Specializing Ropes for Ruby

Authors removed for review

Abstract

Ropes are a data structure for representing character strings via a binary tree of operation-labeled nodes. Both the nodes and the trees constructed from them are immutable, making ropes a persistent data structure. 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 JRuby+Truffle, 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. We also use ropes to work around underlying limitations of the JVM platform in representing strings. Finally, we evaluate the performance of our implementation of ropes and demonstrate that they perform 0.9x – 9.4x as fast as byte array-based string representations in representative benchmarks.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords Virtual Machine, Interpreter, Ropes, Strings, Truffle, Graal, Ruby, Java

1. Introduction

Strings of character data are one of the most frequently used data types in general purpose programming languages. Providing a textual representation of natural language, they are a convenient form for interacting with users at I/O boundaries. They are often used for internal system operations as well, particularly where program identifiers are conveniently represented as strings, such as in metaprogramming. Some languages rely so heavily on strings that they are glibly referred to as "stringly typed."

The choice of string representation must strike a balance between potentially competing concerns. Even modest improvements to either string memory consumption or performance of common string operations could have a tremendous effect on overall efficiency.

The predominant string representation is as a thin veneer over a contiguous byte array. As a consequence, the runtime performance of string operations in such systems follows that of a byte array while the use cases may not.

The Cedar programming language [4] introduced *ropes* [1] as an alternative string representation in order to better match the design goals of the language. By modeling strings as an immutable binary tree where the leaves represent sequences of characters and interior nodes represent lazy operations, such as concatenation, the language was able to drastically alter the cost of common string operations. However, in order to satisfy performance goals for different use cases, ropes were introduced as a data type distinct from strings with limited compatibility between the two. In summary, this paper makes the following contributions:

- We apply the ropes data structure to represent strings in the Ruby programming language and show how self-optimizing AST interpreters facilitate their implementation.
- We introduce new lazy operations that can be represented as nodes in a rope.
- We show how ropes allow further optimizations to metaprogramming operations that are heavily used in idiomatic Ruby.
- We demonstrate that ropes, an immutable data structure, have utility in languages with mutable string data types.
- We show that ropes can reduce the cost of string operations involving multi-byte characters.

2. Background

This section discusses traditional string representations and ropes in more detail, along with the complications introduced by the presence of multi-byte characters. We then present Ruby's string semantics. Finally, we introduce our execution environment: JRuby+Truffle.

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.

By extending this simple representation with additional metadata, such as string byte length or head offsets, the set of fast operations available to strings can be expanded. E.g., string truncation can simply update the length value rather than require a new array allocation and byte copy.

The byte array representation is not well-suited for all applications, however. In some languages, such as Java, the length of an array may have an upper-bound that is smaller than addressable memory, placing a similar limit on the length of a string. Requiring contiguous memory may also 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.

2.2 Ropes

Ropes [1] are a persistent data structure [5] that can be used as an alternative implementation of strings. They were designed for the Cedar programming language [4], 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 deduplication, 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 to an allocated node for each character. The Cedar environment employed heuristics to recognize a handful of such problematic usage patterns and provide specialized operations for them.

2.3 Encodings

The byte array representation of strings is optimal as long as characters can be represented within the space of a byte. As computers have broadened their reach to a more diverse audience, the set of characters that must be representable has grown well beyond the 256 afforded by a byte. *Encodings* have been developed as a means of expanding the set of representable characters by interpreting *byte sequences*, rather than individual bytes, as characters. These byte sequences can either be fixed-width or variable-width.

When encodings are taken into account, traditional string representations can be seen as being optimized for fixed-width encodings with a character width of 1 byte. For any other type of encoding, some of the advantages inherent to byte arrays, such as constant-time character access, are muted.

Ropes, as implemented in Cedar, suffer from some of the same encoding issues as their byte array counterparts. E.g., accessing a character by index cannot be performed by a simple binary search. Instead, a full treewalk must be performed so the byte sequences can be processed.

2.4 Ruby

Ruby[1] is an object-oriented, dynamically-typed programming language with mutable, encoding-aware strings as a core data type. Ruby strings also serve as the language's byte array type via a special binary encoding.

Ruby strings can be interned to another data type known as a *symbol*. Due to their performance advantages and more compact syntax, symbols are used frequently in idiomatic Ruby. However, the symbol API is deliberately quite limited and symbols cannot be easily composed without constructing a string as a temporary buffer. Additionally, symbols were not eligible for garbage collection in Ruby versions prior to 2.2.0. Subsequently, Ruby programs often consist of a mixture of strings and symbols to perform similar tasks. The programmer must decide which type to use as a careful balance between code concerns, such as concision and value safety, and VM concerns, such as performance, garbage collection, and memory reuse.

2.5 JRuby+Truffle

Due to its rich features, such as metaprogramming, comprehensive collection APIs, and a pure object-oriented design, Ruby has been an attractive target for language implementors and researchers. JRuby[2] is an open source, alternative implementation of Ruby that uses its own compilation phases to apply Ruby-level optimizations before generating Java bytecode, including extensive use of the `invokedynamic` instruction. JRuby+Truffle [7] is developed as part of the JRuby project, providing an alternative backend for the JRuby runtime that eschews the guest language compiler in favor of a self-optimizing AST interpreter written using the Truffle framework [9]. To achieve peak performance, JRuby+Truffle must be paired with GraalVM [8] — a convenient distribution that bundles together OpenJDK and the Graal dynamic compiler.

JRuby+Truffle 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

[1] *Ruby*, Yukihiro Matsumoto and others, `https://www.ruby-lang.org/`

[2] *JRuby*, Charles Nutter, Thomas Enebo and others, `http://jruby.org/`

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. Ropes

Ropes are first mentioned by Boehm et al. [1] as an alternative to the byte array representation used by C and Pascal. They present ropes from the Cedar language [4] as an alternative to strings, driven by four main design goals:

1. *immutability* — Strings should be able to be passed around modules without concern for accidental modification. Additionally, immutable strings make concurrency guarantees much easier.
2. *efficiency* — Common operations, such as concatenation and substring, should be fast and memory efficient.
3. *scalability* — Common operations should perform well regardless of string length.
4. *adaptability* — Sources of string data (e.g., files and network streams) should also be treated as strings and support common string operations.

In contrast to mutable byte array representations, ropes are constructed from nodes into a binary tree that represents the contents of a string. The leaves of the tree do contain byte arrays — that is unavoidable without more data types built into the language — but the arrays themselves are immutable, satisfying the first goal. Efficiency and scalability are achieved by providing lazy operations for string concatenation and substring. These operations are represented by operation-labeled nodes in the tree. By modeling the operations as a simple node addition to the tree root, they can be performed in constant time. Adaptability is very much a runtime-specific concern and given Ruby and Cedar had different design goals, we do not consider it in our evaluation.

While Cedar ropes did provide limited compatibility with their traditional string type, called `text`, they were essentially 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. They satisfy goal #1, but otherwise perform their operations in much the same way that C strings have for decades. Rather than introduce ropes as a competing data type, we look to use them as an implementation strategy for Ruby strings.

4. Related Work

Other Ruby String Implementations All other implementations of Ruby use contiguous arrays of bytes to represent strings.

The reference Ruby implementation, MRI, 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, from which JRuby+Truffle was developed, is written in Java. However it does not re-use the standard Java `String` data type as some encodings supported by Ruby are considered by some people to not be faithfully representable in Unicode. 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 array of bytes, the length of strings are limited in JRuby to around 2 GB.

Rubinius is an implementation of Ruby using a VM in C++ but with much of the Ruby-specific functionality implemented in Ruby. Like MRI, Rubinius uses copy-on-write for string literals and for cases where one string is replaced with another, but 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 meta-circular implementation and 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 [3]. 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.

GraalJS, the JavaScript implementation using the same technology as JRuby+Truffle, 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`.

As described for the Ruby implementations, 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 is also exploring one-byte *compressed* strings.

5. Applications

Sharing String Data Other implementations of Ruby use some form of copy-on-write to reduce unnecessary copies of string character data. This can be as simple as a single character array used by different strings, possibly with different start offsets and lengths. In Ruby this is particularly important as string literals in the source code allocate a new string object each time they are executed, rather than referring to a single object as in Java. Ropes can be applied to achieve the same goal, with a single immutable rope shared between all instances of the string literal. We extend this to work for string literals across multiple source files by maintaining a global table of ropes into which we de-duplicate ropes as they are created. We also use this table to share ropes used by other long-lived objects, such as Ruby's symbols.

Lazy String Concatenation 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, in order, to the output buffer. This is a linear time operation that also requires enough memory to hold two copies of the resulting string 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.

Repetitive byte copying with a traditional string representation can be mitigated by allocating spare memory and tracking the length of the string, allowing smaller concatenation operations to occur within the excess space of the resulting string. However, the approach is impractical for large strings as it requires an undue amount of wasted memory in the form of preallocated, but unused, space. It also does not work when the result of the operation is to be non-destructive with regards to the source operands — in that case allocating a new byte array for the result is unavoidable.

Concatenations with ropes are effectively a lazy operation. While a new `ConcatRope` node is created immediately, the complete sequence of bytes for the string are now represented by the structure of the rope. Ropes as described by Boehm et al. never allocate a byte array for the entire string, rather relying on a treewalk whenever the bytes are needed. In our implementation, we have opted to sacrifice immutability of `ConcatRope` instances — yet maintain idempotency — by attaching a byte array to each node, regardless of type, in the rope. The first time the byte array is needed, it is calculated and then cached in the root node.

While we pay the cost of walking the tree for the initial calculation, subsequent requests for the byte array are constant time; similar to a byte array-based representation. However, if a string has a series of N concatenation operations before the full list of bytes is needed, the rope approach would only require a single byte copy operation, whereas the byte array-based representation would require N (1 per operation, modulo reductions due to preallocated extra space).

A special case of concatenation is repeatedly concatenating a string with itself, which Ruby provides as the `*` operator on string objects.

Lazy Substrings Generating a substring from an existing substring is a key operation for Ruby, a language designed to process text and used commonly to handle text data on the web. Breaking apart strings, matching in regular expressions, and parsing HTTP requests & JSON payloads are all examples of producing substrings that could refer back to the original string where they entered the VM's heap. With ropes, even as the substrings are processed, either being concatenated with other things or further substringed, they can still refer back to the original source of the string. As with lazy concatenation, lazy substrings allow us to allocate a new string without copying the data inside it.

Multi-byte Indexing Since strings in Ruby are encoding-aware and may use multi- and variable-byte schemes, retrieving a character by index implies knowing both the string length for bounds checking and the character boundaries for the underlying encoding. For strings with fixed-width encodings, the calculations are straightforward, but for common encodings like UTF-8 finding character boundaries requires non-trivial logic. UTF-8 strings are common on the internet and are the default encoding of Ruby source files which dictates the encoding of string literals in them.

An application of ropes is to help translate a character's index to the byte offset, because ropes can store both their byte length and character length. A string made up of a tree of ropes can then more easily seek to the rope that contains the character by skipping whole nodes based on character length. This is something of a side-effect of ropes, and only applies if they are made up of many rope nodes which may not always be the case.

Metaprogramming Idiomatic Ruby libaries and programs make very extensive use of metaprogramming, or reflective operations. For example, it is common to call methods using their string or symbol name and the `#send` method, if the method to call varies. In extreme cases, method names are constructed dynamically from multiple sources [6].

Language implementations use inline caches 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. For conventional method calls and with method names that are symbols, the names can be interned and so a reference comparison made. If method names are strings then a full comparison of the string's contents must be made.

When ropes have been constructed from the same sources, string comparisons can avoid a linear byte array comparison by performing reference equality checks on either the ropes themselves or their backing byte arrays.

Access to Compiler Optimizations Sophisticated language implementations may use dynamic compilers with features such as escape analysis, partial escape analysis, and scalar replacement of aggregates, which allow small objects that are not used outside of a method to avoid heap allocation and only exist as temporary values. Strings with their own character arrays are unlikely to be small enough to meet the heuristics of these algorithms, but the node objects in a rope probably will be.

A method that performs operations such as concatenation, character retrieval, and substring before returning a final result may be able to keep those intermediate rope node objects from escaping the compilation unit and so allow the compiler to apply the more sophisticated optimizations.

6. Long Strings

The Java language and JVM were designed when 32 bit processors were common, and so specify that arrays are indexed using a signed 32 bit integer, regardless of what the architecture and address space actually support. Java does support 64 bit integers, but not for indexing an array. As the integer types are also signed, this limits the size of a `byte[]`, and so a string that uses such a representation, to 2 GB.

With ropes we can continue to use `byte[]`, which is visible to optimizations and simple to access, but combine them in a rope to produce a string whose length is bound only in practice by available memory. One remaining limitation is that we still use a signed `long` to represent the string length.

Most Ruby applications will have no need for such long strings, but it does represent a deviation of behavior between JRuby and MRI, which is something that we aim to minimize. Ruby strings are used for raw binary data as well as text with a specific encoding, so the 2 GB limit does, for example, currently prevent JRuby from loading and calculating the message digest of a (not unreasonable) 3 GB file, without manually chunking it.

7. 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 repetition (`RepeatingRope`) operations. We also have an abstract `LeafRope`, which represents the root of the leaf rope hierarchy. In a departure from Boehm et al.'s ropes, our ropes codify string encoding information in the leaves. Thus, we have leaves for ropes 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`).

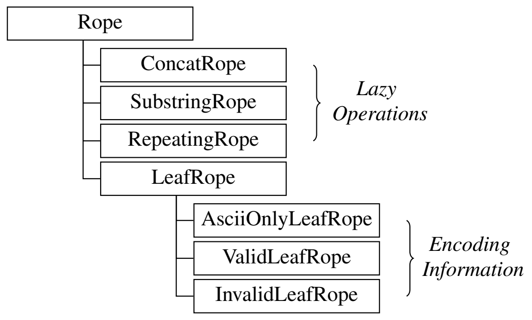

Figure 1. JRuby+Truffle's rope class hierarchy. `LeafRope` subclasses are split by the rope's character encoding. All other Rope subclasses represent lazy string operations.

In addition to the byte array, our ropes store metadata useful for both Ruby language semantics and the Graal compiler. Our ropes store *encoding* and *code range* values, as well as a *single-byte optimizable* flag, to guide optimizations on interpretations of the byte array. 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 of Boehm et al., 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 Boehm et al. applied to limit tree depth are no longer necessary. While an iterative treewalk requires the allocation of a node stack 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 linear time operation if the tree is extremely unbalanced.

7.1 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 performance degrading to that of the slowest encoding.

While a runtime may support a wide array of encodings (JRuby+Truffle 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 in active use. Should a previously unseen encoding, or class of encodings, be encountered, we *deoptimize* and transition to a more generalized form of the operation that provides correct functionality for the entire set of active encodings. We limit the performance impact of heterogeneous classes of encodings to appropriate call sites by *method cloning*.

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.

7.2 Operations

Concatenation and Addition Ruby supports string concatenation and addition. Both operations combine the contents of two strings, with the former being destructive in the first operand, while the latter 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 if necessary. 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.

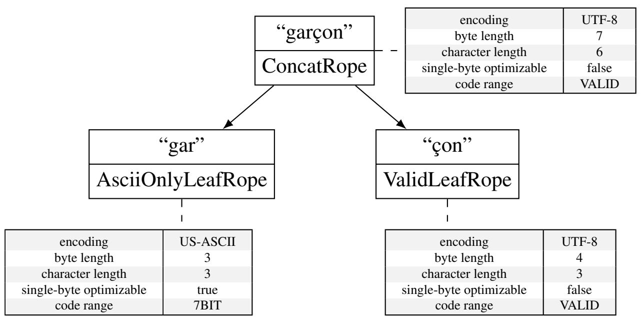

Figure 2. String concatenation and addition are lazy operations represented by a `ConcatRope`.

By storing the metadata at fixed locations within the object, we allow the compiler to perform optimizations on that metadata uniformly across all rope types. Avoiding the lazy evaluation of those values also allows for rope construction operations to occur in constant time, rather than linear time.

Substring Taking the substring of a string is a general operation that can take on many forms. Ruby supports character retrieval by index, string truncation, character replacement, character iteration, regular expression matching, and several 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 fields that all `Rope` instances contain, `SubstringRope` instances also store a *child* reference to the string being substringed and a *byte offset* into the child. As with `ConcatRope`, we eagerly calculate all metadata for optimal performance.

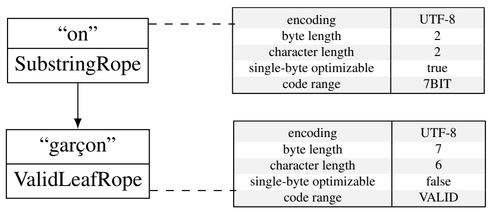

Figure 3. Taking a substring is generally a lazy operation represented by a `SubstringRope`.

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 demonstrated in Figure 4. 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.

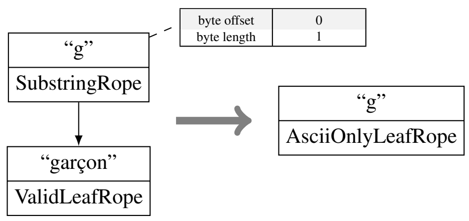

Figure 4. Taking a single-byte substring can be reduced to a single-byte `LeafRope`. Single-byte ropes are called for in many places in JRuby+Truffle and as such, we store them in a lookup table so the same instances can be re-used across the runtime.

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 5. While tree depth is not something we generally must be concerned with in practice, by reducing the construction we limit the likelihood a small substring operation keeps a large interstitial tree live for GC purposes. It also restricts the set of valid tree relationships, which is helpful for any operation that must match against the structure of the tree.

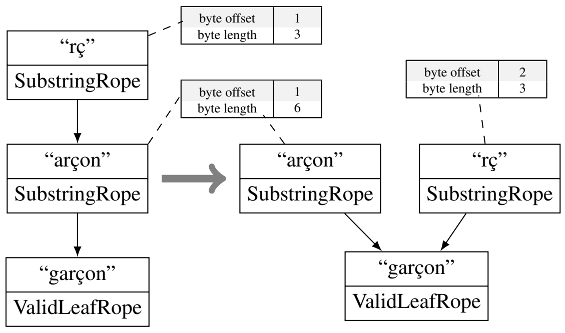

Figure 5. Taking the substring of a `SubstringRope` can be reduced to a new `SubstringRope` of the original’s *child* by combining offsets.

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 would require the allocation of a new buffer of size $|source| \times N$ bytes. The operation would then copy the source string’s byte array to the destination buffer *N* times.

With ropes, we have several ways we could 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` is a lazy operation that stores a reference to the source string and the repetition count. For many Ruby string operations, 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 efficient and time efficient, reducing a linear-time operation to a constant-time one.

JRuby+Truffle specializes on both the metadata of the receiver string and the value of *N* to choose between each of the implementations. While use of `RepeatingRope` performs well in the general case, there are situations in which one of the other two algorithms may yield better results, 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, as noted in Section 5. However, this still keeps the entire tree resident in memory. In some cases we wish to eliminate the tree entirely and thus we have an eager flatten operation. When flattening a `LeafRope` we can trivially return the rope, but in all other cases the result of the operation is a newly allocated `LeafRope` instance, which by definition has its byte array populated. NB: if a new rope is allocated, the result of the operation shares no references with the source rope, which may cause cache misses if the source rope was used as a cache key in any specializations.

All ropes are flattened before being interned in our rope table. Insertion and retrieval from the table are slow-path operations and thus the overhead of the flattening operation is inconsequential. By flattening, we reduce the size of the table and decrease the likelihood of entries being evicted during GC cycles.

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 either the complete or a partial (subrange) byte array until necessary, ropes avoid many costly memory copy operations. Ruby’s `String` class has methods for retrieving a copy of the underlying byte array, walking the bytes with an iterator, and retrieving an arbitrary byte by index, amongst others. When such a method is called, we would like to satisfy the request as efficiently as possible given the structure of the rope backing the string.

For the purposes of byte-oriented operations, ropes can be partitioned into two classes: those that have their internal byte array populated and those with a sparse 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 Section 7.2), all string literals fall into the former category. By specializing on whether the rope’s byte array is populated, we can provide a fast-path method that does a simple field load.

For the ropes with a sparse 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 string operations. Alternatively, rather than populate the byte array, we may opt to flatten the rope and then update the source string’s rope reference to the new flattened rope. As flattened ropes are always instances of `LeafRope`, this also allows future byte-oriented string operations to proceed down the fast path.

8. 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 have extracted a subset of the *fasta* benchmark that focuses on string operations and wrote several of our own benchmarks using the `bench9000` benchmarking tool. Of the benchmarks we wrote, two were micro benchmarks to measure string equality (*micro-string-equal*) and string length and character retrieval by index in the presence of multi-byte characters (*micro-string-index*). The last benchmark is a simulated HTML template rendering engine, drawn from 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 JRuby+Truffle to add `RopeBuffer`-specialized variants of the benchmarked string operations.

We also compare the performance of JRuby+Truffle'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 Xeon Core 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, JRuby+Truffle 0fcb104 with GraalVM 0.11, JRuby 9.0.5.0, and Rubinius 3.15. Reported errors are the standard deviation.

8.1 fasta

The *fasta* benchmark measures the generation of DNA sequences, with benchmarks split between repeated copying from a supplied sequence and random generation from source alphabets. The former benchmark is heavy on string operations, exercising Ruby's string multiplication, truncation, length, substring, and concatenation methods, whereas the latter is a mixture of heterogeneous data type calls. We extracted the repeated fasta benchmark for our evaluation. The benchmark was modified to collect results in a buffer rather than print them out to standard output so as to eliminate I/O overhead from the analysis.

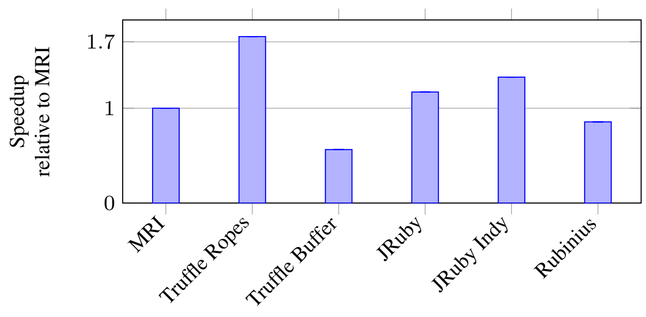

Figure 6. Comparison of Ruby runtime performance on the the *fasta* benchmark.

The selected operations are a mix of immutable and mutating calls so as to not implicitly favor one string representation over another. Figure 6 shows the relative performance of various Ruby runtimes, normalized to MRI's runtime performance. JRuby+Truffle's rope representation outperforms all other runtimes, although the error was very large (too large to show) which we believe is due to large garbage collection pauses.

Much of the difference in execution time can be attributed to the string concatenation operation. With ropes, this is a constant time operation. Other runtimes may need to allocate a new buffer and copy the contents from both the receiver and operand strings to the new buffer.

The substring operation is also performed repeatedly in this benchmark. For runtimes that perform copy-on-write for the byte arrays, the difference between ropes and a byte array representation is negligible in this context, as no modifications are made to either the source string or the resulting substrings after an initial truncation.

We did not evaluate the memory requirements of any implementations, other than to note that they each managed to calculate a 100 million character DNA sequence and split it into 60 character substrings within the bounds of a 2 GB heap. With repeated concatenations of substrings, the overhead of the rope data structure may eventually be problematic in a more generalized case. However, the non-rope representations each eagerly allocate 100 MB of contiguous memory to store the results of the string multiplication operation, whereas a `RepeatingRope` instance can simply store a reference to a `LeafRope` with a 287 element byte array along with a repetition count.

8.2 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.

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 Figure 7. 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 composite 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 JRuby+Truffle rope buffer results illustrate how the JRuby+Truffle runtime performs when comparing strings that begin in the flat state.

In future work we will look to lower the cost of the flattening operation or, optionally, eliminate it entirely. Without caching partially combined results intermittently throughout the tree, our rope comparison will always require a full treewalk. The benchmark only stresses shallow trees, but the 10 million bytes being compared will dominate the number of pointers traversed during a typical treewalk; indeed, this can be enforced by limiting the number of pointers via tree flattening during construction.

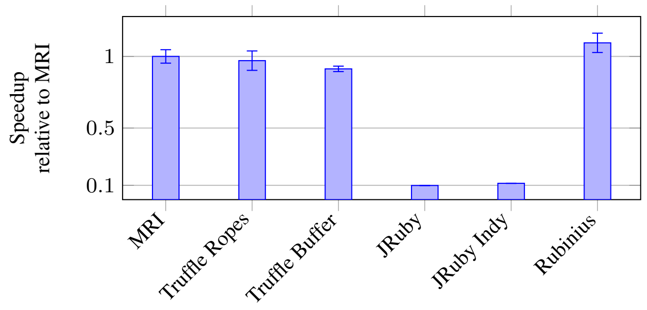

Figure 7. Comparison of Ruby runtime performance on the *micro-string-equal* benchmark.

8.3 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 iterates through each character in the resulting string.

The results in Figure 8 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 first $N-1$ operations will route to the `AsciiOnlyLeafRope` where the request can be satisfied optimally.

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 would qualify for the deconstruction optimization.

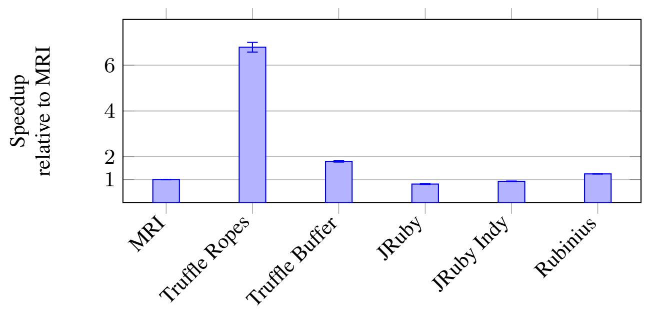

Figure 8. Comparison of Ruby runtime performance on the *micro-string-index* benchmark.

8.4 HTML Template Rendering

ERB is a templating engine that is included as part of Ruby's 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.

The *templating-erb* benchmark extracts a portion of the ERB rendering pipeline for evaluation of string processing. The entire rendering process involves many other aspects of a Ruby runtime which, while interesting, would detract from the evaluation of string performance. In particular, the immediate output of ERB's processor is a fragment of Ruby code to be dynamically evaluated. Our benchmark includes a pre-generated fragment and executes it, much like Ruby's *eval* would, generating the final product.

As ERB is part of the Ruby standard library, it is subject to change along with the standard library. To ensure consistent results across all runtimes, our extracted benchmark is based on ERB from version 2.2.4 of MRI.

The generated Ruby fragment from ERB makes heavy use of string concatenation, which is an operation that is generally favorable to ropes. As shown in Figure 9, we see that JRuby+Truffle ropes are 9.4x faster than MRI. Our rope buffers execute at 2.7x the speed of MRI, suggesting the difference in speed for ropes is not entirely related to a decrease in byte array allocations. JRuby without `invokedynamic` enabled operates at 1.7x the speed of MRI.

8.5 Ropes vs. Rope Buffers

Ruby has been fertile ground for language implementation, spawning several runtimes that see production use. Comparing our performance relative to alternative language implementations helps frame the context for our work. We also evaluated the performance differences between ropes and

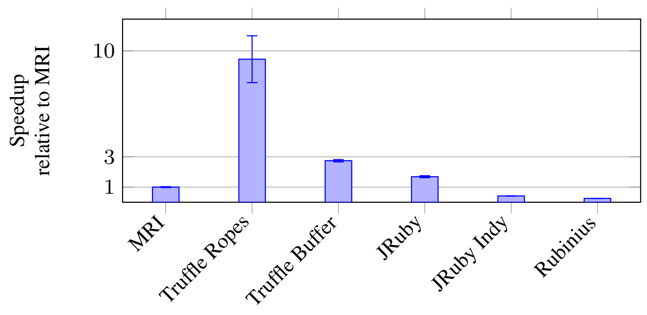

Figure 9. Comparison of Ruby runtime performance on the the *templating-erb* benchmark.

rope buffers within the same JRuby+Truffle runtime to minimize environmental differences. Both representations were specialized within our runtime for the operations being exercised in our benchmarks. However, differences in data representation can affect the partial evaluator in the Truffle/Graal system. We made a best effort to make both representations amenable to online compilation, but we did not normalize the process by disabling optimizations that applied to only one or the other representation. We care about peak performance and thus the choice of string representation is largely influenced by how it impacts compilation.

Ropes outperform buffers by a wide margin in the *fasta*, *templating-erb*, and *micro-string-index* benchmarks. Both ropes and rope buffers had very similar performance for the *micro-string-equal* benchmarks. The *micro-string-equal* benchmark in particular produces a situation that is sub-optimal for ropes. The *fasta* benchmark suffers from a wide margin of error, as noted in Section 8.1. While we are careful not to generalize those results to all string operations, it is encouraging to see that the worst-case observed rope performance is competitive with a byte array representation.

We did not measure differences in memory usage or garbage collection between the two representations. Both operated within the same upper limit of a 2 GB heap permitted to each of the benchmarks.

9. Limitations and Future Work

In some cases it is beneficial to flatten ropes when they reach a certain depth: a trade-off between the time and space needed to create a single byte array with improved indexing performance and the lower overhead of a `LeafRope`. 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 and leave this for future work.

In our evaluation we have not considered the performance of I/O operations, as our benchmarks only measure the construction of strings. We have begun to experiment with I/O operation specializations that can output a rope without flattening it, but also leave this for future work.

10. Conclusions

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 JRuby+Truffle, but we generally saw a performance range of 0.9x – 9.4x 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. The immutable nature of ropes allows us to freely share references, which makes them suitable for caching in the Truffle framework. Immutable metadata in the rope structure also provides context to the Graal optimizing compiler, unlocking optimizations that would not be available with a byte array string representation. Future work will investigate memory and garbage collection trade-offs between the two string representations.

References

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

[2] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-Level: PyPys Tracing JIT Compiler. In *Proc. of ICOOOLPS*, 2009.

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

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

[5] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1999.

[6] C. Seaton. *Specialising Dynamic Techniques for Implementing The Ruby Programming Language*. PhD thesis, The University of Manchester, 2015.

[7] C. Seaton, B. Daloze, K. Menard, P. Chalupa, et al. JRuby+Truffle, High-Performance Truffle Backend for JRuby. `https://github.com/jruby/jruby/wiki/Truffle`.

[8] 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 *Proc. of Onward!*, pages 187–204, 2013.

[9] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proc. of DLS*, page 73, 2013.