Specializing Ropes for Ruby

Kevin Menard
Oracle Labs
kevin.j.menard@oracle.com

Chris Seaton
Oracle Labs
chris.seaton@oracle.com

Benoit Daloze
Johannes Kepler University Linz
benoit.daloze@jku.at

ABSTRACT

Ropes are an immutable data structure for representing character strings via a binary tree of operation-labeled nodes. Ropes were designed to perform well with large strings, and in particular, concatenation of large strings. We present our findings in using ropes to implement mutable strings in TruffleRuby, an implementation of the Ruby programming language using a self-specializing abstract syntax tree interpreter and dynamic compilation. We extend ropes to support Ruby language features such as encodings and refine operations to better support typical Ruby programs. Finally, we evaluate the performance of our implementation of ropes and demonstrate that they perform 0.9× – 9.4× as fast as byte array-based strings in benchmarks of common Ruby string operations.

ACM Reference Format:
Kevin Menard, Chris Seaton, and Benoit Daloze. 2018. Specializing Ropes for Ruby. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang'18).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Strings of characters are one of the most frequently used data types in general purpose programming languages. Providing a textual representation of natural language, strings are a useful type for interacting with users at I/O boundaries. They are often used for internal operations as well, particularly where program identifiers are conveniently represented as strings, such as in metaprogramming.

The predominant string representation is as a thin veneer over a contiguous byte array. Consequently, the runtime performance of string operations in such systems follows that of a byte array.

The Cedar programming language [9] introduced *ropes* [1] as an alternative string representation in order to improve performance of string operations over large strings. In this paper, we use ropes to represent the mutable and encoding-aware strings of the Ruby programming language. We add metadata to each rope to specialize string operations based on both the structure and the properties of the input ropes, such as byte and character length, fixed- vs. variable-width encodings, ASCII-only characters, etc.

In summary, this paper makes the following contributions:

- We enhance the ropes data structure to be encoding-aware and use it to implement Ruby's mutable string type.
- We show how metadata from our enhanced ropes can be used to specialize & optimize string operations, being of particular benefit for strings with multi-byte characters.
- We introduce a new rope type to represent Ruby's repeating string operation, lazily.
- We show how string comparisons can be optimized with ropes, notably for Ruby's meta-programming operations.

ManLang'18, September 2018, Linz, Austria
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

We first discuss traditional string representations and ropes in detail, along with the complications introduced by multi-byte characters. We then present the semantics of Ruby's strings. Finally, we introduce our execution environment: TruffleRuby.

2.1 String Representation

The classical representation of strings is as an array of bytes with an optional terminal character. These strings require contiguous memory cells and optimize for compactness. Consequently, they have the same advantages arrays have, such as constant-time element access and data pre-fetching.

The byte array representation is not well-suited for all applications, however. Requiring contiguous memory may prevent allocations if a free block cannot be found. Additionally, operations that require byte copying, such as string concatenation, may lead to excessive memory fragmentation and suboptimal performance if done repeatedly. Finally, in some languages such as Java, the length of an array may have an upper-bound that is smaller than addressable memory, placing a limit on the length of a string.

2.2 Ropes

Ropes [1] are a persistent [6] and immutable data structure that can be used as an alternative implementation of strings. They were designed for the Cedar programming language [9], with the explicit goal of improving runtime performance over large strings. They were inspired by other immutable string representations, but introduced the efficient sequencing of operations by representing them as operation-labeled nodes linked together in a binary tree.

Since ropes are chained together via pointers, the upper-bound on the length of a rope is the total amount of addressable memory. Moreover, the nodes in a rope do not need to reside in contiguous memory. The price of this flexibility is a heavier base data structure. Depending on the application, however, it may be possible to recuperate that cost via de-duplication, as the immutable nature of ropes means they can be reused by the runtime. Beyond the additional overhead of the data structure, ropes suffer from some inefficiencies that mutable byte arrays do not. In the pathological case of transforming each character in a string, the rope would devolve into a linked list with one node per character. The Cedar environment employed heuristics to recognize a handful of such problematic usage patterns and provide specialized operations for them.

While Cedar ropes did provide limited compatibility with their traditional string type, called `text`, they were two distinct types; any usage of ropes was a deliberate decision by the programmer. We are unaware of any system developed since that has a rope-like representation exposed as a core data type. More commonly, languages such as a Java, provide different types for immutable strings

(`String`) and mutable buffers (`StringBuilder`), but they are both flat representations. Rather than introduce ropes as a competing data type, we consider their viability as an implementation strategy for Ruby strings.

2.3 Encodings

Encodings provide a mapping from a sequence of bytes to a set of characters. The byte array representation of strings is optimized for characters that can be represented within the space of a single byte, such as with the `ASCII` encoding. For modern encodings such as `UTF-8`, which defines its set of characters with byte sequences of varying width, a simple byte-oriented representation such as in C's strings or in Cedar's ropes loses many of its advantages. E.g., looking up a character is no longer a simple array reference; the individual character bounds must be calculated and a multi-byte sequence may be returned. Modern string implementations must work well with non-`ASCII` encodings.

2.4 Ruby

Ruby [12] is an object-oriented, dynamically-typed programming language. In contrast to many languages, it features *mutable* strings as a core data type and supports multilingualization (m17n), which means each string has an associated encoding. Multiple encodings can be loaded and in use within the same process. Ruby strings also serve as the language's byte array type via a special binary encoding. Due to its rich features, such as metaprogramming, versatile collections [10], and a pure object-oriented design, Ruby has been an attractive target for language implementors and researchers.

2.5 TruffleRuby

TruffleRuby [16] is a high-performance implementation of Ruby written as a self-optimizing AST interpreter on top of the Truffle language implementation framework [21]. To achieve peak performance, TruffleRuby must be paired with GraalVM [19, 20] – a convenient distribution that bundles together OpenJDK and the Graal dynamic compiler.

TruffleRuby achieves much of its performance via *specialized* implementations of core Ruby methods. Most often these specializations are based upon argument types so that polymorphic calls can be distilled down to their constituent cases without incurring the overhead of unused code paths. However, it can also specialize on attributes associated with values. In our implementation, the various rope cases are distinct types allowing for type specialization, but they also track metadata that we can specialize on, such as character width. Being able to specialize on both structure and data allows us to tailor the code generated for each call site.

3 IMPLEMENTATION

Our implementation of ropes (Fig. 1) consists of an abstract base `Rope` class and subclasses for the lazy string concatenation (`ConcatRope`), lazy substring (`SubstringRope`), and lazy string repetition (`RepeatingRope`) operations. We also have an abstract `LeafRope`, which represents the root of the leaf rope hierarchy. In a departure from Cedar's ropes, our ropes codify string encoding information in the leaves. Thus, we have leaves for ropes encoding

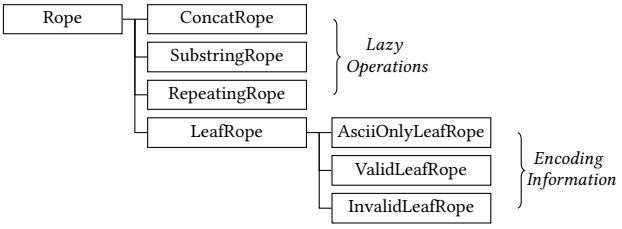

Figure 1: TruffleRuby's rope class hierarchy. `LeafRope` subclasses are split by the rope's character encoding.

ASCII-only characters (`AsciiOnlyLeafRope`), for ropes representing valid multibyte characters (`ValidLeafRope`), and for ropes with invalid byte sequences for the rope's encoding (`InvalidLeafRope`).

Our ropes store metadata useful for both Ruby language semantics and optimization of strings operations. Each rope stores the *encoding* and *code range* (detailed below), as well as a *single-byte optimizable* flag, to guide optimizations of string operations. We also store the *byte length* and *character length*, along with the Ruby-level *hash code*, for the string represented by the rope.

In another departure from Cedar's ropes, we never employ recursion to traverse the tree. Truffle's *partial evaluator* does not work well with unbounded recursion, so we have opted to use an iterative treewalk where necessary. Removing recursion means some of the heuristics Cedar employed to limit tree depth are no longer necessary. While an iterative treewalk requires the allocation of a stack of nodes to track position, it can be heap allocated so its growth is not much of a concern. Recursive walks, on the other hand, must limit tree depth in order to manage the call stack size. As we are not concerned with depth, we have also eliminated tree rebalancing to make concatenation a constant time operation. The trade-off is character retrieval by index may degrade to a linked list if the tree is extremely unbalanced.

3.1 Specializations

Efficient implementation of string operations is an important contributing factor to the overall performance of many language runtimes. The choice of string encoding can drastically impact the runtime complexity of those operations. Without careful consideration, supporting multiple encodings can result in operations that are only as fast as the slowest encoding allows.

While a runtime may support a wide array of encodings (TruffleRuby ships with 101 different encodings), in practice many applications use a small subset of those encodings. Moreover, applications quite often only use encodings that are *compatible* with each other (i.e., they support conversion from one to the other without reinterpretation of the underlying bytes). Compatible encodings can often be treated as a homogeneous type for the purposes of optimization.

By specializing our operations on both the structure and the content of ropes, we are able to avoid any slow paths associated with encodings, or classes of encodings, not used so far for that operation. Should a previously unseen encoding, or class of encodings, be encountered, we *deoptimize* [8] and transition to a more generalized form of the operation that provides correct functionality for the entire set of observed encodings. We limit the performance impact of heterogeneous classes of encodings to appropriate call sites by

method cloning [5]. When cloned, a string operation specializes to only the runtime values passed to it at that specific call site.

Due to the importance of adequately handling string encodings, we have extended ropes to encode the most critical information needed for specialization decisions into the leaf node types and a set of `final` metadata fields within each rope. The leaf node types correspond to a broad classification of encodings, known as *code ranges* in Ruby. They partition strings into those known to consist only of 7-bit ASCII characters, those that have a valid byte representation for their associated encoding (but at least one non-7-bit ASCII character), and those with an invalid byte representation for their associated encoding. The metadata fields can be used to further divide ropes based on more refined criteria, such as the width of a character, regardless of encoding. As a consequence, we specialize our operations on a wide range of discriminators, including empty vs. non-empty, fixed- vs. variable-width, UTF-8 vs. other variable-width encodings, rope node type, rope structure, and rope equality.

3.2 Access to Compiler Optimizations

Sophisticated language implementations like TruffleRuby may use dynamic compilers such as Graal with optimizations such as partial escape analysis [17] and scalar replacement of aggregates, which allow small objects that do not escape the compilation unit to avoid heap allocation and exist only as temporary values. Strings with their own potentially large character arrays are unlikely to be small enough to meet the heuristics of these algorithms, but rope objects with their small fixed size are. For instance, consider a composite operation such as `(stringA + stringB).slice(i,j)`. With a flat representation, a compiler would have to allocate intermediate byte arrays unless the size of stringA and stringB contain very small byte arrays and their size is known at compile time. With ropes, Graal is able to avoid allocations of all intermediary String and Rope objects, regardless of the size of the input strings.

3.3 Operations

Concatenation and Addition. One of the starkest contrasts between a simple byte array-based string representation and ropes is how string concatenation is handled. In its simplest form, the former requires the allocation of a new byte array large enough to hold the contents of both strings and then those strings are copied to the output buffer. This is a linear time operation that must keep two copies of each argument in memory while the operation is being performed. In contrast, ropes simply create a new `ConcatRope` node with its child pointers referencing the two strings being concatenated; a constant time operation with a fixed amount of memory overhead for the additional node.

Laziness pays off if a string performs a series of N concatenation operations before the full list of bytes is needed. Here, the rope approach only requires a single byte buffer allocation, filling the buffer during a tree walk, whereas the byte array-based representation requires N allocations (1 per concatenation, modulo reductions due to preallocated extra space).

Ruby supports string concatenation and addition, both of which combine the contents of two strings. Whereas concatenation in Ruby is destructive in the first operand, addition allocates a new string whose contents are the result of the combination. We represent both operations with the same rope structure, as seen in Figure 2, only modifying the rope reference in the string object for concatenation. A `ConcatRope` node represents the lazy operation and its metadata is populated as a union of its children's values, with Ruby-specific rules governing conflict resolution.

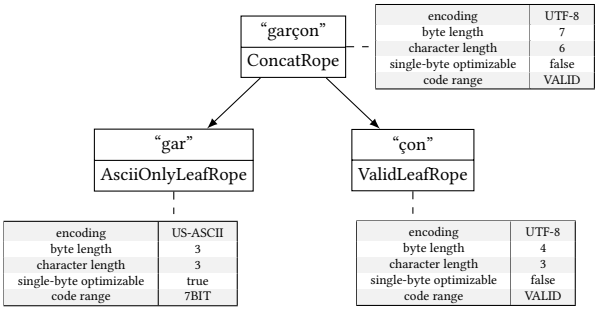

Figure 2: String concatenation and addition are lazy operations represented by a `ConcatRope`.

By storing the metadata for all ropes, we allow string operations to access that metadata uniformly across all rope types. For instance, the character length of a `ConcatRope` is the sum of its children's character lengths. We opt to store the value in a field of the rope instead of doing a full treewalk to determine the value. By doing this everywhere, each child's length must also be populated already, making the eager calculation a simple addition operation.

Substring. Taking the substring of a string is a general operation that can take on many forms. Ruby strings support character retrieval by index (which returns a string containing the character), truncation, character iteration, regular expression matching, and other operations that can be modeled as a variation of substring. In its most general form, we perform the operation lazily and denote it with a `SubstringRope`, as illustrated in Figure 3. In addition to the metadata that all `Rope` instances contain, `SubstringRope` instances also store a *child* reference to the string being substring'ed and a *byte offset* into the child. As with `ConcatRope`, we eagerly calculate all metadata for optimal performance, but only materialize the substring's bytes when called for.

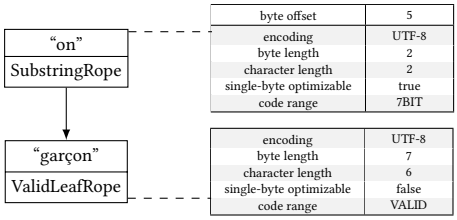

Figure 3: Taking a substring is generally a lazy operation represented by a `SubstringRope`.

In limited cases we opt to eagerly perform the substring operation, either in part or in whole. Taking a single-byte substring is a frequent operation in Ruby. It is advantageous for us to forgo the `SubstringRope` in favor of a `LeafRope` representing the single byte. As an additional step, we cache the single-byte `LeafRope` instances in lookup tables for the most popular Ruby encodings (each sharing the same backing byte array), guaranteeing reference equality for the results of all single-byte substring operations.

When taking the substring of a rope that is itself an instance of `SubstringRope`, we collapse the operation by adding the two substring offsets together. The result is two distinct `SubstringRope` nodes that share the same child, as shown in Figure 4. While tree depth is not something we generally must be concerned with in practice, by reducing the construction we limit the likelihood that a small substring operation keeps a large intermediary tree live for GC purposes.

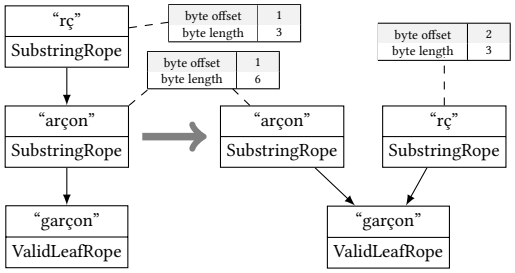

Figure 4: Taking the substring of a `SubstringRope` can be reduced to a new `SubstringRope` of the original's *child* by adding offsets.

Likewise, when taking the substring of a `ConcatRope`, we compare the *byte offset* and *byte length* values of the substring operation to each of the `ConcatRope`'s children. If the values cross the boundary between the two children, then we insert a `SubstringRope` whose child is the `ConcatRope`. Otherwise, we pick the appropriate child and encounter one of two cases: 1) the substring matches the range of the child exactly, in which case the result of the substring operation is simply a reference to that child; or 2) the substring is smaller than the child, and so we restart the substring operation over the child, taking the child node's type into account.

Repetition. Ruby includes a string "multiplication" operation that returns a new string consisting of the receiver repeated N times. In a byte array representation, this operation requires the allocation of a new buffer of size $|source| \times N$ bytes. The operation then copies the source string's byte array to the destination buffer N times.

With ropes, we have several ways to model the operation. We can treat the result as a `LeafRope` and populate its bytes in the same manner as a byte array-oriented approach would. However, it may be more advantageous to make use of the inherent byte array sharing of the source string. In this case we can treat the operation as a series of $N - 1$ concatenations. For successive powers of 2 we can even mirror one side of the concat tree to the other, further maximizing our ability to share already constructed objects.

A third option is to treat the operation as a simple run-length encoding. Our `RepeatingRope` represents a lazy operation by storing a reference to the source string and the repetition count. For many Ruby string operations, such as character retrieval by index, the `RepeatingRope` instance can operate as a lazy sequence, satisfying the request without needing to reify a byte array for the string being represented. This approach is both memory and time efficient, making repetition a constant-time operation instead of linear-time.

TruffleRuby specializes on both the metadata of the receiver string and the value of N to choose between each of the implementations. While `RepeatingRope` performs well in the general case, there are situations in which the other algorithms have an advantage, such as small repetitions of single-byte strings.

Tree Flatten. We eliminate the need for multiple treewalks across the entire rope by caching the resulting byte array at the root. However, this still keeps the entire tree resident in memory. For string interning we wish to store compact ropes and thus we have an eager flatten operation to eliminate the tree entirely. When flattening a `LeafRope` we can trivially return the rope. In all other cases the result of the operation is a newly allocated `LeafRope` instance, which by definition has its byte array populated.

TruffleRuby has a global rope table, consisting of only `LeafRope` instances, which we use to share and de-duplicate ropes for string literals and interned strings (i.e., Ruby symbols). Insertion and retrieval from the table are slow-path operations and thus the overhead of the flattening operation is not a concern for us. We trade off CPU time for reducing the memory usage of the table.

Bytes. Much of the benefit of ropes derives from few string operations ever needing direct access to the underlying bytes. By deferring the construction of their byte arrays until necessary, ropes avoid many costly memory allocation and copy operations. Ruby's `String` class, however, has methods that require direct byte access such as retrieving a copy of the underlying byte array, walking the bytes with an iterator, and retrieving an arbitrary byte by index, amongst others.

For the purposes of byte-oriented operations, ropes can be partitioned into two classes: those that have their internal byte array populated and those without a computed byte array. By definition, all `LeafRope` instances fall into the former category. By extension, due to the flattening operations on the interned rope table (see *Tree Flatten* above), all string literals fall into the former category. For ropes with a populated byte array, we can provide a fast-path specialization that simply uses the byte array for bytewise operations.

For ropes without a computed byte array, we determine on a per-operation basis whether it is cost-effective to populate the byte array. Once populated, that rope can then proceed down the fast path for subsequent bytewise operations. Alternatively, for operations needing only a few bytes, we can walk the rope tree to get bytes from the leaves and avoid the cost of flattening.

4 EVALUATION

In order to evaluate the impact of choice of string representation, we compared our rope implementation against traditional byte array approaches in benchmarks that stress critical Ruby string operations. We wrote two micro benchmarks to measure string equality and character retrieval by index in the presence of multibyte characters. We also benchmarked rendering a HTML template, using the Ruby standard library template engine, *ERB*.

For our comparisons, we implemented a `RopeBuffer` that matches the API of `Rope` but is backed by a mutable byte array. `RopeBuffer` instances are always leaf nodes as any modifications to them can be made *in situ*. We modified TruffleRuby to add `RopeBuffer`-specialized variants of the benchmarked string operations.

We also compared the performance of TruffleRuby's ropes to that of strings in other Ruby implementations. While those comparisons are illustrative of general Ruby string performance, we cannot make claims about how well a rope representation would work in those runtimes due to the innate differences in their virtual machines and compilers. All experiments were run on a system with an Intel Core

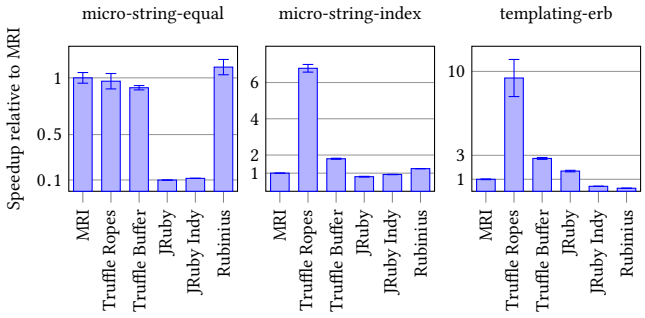

Figure 5: Comparison of Ruby runtime performance on the *micro-string-equal*, *micro-string-index* and *templating-erb* benchmarks.

i7-4390K processor with 6 cores each at 3.4 GHz and 32 GB of RAM, running Ubuntu Linux 14.04. We evaluated MRI 2.3.0, TruffleRuby 0fcb104 with GraalVM 0.11, JRuby 9.0.5.0, and Rubinius 3.15. We measured peak performance by running each benchmark in a loop for warmup until they stabilized, and then recorded 10 iterations. Reported errors are the standard deviation.

4.1 String Equality

Efficient comparison of strings is necessary for a high performing Ruby runtime. While string equality is a concern for typical application usages such as filtering user data and dictionary lookup, it is also used frequently within Ruby for metaprogramming facilities. For example, it is common to invoke methods using their string or symbol name with the `#send` method. In extreme cases, method names are constructed dynamically from multiple sources [15]. TruffleRuby uses inline caches [7, 11] to select methods to call based on a name. The name is compared against cached names that have been used before, which then yields the corresponding method. When the method names are symbols, the names are interned and a reference comparison is enough to check equality. When the methods names are strings, it is possible to take advantage of ropes to enable a faster comparison. Specifically, when the ropes have been constructed from the same sources, reference equality of either rope objects or their backing byte arrays is enough to prove equality, and avoid a byte-by-byte comparison.

The *micro-string-equal* microbenchmark allocates two 10 million character, 7-bit ASCII strings, and measures the runtime's performance in comparing them. In order to prevent byte array sharing between the two strings, a mutation is made to one of them and then reverted. The strings being compared are logically equivalent, so there is no early bailout possible via mismatch detection.

Our ropes perform slightly worse than a byte array representation, as can be seen in the left-most graph in Figure 5. Due to the mutations required to prevent byte sharing, the rope representation is a tree that reflects those operations, rather than a simple `LeafRope`. When comparing non-leaf ropes, our implementation first flattens the tree, which involves additional memory allocations. It then compares the resulting arrays as if they were `LeafRope` instances. The TruffleRuby rope buffer results illustrate how the TruffleRuby runtime performs when comparing strings that begin in the flat state.

4.2 Character Retrieval by Index

The *micro-string-index* microbenchmark measures the performance of Ruby runtimes in retrieving a character by index in the presence of multi-byte characters. It adds together a 10-character ASCII string with a single character, 3-byte wide UTF-8 string, and then retrieves each character by index repeatedly.

The results in the middle graph of Figure 5 highlight the value in making ropes encoding-aware. Our 6.8x speed-up over MRI is largely due to ropes tracking both byte length and character length, making bounds checking a very cheap operation. The semantics of Ruby only allow the concatenation of *compatible* strings, so the resulting string's length must be the sum of its children's lengths – a simple value to carry forward. Of the other runtimes, only Rubinius tracks character length. The remainder must do a byte scan to determine character length and do this for every character access.

Ropes only track a string's character length, not the individual character offsets. As such, determining where a variable-width character exists is also a linear operation for ropes. We minimize that cost by exploiting the rope's structure. In this case, the `ConcatRope`'s children are an `AsciiOnlyLeafRope` and a `ValidLeafRope`. By knowing the character length of each child and the index for the character retrieval operation, we can decompose the rope and choose the child that can best satisfy the request. Here, the indices 0 to 9 will route to the `AsciiOnlyLeafRope` where the request can be satisfied optimally. Retrieving the character for index 10 will route to the `ValidLeafRope`.

We note that if the rope is flattened, the performance differential drops to 2x that of MRI. The choice of benchmark is intended to mimic the behavior of real world templating applications, which often are authored with 7-bit ASCII characters but combine user-supplied input, such as a person's name, in the resulting string. In that situation, the rope structure is similar to the one in this benchmark and the same decomposition optimization applies.

4.3 HTML Template Rendering

ERB is the template engine included in the Ruby standard library. It processes a set of markup tags to handle Ruby expressions, which are typically used to dynamically generate content to be substituted into the document or to provide limited control flow to guide the rendering process. Its inclusion in the standard library makes it a popular first choice for many applications, such as HTML rendering. To ensure consistent results across all runtimes, our benchmark uses ERB from version 2.2.4 of MRI. The ERB processor takes a template file as input and generates a fragment of Ruby code. In a webserver, this Ruby code is then evaluated for each request to render HTML. The *templating-erb* benchmark renders a 1420-bytes HTML page by evaluating the Ruby code generated by ERB.

The generated Ruby fragment from ERB makes heavy use of string concatenation, which is generally favorable to ropes. The right-most graph of Figure 5 shows that TruffleRuby ropes are 9.4× faster than MRI, rendering around 870 000 templates per second. Our rope buffers execute at 2.7× the speed of MRI, suggesting the difference in speed for ropes is in part due to using a generally faster Ruby implementation, in addition to reduced byte array allocations. JRuby *without* `invokedynamic` operates at 1.7× the speed of MRI.

4.4 Summary

Ropes outperform buffers by a wide margin in the *templating-erb* and *micro-string-index* benchmarks. Both ropes and rope buffers have very similar performance for the *micro-string-equal* benchmark, even though the benchmark produces a sub-optimal situation for ropes. While we are careful not to generalize those results to all string operations, it is encouraging that the worst-case observed rope performance is competitive with a byte array representation.

5 LIMITATIONS AND FUTURE WORK

In some cases it is beneficial to flatten ropes when they reach a certain depth. Flattening can be an expensive operation, but yields a `LeafRope` that provides better performance than a tree for some string operations. We do flatten ropes in some cases such as when a string is interned as a symbol, but we are unaware of any work on sophisticated flattening heuristics.

In pathological cases a rope can degrade to a linked list of nodes representing each character in a string. We plan to investigate ways to either prevent or recover from this situation.

In our evaluation we have not considered the performance of I/O operations, as our benchmarks only measure the construction of strings. Future work could explore I/O operation specializations that can output a rope without flattening it. Additionally, Ruby strings have a dual purpose as a general byte buffer. We have not considered this potentially write-heavy use case in our evaluation.

Strings in TruffleRuby are currently limited to 2 GB, just as if they were backed by a Java `byte[]`. In follow-on work we would like to investigate supporting larger strings with our ropes.

In future work we will also investigate memory and garbage collection trade-offs between the two string representations.

6 RELATED WORK

Other Ruby String Implementations. All other implementations of Ruby use contiguous arrays of bytes to represent strings. Since Ruby strings are mutable, each time a string literal is encountered a new object must be created. Without careful consideration, this can lead to memory bloat if strings are mostly read-only.

The reference Ruby implementation, MRI [12], is written in C and uses a pointer and length field. In some cases the pointer can be copy-on-write shared by referring to the same character array, however this is only implemented for simple operations, such as strings from the same literal and simple substrings.

JRuby [13] is implemented in Java. However it does not re-use the standard Java `String` data type because Ruby strings are mutable and converting every external string to UTF-16 (the internal encoding of Java `String`) would be expensive in a language supporting strings with different encodings. Instead JRuby uses a `byte[]`, which it encapsulates in a `ByteList` class. JRuby's byte lists support copy-on-write for literals and substrings but do not support any lazy operations such as concatenation. As JRuby uses a standard Java `byte[]`, the length of strings are limited in JRuby to 2 GB.

Rubinius [14] is an implementation of Ruby using a VM in C++ but with much of the Ruby core library implemented in Ruby. Like MRI, Rubinius uses copy-on-write for string literals and for cases where one string is replaced with another. However, Rubinius does not share character data for substring operations.

Ropes in Other Language Implementations. PyPy is a high-performance implementation of the Python language using a meta-tracing just-in-time compiler [2]. PyPy has concatenation ropes, and earlier versions experimented with additional rope operations such as lazy substrings, but useful speedups were not observed and the more complex ropes were removed [4]. Python strings are immutable and idiomatic concatenation of long strings is often achieved by creating an array and then joining in a single operation, so the benefits of ropes may not be as clear in Python as they are in Ruby. Also, this work was prior to the meta-tracing JIT used in modern versions of PyPy, so interaction with compiler optimizations such as allocation removal were not considered.

Graal.js [18], a JavaScript implementation built on GraalVM [19], has a lazy concatenation string object, but no other rope operations. Other JavaScript implementations have similar string implementations. V8 calls concatenation ropes `ConsString`, and SpiderMonkey has `JSRope`.

Many language implementations implement substrings as a view into a shared character array. Java did this until 7u6, when it was changed to avoid potential memory leaks, which is an issue we do not address in this paper. Implementations that support ropes for concatenation, such as V8, will flatten a concatenated rope before sharing the character array.

Java, JavaScript, and Python have simpler string encoding semantics than Ruby with just one or two explicit encodings used. However implementations may use additional encodings internally. For example, V8 has two-byte strings as the JavaScript standard describes but also one-byte strings, and SpiderMonkey has optimizations including inline strings that do not allocate a separate character array. Java 9 now has single-byte compact strings [3].

7 CONCLUSION

We have evaluated the performance of ropes as a string representation for the Ruby programming language. Despite the incongruity of Ruby strings being mutable while ropes are immutable, we have found worst-case performance of ropes on some critical string operations to be competitive with a traditional byte array representation. We have demonstrated that ropes can have a significant performance advantage over byte arrays. Rope performance does vary with its structure due to our specialized methods in TruffleRuby, but we generally saw a performance range of $0.9\times - 9.4\times$ of MRI for representative benchmarks.

By tailoring our rope implementation to Ruby's semantics – notably making our ropes encoding-aware – we have reduced some core linear time operations to constant time. E.g., by tracking character length in addition to byte length we can perform bounds checking without having to do a full byte scan, regardless of the string's associated encoding.

The immutable nature of ropes allows us to freely share references, which makes them suitable for inline caching in the Truffle framework. Immutable metadata in the rope structure also makes it easier for the Graal compiler to perform constant-folding.

Our evaluation has been so promising that ropes are now how we implement Ruby strings in release versions of TruffleRuby.

REFERENCES

[1] Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An Alternative to Strings. *Softw. Pract. Exper.* 25, 12 (dec 1995), 1315.

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, 18–25.

[3] Brent Christian. 2018. JEP 254: Compact Strings. http://openjdk.java.net/jeps/254

[4] Carl Friedrich Bolz. April 2016. Private correspondence with the authors.

[5] Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 49–70.

[6] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1986. Making Data Structures Persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC '86)*. 109–121.

[7] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Conference on Object-Oriented Programming*. 21–38.

[8] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. New York, NY, USA, 32–43.

[9] Butler W. Lampson. 1983. A Description of the Cedar Programming Language: A Cedar Language Reference Manual. Xerox PARC Technical Report CSL 83-15.

[10] Stefan Marr and Benoit Daloze. 2018. Few Versatile vs. Many Specialized Collections – How to design a collection library for exploratory programming?. In *Proceedings of the 4th Programming Experience Workshop (PX/18)*. ACM.

[11] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-overhead Metaprogramming: Reflection and Metaobject Protocols Fast and Without Compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. 545–554.

[12] Yukihiro Matsumoto, Koichi Sasada, Nobuyoshi Nakada, et al. 2018. MRI – Matz's Ruby Interpreter – The Ruby Programming Language. https://www.ruby-lang.org/en/

[13] Charles Nutter, Thomas Enebo, Ola Bini, Nick Sieger, et al. 2018. JRuby – The Ruby Programming Language on the JVM. http://jruby.org/

[14] Evan Phoenix, Brian Shirai, Ryan Davis, Dirkjan Bussink, et al. 2018. Rubinius – An Implementation of Ruby Using the Smalltalk-80 VM Design. https://github.com/rubinius/rubinius

[15] Chris Seaton. 2015. *Specialising Dynamic Techniques for Implementing The Ruby Programming Language*. Ph.D. Dissertation. The University of Manchester.

[16] Chris Seaton, Benoit Daloze, Kevin Menard, Petr Chalupa, Brandon Fish, and Duncan MacGregor. 2018. TruffleRuby – A High Performance Implementation of the Ruby Programming Language. https://github.com/oracle/truffleruby

[17] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. 165–174.

[18] Andreas Wöß, Christian Wirth, Danilo Ansaloni, Daniele Bonetta, et al. 2018. Graal.js – A JavaScript implementation built on GraalVM. https://github.com/graalvm/graaljs

[19] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. 662–676.

[20] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'13)*. 187–204.

[21] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium (DLS '12)*. 73–82.