the BLAS and LAPACK numerical libraries and includes a modified version of LINPACK.

Interpreter. GNU-R is implemented in C and Fortran. GNU-R parses source code and generates an AST represented by lists of language objects. Evaluation follows the AST structure. Most nodes translate to function calls. Special functions, such as assignment, loops, or even braces, are dispatched to native C code that also evaluates argument expressions. Calls to closures are dispatched to evaluation of their bodies, with argument expressions packed into promises. The interpreter keeps a stack of execution contexts for stack traces in case of errors and for control flow operations such as loop break, continue, function return, or exception. Each execution context includes a target for C level non-local jump. Function environments are linked lists, where each element has a value slot and a symbol. Function environments are searched linearly whenever looking up a variable. Arguments are passed in heap-allocated lists of name/value pairs. Argument matching is done by reordering argument lists in three passes: for exact name matching, unique name prefix matching, and positional matching. Excessive copying is avoided through dynamic alias analysis (or bounded reference counting) — each value has a reference count 0 (temporary), 1 (bound to at most one variable), 2 (possibly bound to more than one variable). Vector update operations avoid copying when the reference count is at most 1. The reference count is never decremented and is not relevant for memory reclamation. GNU-R implements a non-moving generational mark-sweep collector with free-list based allocation.

3. Architecting a new R engine

FastR is a Java-based interpreter for a subset of the R language including most of the features of the language such as data types, functions, global environment, lazy evaluation, language objects, and eval. FastR is currently lacking support for packages and the different object systems implemented on top of R.

FastR is implemented in Java 7 and runs on any JVM. It relies on the ANTLR 3.4 to generate a Java parser. Mathematical operations are implemented by a mixture of Java and native code linked from GNU-R's library (mostly NMath and modified LINPACK), BLAS, LAPACK, and the system Math library. We use Javassist 3.18 to generate Java source code on the fly for operator fusion.

The current code base consists of 1358 classes, and 66KLoc (including comments and empty lines). The package structure is given in Figure 1; the majority of programming effort is split between data types (13KLoc), built-in functions (18KLoc) and basic operations (26KLoc in package `r.nodes.exec`). The ANTLR generated parser and lexer account for additional 10KLoc. In comparison, the core of GNU-R is 141KLoc of C code and the whole GNU-R code base goes up to 1.3MLoc.

R code can be entered at the console or input from a file. The parser creates a *parse tree* from source code. For execution, a more suitable kind of ASTs, called *executable trees*, is used. The conversion from parse trees to executable trees happens on demand, nodes lazily convert themselves whenever they are encountered during evaluation. Executable nodes retain a reference to their parse tree so as to produce user-friendly error messages. During execution, the interpreter continuously rewrites executable trees, selecting optimal implementations of operations based on runtime information available at each node. This form of *code specialization* is one of our key optimizations. FastR supports *data specialization* with multiple implementations of common data types optimized to take advantage of characteristics of the actual data. The other important optimization is the delayed evaluation of vector operations. FastR introduces *views* which perform *data and code co-specialization*. A view represents a vector operation that can be performed element-wise or in bulk and that can depend on other views, forming a *view tree*. Individual views in a view tree can also be fused together. Figure 2 summarizes the high-level architecture of the interpreter.

Package	# of files	# lines of code
r	8	1625
r.builtins	179	18715
r.data	20	4659
r.data.internal	28	8252
r.errors	2	1789
r.ext	3	110
r.ifc	1	172
r.nodes.ast	56	1794
r.nodes.exec	32	26113
r.nodes.tools	5	1880
r.runtime	7	1436

Figure 1. FastR package structure.

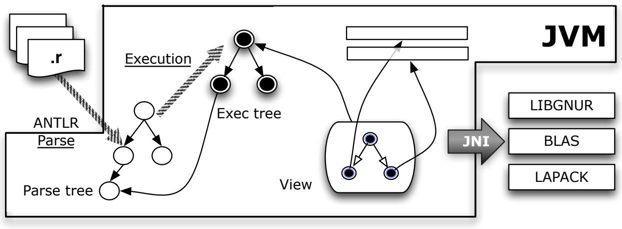

Figure 2. FastR Architecture. R code is turned into parse trees. As execution proceeds, parse trees are transformed into executable trees. Data is allocated in the Java heap and views are created to delay and bunch the execution of vector operations. Native code is accessed through JNI.

3.1 A traditional AST interpreter

FastR, at its core, has a naive abstract syntax tree interpreter that uses the native Java stack to store the execution state (a pointer to an executable node) and `Frame` objects to hold name-value bindings for arguments and local variables. Execution of a node boils down to calling that node's `execute()` method. Figure 3 shows an `If` node with children representing the condition and the true and false branches. The call to `executeScalarLogical` speculates that the condition evaluates to a scalar logical, as opposed to a vector or any other type, otherwise an exception is thrown and conversion or subseting is performed.

```
class If extends BaseR {
  RNode cond, tBr, fBr;
  Object execute(Frame frame) {
    int c;
    try {
      c = cond.executeScalarLogical(frame);
    } catch (SpecException e) { ... } //Recovery
    return (c==TRUE ? tBr:fBr).execute(frame);
} }
```

Figure 3. Sample executable node (simplified).

3.2 Data layout

Basic R data types are represented by Java objects in the most straightforward fashion. Classes `ComplexImpl`, `DoubleImpl`, `IntImpl`, `ListImpl`, `LogicalImpl`, `RawImpl`, `StringImpl` denote vectors of the corresponding base types. Each holds a Java array for the payload, an array of dimensions in case the vector is used as a matrix or n-dimensional matrix, a list of names, a set of attributes and an integer reference count. FastR has two other data types, `Closure` which bundles a `Frame` with the attached code as executable node and `Language` which represents source code using AST tree and can be passed to eval.

4. FastR optimizations

We detail the optimizations applied to our base interpreter.

4.1 Pre-processing

The pre-processing step that translates parse trees into executable trees is an opportunity to perform optimizations. After profiling of our workloads, we identified three transformations as being profitable: *return elision*, *loop unrolling* and *variable numbering*.

Return elision. In R, `return` is a one-argument call which terminates execution of the current function and returns the argument to the function's caller. FastR implements it using a Java exception — `return` throws a pre-allocated singleton and the return value is stored in the `Frame`. The semantics of R specifies that, absent an explicit `return`, a function yields the value of the last expression evaluated. Thus, semantically, any function that has, as its last statement, a `return`, is equivalent to the same function without it.

Trailing `return`s are elided by a simple control-flow analysis and rewriting to deal with control flow patterns such as:

```
function(...) {              function(...) {
  if (condition) {             if (condition) {
    ...                          ...
    return(value)      ⇒         value
  }                            } else {
  rest                           rest
}                            } }
```

As `return` can be redefined by the user, FastR monitors redefinitions and rewrites the executable tree to implement the semantics of the non-optimized form.

Loop unrolling. The code specialization optimization that we will describe next requires a certain amount of stability in the code executed. We have observed that many loops must be evaluated once before they stabilize, i.e. the type information required for specialization be apparent to the interpreter. Consider a loop that performs double arithmetic on an integer vector. The first iteration of the loop will cause coercion of the vector and the change of type from integer to double will confuse the specialization optimization. To avoid this, all loops are unrolled once by rewriting loop nodes to special nodes that have two bodies. The first loop body runs for the first loop iteration, while the second runs for the following iterations. We have observed significant performance improvements after adding this optimization to `for` loops.

Variable numbering. R has baroque variable declaration and lookup rules which allow new variables to be injected into an existing environment. Thus, a naive implementation would implement an environment as a hash map of symbols to values (or a list of pairs), and the lookup would have to check all lexically enclosing environments up to the global scope, making variable access expensive. FastR attempts to speed up the common case when local variables are looked up and reflection is not employed.

Environments use different `Frame` classes: `GenericFrame` has an `Object[]` to hold values and 5 specialized `Frame` classes for environments of 0 to 4 variables hold their values in fields, avoiding the need for indirection through the array. A `FrameDescriptor` holds the names of frame's variables. Descriptors are shared across multiple invocations of a function, while each invocation has its own frame. Additional variables can be kept with their names in a `FrameExtension`, which can be either linear (values searched sequentially by name), or hashed. A frame extension is private to an invocation of a function.

To avoid hashing on local variable lookup, a simple static analysis runs at parse tree translation time. We collect a *write set* W of variables written to in each function. Each variable in W is assigned a unique number. This number is the index of the variable's value in the frame. Accessing a variable by index side steps hashing. A reflective operation may introduce a new variable to a frame: if a variable does not occur in W, it will be installed in the frame extension. The new variable may also shadow a variable defined in an enclosing environment. If this happens, the corresponding frame is marked dirty.

Reads are slightly tricky. For a non-reflective read of a variable in W, the variable's index in the frame can be used. If the variable is not in W of the current function, but is in W of some lexically enclosing function, its index and the number of hops required to get from the current frame to the target are recorded during the static analysis and are used to speed up the lookup. When performing a non-local read, the dirty bit of the frame that has the variable in W must be checked. If a dirty frame is encountered, then the interpreter reverts to the slower hashed lookup. A read must also check that the variable has indeed been defined, because entries in W include all possible variable declarations, some of which may be in branches not taken or not-yet-taken.

4.2 Code specialization

Specialization is used pervasively. As a program executes, its executable tree is continuously updated to leverage runtime information. Execution of a program represented by an executable tree P (or AST in systems that execute it directly) and a program state S proceeds stepwise. Evaluation of a node updates both program and state:

$$P, S \hookrightarrow P', S'$$

In practice, we restrict ourselves to local modifications of executable trees. Assume a tree with nodes $n_0, n_1, n_2 \ldots$ such that $n_0 \rightarrow n_1 \rightarrow \{n_2 \ldots\}$ ($\rightarrow$ denote the parent relationship). If the current node is n_1, we allow:

1. Replace node: $n_0 \rightarrow \mathbf{n} \rightarrow \{n_2 \ldots\}$.

2. Replace child: $n_0 \rightarrow n_1 \rightarrow \{\mathbf{n}\ldots\}$.
3. Insert above: $n_0 \rightarrow \mathbf{n} \rightarrow n_1 \rightarrow \{n_2\ldots\}$.

Specializations are driven by runtime information with no inherent guarantee of soundness. Any rewriting may alter the meaning of the program to yield incorrect results when the node is encountered again. To prevent this, we perform *guarded* specialization. There are two kinds of guards: inline guards and watchpoints. Inline guards simply check that the predicate that was used to specialize a node holds when the node is evaluated again. Watchpoints can register node rewritings to be performed if a particular event occurs, such as redefinition of a symbol in the global scope. Inline guards can sometimes be replaced with watchpoints or elided. For instance, an `If` node is specialized on the assumption that the condition yields a scalar, non-`NA`, value; if the condition happens to be a call to, say, `is.na`, FastR infers that unless the built-in is redefined it will return either `TRUE` or `FALSE`. Thus, no guard is required, instead a watchpoint is registered for changes to `is.na`. If we further assume that changes to built-ins are disallowed, the check disappears entirely.

When a guard fails, a Java exception is thrown and the programmer must provide recovery code to rewrite the tree to a semantically meaningful version. Our implementation does not mandate convergence of rewritings. This can be the right behavior for code that processes different types at different stages in the program, but pathological cases are possible where a node keeps getting optimized only to be de-optimized right after. If this is a risk, programmers can choose to maintain a bound on rewriting any given node, which we do in FastR.

Dealing with recursion is tricky as a node may appear multiple times on the call stack. This can lead to an inconsistent tree rewrite. Consider the evaluation of $n \rightarrow \{\ldots n_i \ldots\}$ in function `f`. Rewriting n to n' can occur at any time, say after child n_i has been evaluated. If another instance of `f` is also in evaluating n, it may not realize that the node has been removed from the tree and it may try to perform a replacement itself. To prevent this, developers must check that a node is in the tree to execute it. When a node is kicked out of a tree, it retains a reference to its replacement node. Consider the `execute()` method of the `If` node. It checks if the current node has been replaced during evaluation of the condition node, and if so, proceeds with the replacement (Figure 4).

We have described the mechanics of specialization. But *what* and *when* should one specialize? The key for performance is choosing specializations with inexpensive guards and reducing the risk that hot code gets stuck in generic nodes. Careful profiling of workloads is necessary along with a deep understanding of the semantics of the language and its libraries. As to when to specialize, we sometimes create uninitialized nodes which have no other behavior than wait to see the shape of their arguments and immediately rewrite themselves. In other cases, we have generic nodes that implement the full, but slow, semantics of the operation, and rewrite themselves, after execution, to a more specific version. Lastly, some specializations are eager, speculating on likely values and properties of arguments.

```
Object execute(Frame f) {
  int c;
  try {
    c = cond.executeScalarLogical(f);
  } catch (SpecializationException e) {
    return (replaced() ? replacement : this).
          executeWithFailedCond(f, e.getResult());
  }
  return (replaced() ?
    replacement : this).executeWithCond(f, c);
}
```

Figure 4. The `If` node's execute method.

Arithmetic. The `Arithmetic` class is one of the largest with close to 8KLoc; it contains nested classes that implement the `Add`, `Sub`, `Div`, `Mod`, `Mult`, and `Pow` operations for numeric data types. Figure 5 shows an extract of this class. This executable node has two fields for children and fields that describe the specific operation to be performed by the node. The `execute` method first evaluates both children, checking, each time, that the current node was not replaced. Equipped with values of subexpressions, a specialized version of the node is constructed. The `SpecializedConst` class is used if one of the arguments was a constant expression. This class specializes the evaluation by hard-wiring a constant. The node is then replaced and the `execute` method of the specialized node is invoked to perform the actual computation. From that point on, the specialized node will be called directly.

```
class Arithmetic extends BaseR {
  RNode l, r; ValArit arit;

  Object execute(Frame f, ASTNode src) {
    Object lx = l.execute(f);
    if (replaced())
      return replacement.execute(f, lx, src);
    Object rx = r.execute(f);
    if (replaced())
      return replacement.execute(lx, rx, src);
    Arithmetic s = (l instanceof Constant
     || r instanceof Constant) ?
       SpecializedConst.mk(lx, rx, src, l, r, arit) :
       Specialized.mk(lx, rx, src, l, r, arit);
    return replace(s).execute(lx, rx, src);
} }
```

Figure 5. The `Arithmetic` node is the entry point for arithmetics.

Arithmetics specializes on scalar arguments with inline guards for particular combination of argument types such as: both are integers, left is double and right is integer, etc. Specialized nodes include necessary casts and `NA` checks. FastR has nodes for operations where argument types alternate between integer and double as this shows up in our workloads.

For vector arithmetics, specializations exist for common shapes, i.e. combinations of types and dimensions, e.g. when vectors are of the same length. Vector operations check the reference count of the arguments; if the reference count is zero then that vector is a temporary and can be reused for the result. Some arithmetic operations like `mod` and `pow` can be off-loaded to native code, as their Java implementation is notoriously slow. Data is passed as primitive array without copying.

Variable access. Variable access is a frequent operation with a number of specialized nodes. Pseudo-code for the uninitialized read node is in Figure 6. If the node executes in the global environment (the frame is null), it will always execute that way, and hence goes directly to the symbol table without checking dirty bits (`readOnlyTopLevel`). Otherwise, if the variable has a local slot (is in W), it will always be at the same index, the read unconditionally accesses the slot in the frame (`readLocalSmallFrame`) or in an array referenced from the frame (`readSimpleLocal`). Otherwise, the value has to be searched in the global environment, but checking the dirty bit to ensure it had not been inserted reflectively (`readTopLevel`).

We observe that variables in the global environment rarely get changed after initialized. Hence, if the uninitialized read node is able to find a value for the variable in the global environment and the symbol is clean, it creates a specialized node to always return this value (`readStableTopLevel`). If the value changes or is shadowed by a reflective write, a watchpoint will undo the rewrite.

Last, in the uninitialized read node, if there is no local slot for the variable, but there is one in a lexically enclosing function, that enclosing slot will always be there when this read node executes and it will be the closest one — `EnclosingIdx` contains the number of hops to a lexically enclosing frame and the index of the slot in that frame. The read node hence specializes to one that always checks that enclosing slot (`readEnclosing`) for the value and also the dirty bit of that enclosing frame. When a dirty frame or symbol is found, the read operation has to check extensions of all frames on the lexical stack, up to the one that has a fixed slot for the variable.

```
class UninitializedRead extends ReadVariable {
  Object execute(Frame f) {
    ReadVariable node;
    int idx; EnclosingIdx eidx;
    if (f == null)
      node = readOnlyTopLevel(ast, symbol);
    else if ((idx = f.findVariable(symbol)) != -1)
      node = f instanceof SmallFrame ?
         readLocalSmallFrame(ast, symbol, idx, f):
         readSimpleLocal(ast, symbol, idx);
    else if ((eidx = f.readSetEntry(symbol))==null) {
      node = readStableTopLevel(ast, symbol);
      if (node == null)
        node = readTopLevel(ast, symbol);
    } else
      node = readEnclosing(ast, symbol, eidx);
   return replace(node).execute(f);
} }
```

Figure 6. This class is the entry point for all variable reads.

`ReadSimpleLocal` (Figure 7) reads a variable from a known index of a frame. If the read value is a promise, it is forced. The node speculates that the variable has been initialized. If this assumption fails, the node rewrites to `readLocal`, which can handle the general case (looking up through enclosing slots, their dirty bits, possibly frame extensions and global environment). Generally, we observed that keeping performance critical nodes small helps the JIT compiler of the JVM generate faster code.

```
class ReadSimpleLocal extends ReadVariable {
  final int idx;
  Object execute(Frame f) {
    Object value = f.getForcing(idx);
    return value != null ? value :
      replace(readLocal(ast,symbol,idx)).execute(f);
} }
```

Figure 7. This class implements read of local variables by index.

Functions. Function calls are costly as function arguments can be passed in any order and, thus, they must be matched. However, we observe that programs almost always call the same function at a particular call-site. We therefore compute the matching once and re-use it over subsequent calls. The closures expect their arguments to be provided in declaration order. Each call must create a frame for the closure and populate it with arguments in the expected order. Figure 8 shows the generic call node; `argsPos` holds the mapping of argument positions.

We have observed that it is common for functions to be defined in the global environment and not re-defined. Moreover, functions quite often take only a few arguments passed by position. Thus, specialized nodes exist for a positional closure call through a global environment for 0, 1, 2 and 3 arguments. Except a watchpoint for re-definition, this specialization requires no guards.

Built-ins are slightly different. A *builtin* is a function that, unlike a closure, does not have its own environment. A custom instance of an executable node is created for each builtin call, specializing for a particular set of arguments. The specialization covers values of literals, e.g. `log(base=10, x)` will use a node specialized for log10. Builtin call sites directly refer to the body of the built-in and they are protected by a watchpoint which will rewrite them if needed.

4.3 Data specialization

Data types are represented using Java interfaces, such as `RInt`, as to allow data specialization. There is a most general implementation, e.g. `IntImpl`, which holds multiple values, names, dimensions, attributes and a reference count. For numeric types values are held in primitive arrays so that they can be passed to native code without conversions or copying. Figure 9 shows an interface to a double vector. Mutating operations, such as `set` and `setAttributes`, may return a new value of the whole data type (i.e. possibly copying a vector). We support immutable values, which always return a copy

```
class GenericCall extends FunctionCall {
  RClosure lastClosure;
  int[] argsPos; RFunction fun;
  RBuiltin lastBuiltin; RNode bnode;

  Object execute(Frame f) {
    Object callable = callableExpr.execute(f);
    if (callable == lastClosure) {
      Frame nf = fun.newFrame(lastClosure.frame());
      placeArgs(callerFrame, fun, argsPos);
      return fun.call(nf);
    } else if (callable == lastBuiltin) {
      return bnode.execute(f);
    } else {
      //if closure, set lastClosure, argsPos, fun,
      //if builtin, set lastBuiltin, bnode; and call
} } }
```

Figure 8. This class implements generic function/builtin call.

of themselves on an update. The `materialize` method performs the conversion to the general representation. Method `getContent` returns a primitive array in the standard format, which, may require materialization. Method `ref` increments the internal reference count, `isTemporary` means `ref` has not been called, `isShared` means `ref` was called more than once. For an immutable representation, there is no reference count and the value is always shared.

Simple scalars. Scalars with no names, dimensions, or attributes have an immutable representation with a single field to hold the payload, e.g. a double for `ScalarDoubleImpl`. Literals are represented as simple scalars, and results of computations that can be represented by simple scalars are automatically converted to those. This speeds up code by removing the indirection required by the primitive array. Also, checking that a value is a simple scalar is just a simple type check, making guards in specialized code cheaper.

Simple range. A sequence of integers, written in R as `1:n`, is immutable and has no attributes. The upper bound is remembered instead of creating a primitive array (`RIntSimpleRange`). Simple ranges often appear in `for` loops and vector indexing. The data specialization allows code specialization of such loops and vector indexing through a type check for `RIntSimpleRange`.

```
interface RDouble extends RAny {
  int size();
  Attributes attributes();
  double getDouble(int i);
  double[] getContent();
  double sum(boolean narm);
  RDouble materialize();
  RDouble set(int i, double val);
  RDouble setAttributes(Attributes attributes);
  void ref();
  boolean isShared();
  boolean isTemporary();
}
```

Figure 9. This interface represents a double vector (simplified).

Integer sequence. An integer sequence has an arbitrary starting point, arbitrary (possibly negative) step, and a size (`RIntSequence`). Integer sequences result from R expressions like `m:n` and from calls to `seq`. FastR has immutable specialization for these sequences, saving memory and allowing code specialization of loops and vector indexing.

4.4 Code and data co-specialization

Our last optimization intertwines code and data specialization by turning code into data. More precisely, rather than performing operations on vectors, FastR can defer them constructing expression trees (or *views*). Views are first class values, transparent to the users, which can provide both speed and memory improvements by defining a different evaluation order for the vector operations.

Views should not be mistaken with the lazy semantics introduced with promises. First, promises are rather eager [10], any assignment or sequencing operation will cause them to get evaluated. Second, promises are exposed to the programmer as they are used for meta-programming.

A view is an implementation of one of the R data types that contains a mixture of data objects and unevaluated operations. It supports materialization to compute the entire result, individual reads to obtain subsets of the result, and selected aggregate operations such as `sum` to perform computation on the result without materializing it. Views are stateless trees of arbitrary size. Views can cache their results and still appear stateless to the rest of the interpreter.

Views are built incrementally as the program performs vector operations. The actual computation is deferred. The operations allowed in a view include arithmetic and logical operations (`+, /, %/%, %%, *, -, !, &, &&, ...`), casts (int to double,...), builtin functions (`ceiling, floor, ln, log10, log2, sqrt, abs, round, exp, Im, log, Re, rev, is.na, tolower, toupper`), vector index (`x[1:10], x[[1]], x["str"], x[-1], ...`). The key criterion for inclusion is operations whose behavior is solely defined by the value of their inputs, where computation of individual vector elements is independent, and that do not have observable side-effects.[2] Views form trees that have values at their leaves. R semantics (and FastR's dynamic alias analysis) ensure that these values will not be modified after the view is created.

In R expressions, operations on vectors can include vectors of different lengths (shorter vectors are reused) and types (conversions are applied). The size and type of a view thus depends on its leaves and on the operations applied to them. FastR uses a view only when determining the result type and size is cheap. This is true for arithmetics, but not for many vector indexing modes, such as indexing with a logical index. Consider the following R program, on the left, and the view it creates, on the right:

[2] Warning messages are one exception, views report warnings retrospectively and in a different order from the original computation.

```
add1 <- function(a) a+1

x <- 1:1000
y <- add1( x )
z <- y + x
```

```
"+"(
  "+"(
    "cast"(Seq(1000)),
    1
  ),
  "cast"(Seq(1000))
)
```

Executing the above code will leave `z` pointing to the view on the right, with no actual computation performed. The benefits of views come into play when we try to access the result, for instance `print(z)`. In this example, we can avoid allocating temporary vectors for `x` and `y`. No temporary is needed and, if `print` accesses `z` element-by-element, there will even be no allocation for the result vector: if the user requested only one value, e.g. `print(z[[1]])`, then no allocation would be needed and the value would be computed directly for the first element only.

Deferred element-wise computation is not always going to help. For very small vectors avoiding temporaries will not make a big impact and the overhead of interpreting the tree will not be amortized. Moreover, views can lead to redundant computation due to element vector re-use (repeatedly computing data points of a shorter vector) or simply by reusing the view in the R program for multiple computations. An extreme example of a redundant computation would be a recursive view, e.g. a computation such as `x = x * x + c` performed in a loop.

Specialization is used to heuristically find when it pays off to compute using a view. Every executable node that performs an operation supported by views will return a profiling view (*PView*) when first executed. PViews record the number of calls to methods of the view that access individual elements (`get`), access all values (`materialize`) and call to aggregators (e.g. `sum`), as well as other statistics about the views such as size and type. A PView also installs a listener on assignment of a view to detect recursion. The profile filled in by a PView is attached to the executable node which created it. When this node executes next, it will rewrite itself based on the profile, either to a node that always creates a view, or to a node that always materializes its inputs and performs the computation on them eagerly. The heuristic of Figure 10 decides when to be eager. There could be adversary programs which will not work well with it, such as when multiple views are created by a node before the first one is used, but the view is not recursive. Or, when the size of a vector created by a particular node will significantly increase during execution. Or, when a PView is passed to a very expensive computation.

Even eager computation can be optimized, for instance when an operation has a temporary input (ref count 0) of the same type as the result, it can be re-used. Performing operations one vector at a time can be easily off-loaded: we have obtained big speed-ups for `pow` and FP `mod` by evaluating them in native code as these functions are notoriously slow in Java. Some operations can be off-loaded even if they are in a view

```
boolean shouldBeLazy() {
  if (isRecursive || size == 0) return false;
  if (noAccesses) return true;
  if (size < 20) return false;
  if (externalMaterializeCount > 0 || externalGetCount
      > size || internalGetCount > size) return false;
  return true;
}
```

Figure 10. Heuristic choice of lazy/eager arithmetic.

that is being materialized. When materializing, one needs first to obtain a vector for the result; this is done by allocating a fresh vector. This vector can be used to store temporary results and allow some views of a view tree to be materialized eagerly, possibly off-loading the operations. FastR implements a heuristic when it performs materialization of a view using eager computation, whenever possible without allocating an extra buffer.

In addition to avoiding temporaries, view trees encode a simple program with semantics far simpler than Java or R. This program is easier to optimize. E.g., a view tree that is externally only used from its root can be automatically *fused* into a single view, which implements all operations of the view tree. FastR implements fusion using Java bytecode generation and dynamic loading. As we have observed with our benchmarks, even without explicit fusion, the JIT of the underlying JVM can often devirtualize the `get` call from a parent view to its child and inline it, essentially doing the fusion. This is, however, unlikely to work well in complicated programs.

4.5 Implementation complexity

The optimizations we describe have a cost in code complexity and code bloat. This cost is hard to measure, as the optimizations are not encapsulated like e.g. phases in a compiler. Instead, they are mostly rules and tricks for how to write the interpreter code, and they make the code harder to understand and bigger compared to a hypothetical naive AST interpreter. Our design choice was to isolate complex code and reduce code bloat with standard Java features (polymorphism, etc) rather than using additional tools e.g. for code generation or meta-programming.

The *code complexity* is increased by self-optimization. The AST tree has to be copied into the executable tree. Each executable node has to be written so that it can be safely removed from the tree, and even so while an instance of it is executing. After executing each child, a node has to check if it has been replaced, possibly continuing execution in the replacement node. This replacement logic has to be implemented specifically for each node and each of its children. The replacement code (catch blocks of `SpecializationException`) is about 3% Loc of the code base excluding the generated parser. There is an additional overhead in `execute` methods related to checking if a node has been replaced and some overhead with maintaining replacement nodes. While the replacement logic is complex, it does not add many lines of code, because it is shared by executable nodes of a similar kind (e.g. arithmetics, logical operations, comparison, unary Math function, etc). Also, the replacement code is expected to run rarely, so it does not have to be fast.

Code specialization by definition increases the code size. For example, the `ReadVector` class implements nodes for indexing a vector (2KLoc), but 70% of this code is for specialized cases, such as that the index is a scalar integer within bounds or the index is a logical vector of the same length as the base vector. Implementing these cases is no harder than implementing the general case any AST interpreter will need to have. Still, they pose substantial code bloat: over two thirds of the specialized cases in `ReadVector` require thought and in their present form could not be generated. This increases the amount of maintenance work needed, but not the set of skills to do the maintenance.

Less than a third of specialized cases in `ReadVector` are copy-pasted with mundane edits, e.g. the implementation of a subset of a double vector using an integer sequence is essentially the same as of an integer vector, but has to be implemented as a distinct class. Similar types of mundane bloat appear in the whole code base. Such bloat is due to the choice of Java, not because of our optimizations. In C/C++, one could avoid it using templates, macros, multiple inheritance, and unsafe casts.

5. Related Work

Our work was inspired by Truffle [17], a framework for writing AST interpreters under development at Oracle Labs. It encourages programming in a self-optimizing style. Future version of FastR will use it and the companion optimizing JIT compiler called Graal. Our approach to code specialization is related to techniques developed for Self [3] and trace-based compilation [6]. Earlier work on program specialization, e.g. [4], used the term *data specialization* to mean memoization of expensive computations, where we actually specialize the data types themselves. In the terminology of [9], we perform code specialization that is both dynamic and optimistic. The idea of data specialization can be traced back all the way to the notion of *drag-along* in APL implementations [1].

The GNU R bytecode interpreter [15] translates AST to bytecode on-the-fly at the granularity of a generalized call (e.g. a function or a loop). The only optimizations performed by the compiler are constant folding, inlining, and tail call optimizations. Many non-local jumps can be replaced by local transfers of control. No specialization is performed. The bytecode compiler optimizes local variable look-up by caching them into a constant-indexed array, but unlike FastR, variables in enclosing environments are not optimized. GNU-R can be extended with RcppArmadillo which is a lazy vector Math library [5]. But unlike FastR, lazy vector operations have to be extracted manually by programmers from their R code.

Renjin[3], like FastR, is an AST interpreter for a subset of R running on a JVM. Unlike FastR, it mirrors the implementation of GNUR-AST, including internal data structures. Renjin implements lazy computation similar to views. Certain views can be cached, parallelized and sometime fused, but arithmetic vector operations are not parallelized and are prone to repeated computation. Work on compiling simple basic blocks into Java bytecode is ongoing.

Riposte [14] implements a subset of R. It defers vector operations, producing a trace with operations on typed vector futures analog to our views. Traces are limited to operate on vectors of the same length, but the operations may originate from different expressions in the R source code. Traces are compiled to 64-bit Intel machine code with optimizations that include vector instructions, algebraic simplification, constant folding, and common subexpression elimination. Traces can be fused and parallelized. To avoid redundant computation, Riposte computes and caches all futures reachable from R variables at the time the trace is compiled and executed. This may lead to unnecessary computation of futures. Scalar code is not optimized.

pqR[4] is a modified version of GNU-R 2.15.0. It adds some data specialization, e.g. for integer sequences. It replaces the bounded reference counts by full reference counting with decrements to reduce the need of copying on vector update. It offloads some numerical computations to helper threads, running them asynchronously.

6. Performance

We compare performance of FastR against the GNUR-AST to evaluate the impact of all optimizations implemented to date. We also compare against GNUR-BC with its highest optimization level, as it is the best performing official implementation of R. We are targeting longer-running programs (benchmarks are dimensioned to run 1-2 minutes with GNUR-BC). There are fixed costs of starting up a JVM, JIT compilation, and FastR self-optimization. We include the whole execution into the measurement, including these start-up costs.

6.1 Benchmarks

We run all benchmarks from the Benchmark 2.5 suite [16] (b25) and the Language Shootout benchmarks [7] (shootout).

Benchmark 2.5. The *b25 benchmarks*[5] comprise of three groups, each with five micro-benchmarks: matrix calculation (`mc`), matrix functions (`mf`) and programming (`pr`). The workloads are summarized in Figure 11. Most but not all of the workloads include a trivial amount of R code and R interpretation only takes a small fraction of their execution

[3] `http://www.renjin.org`

[4] `http://radfordneal.github.io/pqR`

[5] Sometimes referred to as AT&T R Benchmarks.

`mc1`	Double square matrix transposition
`mc2`	Power function, double vector over scalar
`mc3`	Quicksort on a double vector
`mc4`	Cross-product of a double square matrix ($A^T \times A$)
`mc5`	QR decomposition of a double square matrix
`mf1`	Fast Fourier transformation of a double matrix
`mf2`	Eigenvalues of a double matrix
`mf3`	Determinant of a double matrix
`mf4`	Cholesky decomposition, cross-product of a double matrix
`mf5`	Solve equations via QR decomposition
`pr1`	Power function scalar over double vector, arithmetics
`pr2`	Square integer matrix transpose, arithmetics
`pr3`	Grand common divisors, vectorized
`pr4`	Toeplitz matrix (nested loops with scalars)
`pr5`	Escoufier's method (vectors, loops, function calls)

Figure 11. Benchmark 2.5.

time. Much time is spent in numerical algorithms in native code. The benchmarks use BLAS and LAPACK numerical libraries. Most of the benchmarks use random number generators implemented in native code (NMath part of GNU-R). One of the benchmarks uses the native `fft`. FastR uses the same code through JNI. Some numerical algorithms are harder to re-use and were re-implemented in Java following the original C implementation, an example is the estimation of correlation coefficients. Two of the benchmarks use matrix transposition. We implemented a blocked version of a square matrix transposition in Java that is faster than the simple implementation in GNU-R. We also specialized the random number generators for batch generation from one distribution with identical parameters because calling through JNI for every single number was too slow. The most R intensive workloads are `pr5` and `pr3`; `pr4` also spends all time in R, but is quite simple. Workloads `mc2`, `pr1` and `pr2` are based on arithmetics on long vectors, so they are still somewhat affected by the R interpreter design. None of the original benchmarks outputs any results, which makes validation of the computation hard. Worse yet, FastR with lazy computation can run some of them without actually doing most of the computation. We modified the benchmarks so that they aggregate their result and print it. This forces computation and provides a value to check.

Shootouts. The shootouts are R implementations of problems from the Computer Language Benchmarks Game [7]. The R implementation was written by a computer science student and optimized for speed on GNUR-BC. The programs are small applications, they produce an output to check, and stress different parts of an R implementation. Unlike b25, most of the shootouts are dominated by execution in the interpreter. The exceptions are sn6 (spends a lot of time in BLAS) and `rd` (dominated by regular expressions matching library). For each problem, the R version of the shootout suite [11] includes a "main" implementation and several alternative implementations. We run all variants unmodified, except for a performance irrelevant fix needed to run `kn3` in R. We also replace the use of internal calls to the GNU-R interpreter by more standard equivalents (`kn1-4`).

binarytrees	Allocates and traverses binary trees
`bt`	*GC benchmark, recursive calls, recursive lists*
pfannkuchred	Solves a combinatorial problem
`pr`	*Loops, indexing short vectors*
fasta	Generates DNA sequence by copying, rand. selection
`fa`	*String operations, scalar arithmetic*
fastaredux	Solves same problem as fasta
`fr`	*Adds more loops, vector indexing and arithmetic*
knucleotide	Finding patterns in gene sequences
`kn`	*Uses environment as a hashmap, string operations*
mandelbrot	Calculates a Mandelbrot set (fractal image)
`ma`	*Vector arithmetic on complex numbers*
nbody	Solves the N-body problem (simulation)
`nb`	*Arithmetic, Math with short vectors*
pidigits	Calculate digits of pi using spigot algorithm
`pd`	*Arbitrary precision arithmetic in R (diverse code)*
regexdna	Matching, replacing regex-specified gene sequences
`rd`	*Regular expressions (falls back to regex library)*
reversecompl	Computing reverse-complements for gene sequence
`rc`	*String vector indexing using string names*
spectralnorm	Computing eigenvalue using power method
`sn`	*Loops, function calls, scalar arithmetic*

Figure 12. Shootout.

The shootout problems are listed in Figure 12, including which aspect of an R interpreter does the main implementation stress. The alternative implementations always solve the same problem as the main one, but may stress different parts of an R interpreter from the main version. In particular, some of the alternative versions of `sn` and `fa` use more vector arithmetic than the main version. The biggest and most complicated program is `pd` (400 lines of code).

6.2 Measurement Methodology

We dimensioned benchmarks to run approximately 60 seconds with GNUR-BC on a development laptop and fit to 7G heap (`sn3` needed a shorter run to fit in memory). On the measurement platform, the benchmarks run mostly over a minute, some about 2 minutes. We dimensioned the shootouts via their size parameter (those that accept output from the `fa` benchmark via sizing `fa`). We dimensioned b25s by finding the best iteration count for the outer loop, hence the same computation is repeated, but on different data (these benchmarks generate random inputs and set the random seed prior to entering the outer iteration loop). For Renjin and the b25 experiments, however, we measure b25 with a smaller number of iterations (so that Renjin runs about a minute), and then scale the result to the iteration count of the other VMs. This was needed at least for `pr4` which has a two orders of magnitude slow-down, and we did it for all b25 benchmarks with Renjin.

We run each benchmark from either suite 10 times, for each VM, and then report a ratio of mean execution times of FastR, GNUR-BC, and Renjin each against GNUR-AST. We calculate a 95% confidence interval for the ratio of means using the percentile bootstrap method. For a quick summary, we also calculate the geometric mean of the ratios (that is the ratios of means) over each benchmark suite. The geometric mean has to be taken with a grain of salt, though, as performance changes tend to be dominated by several benchmarks from each suite (pr4 from b25 and kn from shootout). The overall speedup is thus a measure of how many of outlying benchmarks are in the suite. If the author chose to add say one more variant of kn to the shootouts, FastR speedups will likely be also great for it, and the geometric mean would increase a lot, as opposed to adding another variant of say rd. Similarly, if the author of b25 chose to add yet another benchmark that spends all time in LAPACK, the overall FastR speedup will drop. If that was another benchmark like pr4, it would increase. Still, the choice of geometric mean is more robust against the outlying benchmarks than would e.g. an arithmetic mean be. We intentionally do not calculate an error bar for the overall mean, as it would be giving a false level of confidence given the described issues.

6.3 Platform

We run our benchmarks on Dell PowerEdge R420 (2x Intel Xeon CPU 2.10GHz, 48G RAM) with 64-bit Ubuntu 12.04. We use GNU-R 3.0.2 compiled with default options by GCC. For GNUR-BC, we use the highest optimization level available (3). We run FastR on 64-bit Oracle JDK 1.8 (early access release), linking against system libraries including the R library. We run Renjin on the same JDK8, but compile it with JDK7 due to build problems with JDK8. We use the Oct-15-2013, github version of Renjin. We run all VMs with the Ubuntu version of openBLAS.

6.4 Results

The relative execution time of FastR and GNUR-BC normalized to GNUR-AST is shown in Figure 14 (b25) and 13 (shootout). The plots include error bars (95% confidence intervals), but the variation with most benchmarks is small.

Shootout benchmarks. On geometric average, FastR is 8.5x faster than GNUR-AST (while GNUR-BC is 1.8x faster than GNUR-AST). The biggest speed-ups are on the kn benchmarks. The benchmark uses a hash-map to look up gene sequences, and individual versions of kn differ in how they represent the hash-map (kn1 uses an environment, kn2 and kn3 use an attribute list, kn4 uses a named vector). FastR speed-ups come from optimization of these structures, e.g. a named vector in FastR remembers a hash-map of its names; this hash-map and the names are immutable, and hence can be propagated through operations with no cost. Attributes in FastR have a trie structure for fast matching of attributes based on their prefix. This trie is mutable, but can still be propagated through operations (the originating owner looses the trie and will have to re-build it in case matching becomes needed again). Most benchmarks then benefit from code and data specialization, and few from lazy vector computation. bt particularly benefits from function call optimizations and data/code specializations for scalars. pr particularly benefits from data specialization for integer sequences and from loop and vector indexing specializations. nb particularly benefits from optimized vector computation (not computing small vectors lazily). rc particularly benefits from vector indexing optimizations. sn benefits from vector and matrix indexing optimizations, loop optimizations, and lazy vector arithmetics. fa and particularly pd are relatively diverse and benefit from various optimizations.

Benchmark 2.5. On geometric average, FastR is 1.7x faster than GNUR-AST (and GNUR-BC is 1.1x faster than GNUR-AST). The relatively small speedup is explained by the preponderance of native calls. FastR offers biggest speed-up with the pr4 benchmark. This is due data/code specialization for scalars, including loops, matrix indexing, arithmetic and Math operations. pr3 speedups are thanks to specialization of vector operations (arithmetics, comparison, vector indexing using the result of a comparison). pr2 and mc1 benefit from optimized matrix transposition, pr2 also from lazy vector arithmetic. pr5 benefits from specialization for integer sequences and from code/data specializations of vector operations. Also, pr5 takes advantage of function call optimizations. The remaining speed-ups are mostly from specialization of the random number generator wrappers.

Renjin. Renjin performance results are shown in Figures 16 (b25) and 15 (shootout). We only show benchmarks that run; the other did not due to missing features. On geometric average over supported benchmarks, Renjin is 2.2x slower than GNUR-AST on b25 and 1.8x on the shootouts. The pr4 benchmark from b25 fills in a matrix of 500x500 elements in a loop. Semantically in R, any element update of a matrix creates a new matrix. Renjin copies the matrix in each iteration of the loop. GNUR-AST (and FastR) know through dynamic alias analysis that the matrix is private and avoid the copy, hence the 107x overhead of Renjin.

The 15x slowdown of Renjin on sn2 is because of redundant computation of a view. sn2 pre-computes two matrices (one using an outer product, another as the transpose of the first) and uses them read-only in a loop for computation. In Renjin, due to lazy computation, the matrices are not in fact pre-computed. Instead, each element is repeatedly recomputed on-the-fly. Renjin supports views that can cache the computed result, but they are not used for these operations. We do not have a definitive advice for Renjin on this. FastR implements profiling views to fight redundant computation, but they are based on a heuristic that does not work in the sn2 benchmark — redundant computation is avoided by coincidence (the particular computation of the initial matrices in sn2 happens to be always eager in FastR). GNUR does not run into these problems, because it always computes eagerly.

The nearly 5x slowdowns of Renjin on bt benchmarks (shootout) is due to the return statement (the benchmark is dominated by calls to a cheap recursive function which uses a return statement). In Renjin, every single call to return allocates a new Java exception and throws it. GNUR-AST, instead, uses a C non-local jump, which is much cheaper. In

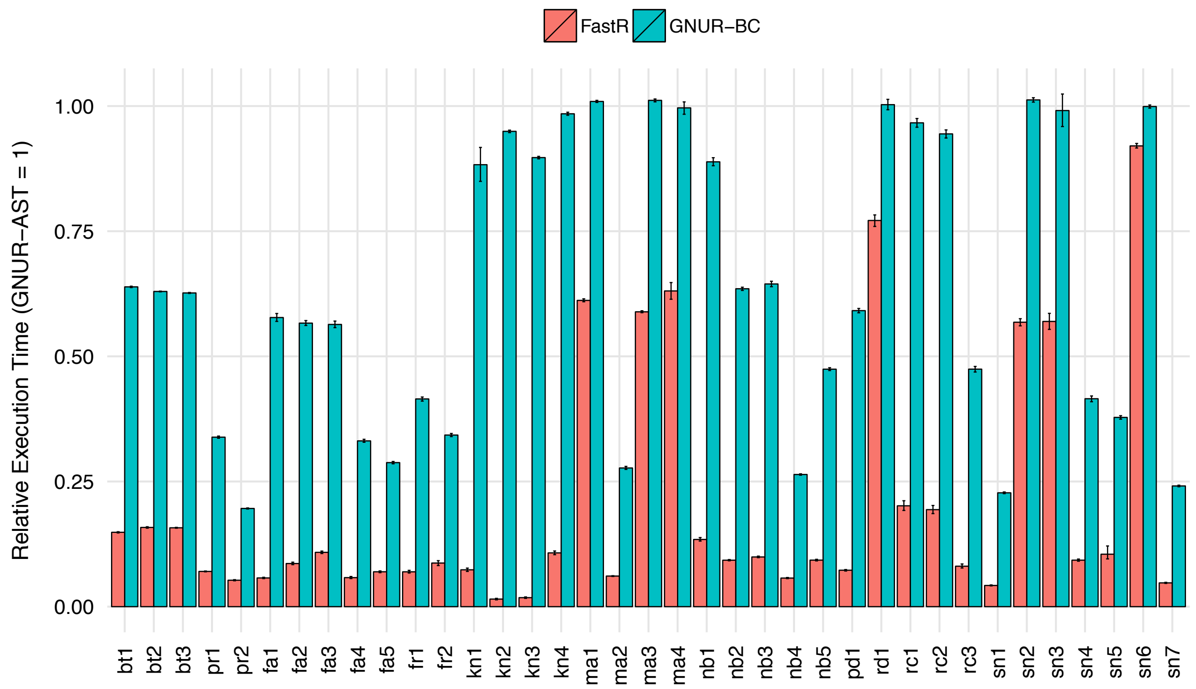

Figure 13. Shootout Relative Execution Times (lower is better). Geo. mean speedup for FastR is 8.5x and 1.8x for GNUR-BC.

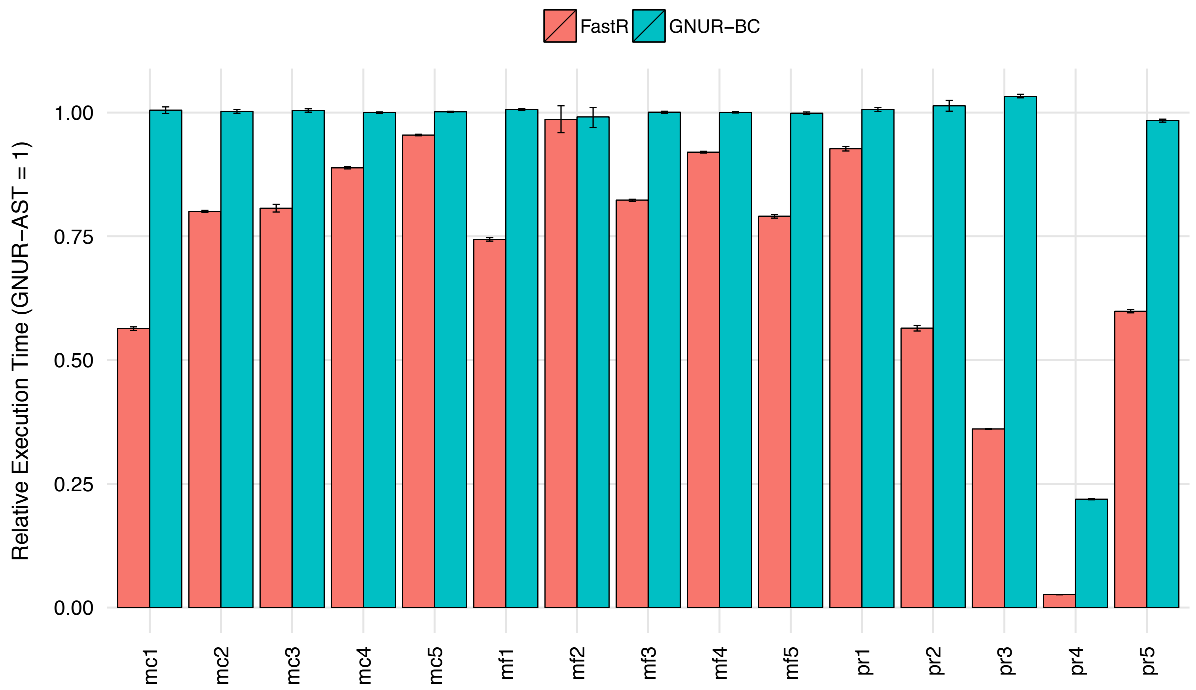

Figure 14. Benchmark 2.5 Relative Execution Times (lower is better). Geo. mean speedup FastR is 1.7x and 1.1x for GNUR-BC.

FastR, we use a pre-allocated exception (the return value is stored in the R Frame). Moreover, FastR's return elision optimization avoids executing the return statement completely in the bt benchmarks.

Fusion. Our implementation of fusion of view trees into Java byte-code, on average, provides no performance change on the b25 and the shootout benchmarks (numbers shown in graphs are without fusion). Lacking a realistic application that would stress vector computation, we use a trivial micro-benchmark to validate the potential speed-up of fusion. We

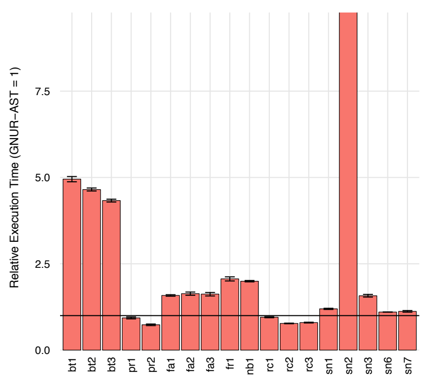

Figure 15. Shootout Relative Execution Times: Renjin (lower is better). Y-axis cut off at 10x slowdown (sn2 slowdown is 15x). Geo mean slowdown is 1.8x.

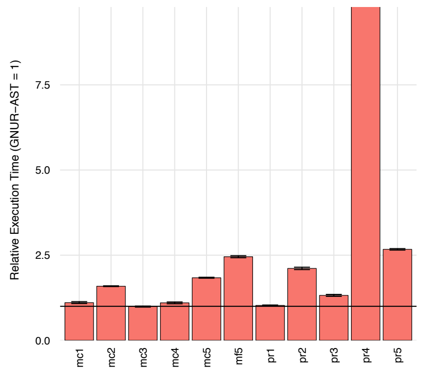

Figure 16. Benchmark 2.5 Relative Execution Time: Renjin (lower is better). Y-axis cut off at 10x slowdown (`pr4` slowdown is 107x). Geo. mean slowdown is 2.2x.

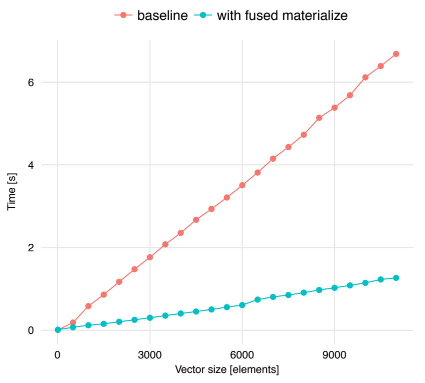

Figure 17. FastR Fusion. (smaller is better). Fusion speeds up materialization on a micro-benchmark as vector size increases. Approx. 7x speedup.

measure the time to compute a sequence of commands `x = y + z * y + z - 2 * (8 + z); x[[1]] = 3` for primitive vectors `y` and `z` of increasing size. Note the vector update of `x` which triggers materialization of the full vector `x`. With fusion enabled, the materialization will be performed using automatically fused and dynamically loaded byte-code for the particular computation. We report average time to calculate 10,000 repetitions of the code sequence (Figure 17) for increasing vector size. We exclude initial 30,000 iterations, focusing on peak performance of repeated computation rather than fixed cost of fusing a view tree.

7. Conclusions

Many languages start with a straightforward AST interpreter. As their popularity grows, the initial implementation usually starts to feel slow. This paper shows that the battle is not necessarily over. We have implemented a Java-based AST interpreter for a subset of the R language which, on some benchmarks, is 8.5x faster than a C AST interpreter and 4.9x faster than a byte-code interpreter. The techniques we used are all, individually, simple and require understanding of the application domain rather than heroic compiler skills. These techniques can be ported to a C interpreter and the ideas to other languages.

Our implementation leveraged the Java runtime system in a number of ways. We benefited from Java's garbage collector, from its ability to generate and load code dynamically, and from the productivity that comes with a type safe object-oriented language. But, there is a price too. Integration with the myriad of native functions used by GNU R is painful as JNI is cumbersome and slow. The lack of complex numbers and accompanying complex number arithmetics complicates the implementation. Math functions implemented in Java are often slow compared to their native equivalent.

Specialization worked well on our workloads, but one should be careful about generalizing. On one hand, the problems we looked at are kernels that are simpler than real code. On the other hand, they manipulate relatively small amounts of data. Some of our speedups will be more pronounced with large vectors. But this remains to be shown in practice. As we increase coverage of the language we will be in a position to better evaluate the true benefits of our optimizations.

Drawbacks of specialization are that it greatly increases code size of the interpreter and result in a non-linear body of code. We found we were writing boiler plate code for multiple variants of a node, yet it was not sufficiently repetitive, so there was not a clear way to generate it automatically. Understanding control flow in the interpreter is made difficult by the fact that classes are related by rewriting relationship.

Working at the AST level was convenient because tree rewriting is easy to implement, but it is a less efficient representation than bytecode and each node is optimized in isolation, with no information from its context. An extension to our work would be to look at how to perform similar changes directly on the bytecode and combine them with some program analysis. It is not clear if going to bytecode rewriting would raise the complexity bar too high for non-experts, though. We plan to add parallelism to our implementation, hence we will be forced to deal with concurrency.

Availability

FastR v0.168 is released under a GPL license and can be downloaded from:

`http://github.com/allr/fastr`

Acknowledgments

The authors thank Leo Osvald, Brandon Hill and Gaël Thomas for their help early in the project. Michael Haupt, Helena Kotthaus, Luke Tierney and Duncan Temple Lang provided us with feedback. We thank Tomas Würthinger, Mario Wolczko and the Truffle team for the idea of using code specialization. This work was supported in part by grants from the NSF and the ONR, and by gifts from the Oracle Corporation.

References

[1] P. S. Abrams. *An APL machine*. Ph.D. thesis, Stanford University, 1970.

[2] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The new S language: a programming environment for data analysis and graphics*. Wadsworth and Brooks/Cole Advanced Books & Software, 1988.

[3] C. Chambers and D. Ungar. Making pure object oriented languages practical. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1991.

[4] S. Chirokoff, C. Consel, and R. Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4), 1999.

[5] D. Eddelbuettel and C. Sanderson. RcppArmadillo: accelerating R with high-performance C++ linear algebra. *Computational Statistics & Data Analysis*, 71, 2014.

[6] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2009. .

[7] I. Gouy. The computer language benchmarks game[6]. `http://benchmarksgame.alioth.debian.org`, 2013.

[8] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3), 1996.

[9] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2), 2001.

[10] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[11] L. Osvald. R shootout. `http://r.cs.purdue.edu/hg/r-shootout`, 2012.

[12] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2008. `http://www.r-project.org`.

[13] D. Smith. The R ecosystem. In *R User Conference (UseR)*, 2011.

[14] J. Talbot, Z. DeVito, and P. Hanrahan. Riposte: a trace-driven compiler and parallel VM for vector code in R. In *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[6] This citation was corrected after publication of this paper.

[15] L. Tierney. *A Byte Code Compiler for R*. University of Iowa, 2012. `http://homepage.stat.uiowa.edu/~luke/R/compiler/compiler.pdf`.

[16] S. Urbanek. R benchmark 2.5. `http://r.research.att.com/benchmarks`, 2008.

[17] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of Onward!, the ACM Symposium on New Ideas in Programming and Reflections on Software*, 2013.