Implementing a Generic Radix Trie in Rust

Michael Sproul
University of California, Santa Cruz
University of Sydney

Introduction

Rust is a systems programming language developed by Mozilla Research that is rapidly approaching its 1.0 release [1]. Built on an imperative core with C-like syntax, Rust also includes several sophisticated features inspired by the ML family of languages. Notable features include type generics, traits (similar to Haskell's type classes), type inference, algebraic data types and pattern matching. Rust's status as a systems programming language is established by its memory management system, which guarantees memory safety without garbage collection. It achieves this through a system of ownership and borrowing inspired by the Cyclone Programming Language [11].

A trie is a tree data structure which differentiates its keys by treating them as sequences of characters from a finite alphabet. Each node of a trie contains a bucket for each alphabet character and each key-value pair is stored so that the path from the root node matches the key's character representation. In this paper, I take the term *radix trie* to mean any trie which compresses common prefixes, of which a PATRICIA trie is a variant with 2-way branching [12]. To achieve generality, my trie takes its alphabet to be the set of 4-digit binary numbers, thus allowing any type which can be serialised as a sequence of bytes to be used as a key. This yields 16-way branching – in contrast to a PATRICIA trie, which uses single binary digits as its alphabet. One further difference between the radix trie implemented and other implementations I've come across is that my trie allows values to be stored in internal nodes.

Ownership, Borrowing and Trees

Rust's ownership system [9] interacts in several interesting ways with hierarchical data-structures. The following two sections discuss my experiences with Rust's memory management, including some ownership related challenges and their solutions.

Positive Interactions

The core type used to implement the radix trie is a struct which contains an array of pointers to its children.

```
struct TrieNode<K, V> {
    key: NibbleVec,
    key_value: Option<KeyValue<K, V>>,
    children: [Option<Box<TrieNode<K, V>>>; BRANCH_FACTOR],
    child_count: usize
}
```

The `NibbleVec` type is a wrapper I wrote around a vector of bytes that behaves like an array of 4-byte values (nibbles). The `Box` type is an *owned* pointer type which allows the `TrieNode` type to be recursive without being infinitely sized. Rust's box pointers are the core point of interaction between the trie and Rust's ownership system. In some ways, the move semantics for owned pointers allowed me to treat the tree structure as I would in a garbage collected language. Children can be replaced without having to free the old child, as box pointers automatically free their contents when they are overwritten or go out of scope. Compared to the currently popular languages for systems programming, C and C++, this is a nice usability improvement and removes the possibility of leaking memory.

Furthermore, it is comforting to have the compiler guarantee the validity and uniqueness of all pointers. Uniqueness is useful because multiple internal pointers to a single node in a tree structure are never desired, and validity allows code to be written without fear of forgetting a null check. In place of null pointers, Rust's `Option<T>` type makes the possibility of non-existence explicit in the value's type, and forbids accidentally omitting non-existence checks. Option types have been present in functional languages like Haskell and ML for more than 20 years, but have seen relatively scarce adoption in mainstream languages due to the lack of type system infrastructure (ADTs and pattern matching). As a systems language, Rust strives to avoid runtime penalties for its abstractions, and to this end an `Option<Box<T>>` is optimised to a raw pointer with value `NULL (0)` if and only if its Rust value is `None` [4].

Rust's ownership system requires that the mutability of variables be defined explicitly (with immutability being the default). This enables a style of programming where all values are immutable and only made mutable as required. In contrast to C++'s `const` I found this to be very unobtrusive, with most values remaining immutable. Tracking down mutation bugs when most values are immutable is also more straight-forward.

Challenges

The primary challenge when constructing a hierarchy of owned pointers is how to sensibly and efficiently deal with back-links (or rather, the lack thereof). In a tree, all nodes are owned by their parent and no back references are permitted, as borrowing even an immutable pointer to the parent would prevent any mutation of the parent for the life of the borrow (this is the heart of Rust's borrowing system). What then should be done in the case where a child needs to update its parent? In Java or another imperative language, another approach would be to store two (mutable) references, one to the parent and one to the current node, but this too is disallowed by Rust's borrow checker. Drawing inspiration from functional languages, the solution I employed uses recursion liberally and an abstract action type that can be applied after the initial borrow expires.

```
enum DeleteAction<K, V> {
    Replace(Box<TrieNode<K, V>>),
    Delete,
    DoNothing
}
```

This action type captures the three possible operations that might need to be performed during the removal of a key from the trie. Conceptually, the removal is performed from one level above the node to be removed, with the action determined by recursively examining the child node's subtree.

```
// Recurse doum the Trie working out what to do.
let (value, delete_action) = match self.children[bucket] {
    None => (None, DoNothing),
    Some(box ref mut existing_child) => {
        // Examine the child to work out if it should be deleted...
    }
};

// Apply the computed delete action.
match delete_action {
    Replace(node) => {
        self.children[bucket] = Some(node);
        (value, DoNothing)
    }

    Delete => {
        self.take_child(bucket);

        // The removal of a child could cause this node to be replaced or deleted.
        (value, self.delete_node())
    }

    DoNothing => (value, DoNothing)
}
```

The borrow of `self.children[bucket]` expires at the end of the let statement, allowing it to be re-assigned or deleted when applying the delete action. After initially using boolean values to convey deletion information, I was pleased to settle on the action enum, as it makes the intent very clear. Rust's support for tuples can be seen here, with the first element of the tuple propagating the removed *value* up to the top. Interestingly, the call stack can be seen as providing the necessary insulation for memory safety; it ensures that only one mutable reference to a `TrieNode<K, V>` is valid at any one time. Unlike the recursive approach, using a stack data structure and a loop cannot be verified to be memory safe by the Rust compiler.

One negative side-effect of the delayed action approach is that it prevents the function from being tail-recursive, thus destroying the opportunity to perform tail call optimisation. Within the context of Rust this turns out not to be a problem, as tail call optimisation was deemed to interact badly with Rust's memory model and is unlikely to be added in the near future [2].

Parametric Mutability and Macros

Rust's strict distinction between mutable and immutable values becomes slightly awkward when writing functions that differ only in the mutability of their arguments or results. Following the design of the Rust standard data-structures I set out to provide both `get` and `get_mut` methods for my radix trie. The logic for these two methods is identical except for the sprinkling of a few `mut` keywords, which present-day Rust doesn't provide syntax to abstract over. Several Rust developers have suggested a syntax similar to the following [5]:

```
fn get<~m>(&~m self, key: K) -> Option <&~m self>;
```

Here `~m` is an imagined mutability parameter that could take the values `const` or `mut` and be used in patterns like `Some(ref ~m x)`.

To work around the lack of parametric mutability I turned to Rust's macro system, which was recently stabilised in preparation for Rust 1.0 [10]. Although generally agreed to be less than ideal, the macro system includes several nice features including hygiene and separator-based capturing of multiple items. As the two get functions differ only in the placement of the `mut` keyword, I designed the macro to take the keyword (or lack of keyword) as an argument, drawing inspiration from the approach used in `TrieMap`. The resulting macro signature looks like this:

```
macro_rules! get_function {
    (name: $name:ident, mutability: $($mut_:tt)*) => {
        // body
    }
}
```

The macro takes the function name (`$name`) as an identifier, and zero or more arbitrary *token trees* to describe the mutability. Of the current categories of parameters which can be passed to a macro, a token tree is the only kind that accepts a `mut` keyword. The two macro invocations used to generate the get functions are:

```
get_function!(name: get_mut, mutability: mut);
get_function!(name: get, mutability: );
```

The zero-or-more match is required for the immutable case where no mutability argument is passed. Inside the body of the macro, this leads to some messy patterns (a total of 7).

```
match existing_child.key_value {
    Some(ref $($mut_)* kv) => {
        check_keys(&kv.key, key);
        Some(& $($mut_)* kv.value)
    },
    None => None
}
```

A quirk in the macro system whereby token trees can only be used as macro arguments also requires the entire macro body to be wrapped in an identity macro. The resulting tangle of dollar signs and arguments broken over multiple lines is quite intimidating, but still seems better than writing and maintaining two near-identical functions. In the future I may try to replace the macro with a generic function that takes several higher-order functions as its arguments. My only fear is that this approach would needlessly increase conceptual complexity relative to shuffling `mut` keywords around.

Bugs and Testing

Although Rust's memory model removes several classes of bugs, it is naturally still possible to compile code containing logical errors. During the implementation of my radix trie I spent a lot of time debugging problems which ended up being logical errors in the supporting `NibbleVec` library, which I wrote a few months ago to support a radix trie implementation. As per the test driven development methodology, I wrote reasonably extensive unit tests and convinced myself

of the library's correctness. However, when it came to using the `NibbleVec` code in situations where numerous methods were called sequentially, things started to go awry.

The first bug I discovered was in the `split_odd` method, which is meant to split a `NibbleVec` at an odd index into two new nibble vectors. Recall that `NibbleVec` is a wrapper around a vector of bytes, Rust's `Vec<u8>` type. The bug involved the split byte not having its second half zeroed after it was copied to the tail vector [8]. Subsequent calls to push, which uses binary-or to mix in a new value, then resulted in stale data being combined with newly inserted data. This created all sorts of zany bugs in the radix trie implementation which didn't appear in any of the `NibbleVec` unit tests due to the reliance on sequential operations. I argue that this bug could have been prevented with invariant checking, or unit tests that examine the library's underlying data representation instead of just its interface. The invariant violated is one that I had in mind while writing the library, and could ideally have generated a compile-time error in a formally verified language:

$$\texttt{self.length \% 2 == 1} \rightarrow \texttt{self.data[self.data.len() - 1] \& 0x0F == 0}$$

Alternatively, a function to check the integrity of a `NibbleVec` could be written and called after each operation in the test suite. This approach fits better with the rest of the Rust language, and is similar to the approach I adopted in the radix trie library itself (see `Trie::check_integrity`).

The second bug was rather similar to the first, but involved `split_even`, which is meant to split a nibble vector at an even index. An off by 1 error in some of the arithmetic lead to the underlying data vector containing one more element than it should, resulting in corruption upon subsequent operations. The integrity/invariant checking approach described above would have caught this bug as well.

The lesson I've learnt from this is that careful testing is still required in nice languages where everything *looks* correct. Unit testing in Rust is quite pleasant thanks to the built-in testing framework and the vast number of tests that don't have to be written. Unlike C/C++, there's no need to test safe Rust code for memory corruption (although `unsafe` blocks forfeit this nicety). On the other end of the spectrum, Rust is unlike dynamically typed languages in that there's no need to test the behaviour of functions on badly typed input (for example: passing an integer to a function that expects a string in JS or Python). Furthermore, unlike Go, Java and other statically typed languages that permit null references, there's no need to write test cases involving nulls. In this respect, testing Rust code feels like testing Haskell code.

Performance

To test the performance of my implementation I benchmarked it against open source implementations written in Go and Haskell.

- **Haskell:** bytestring-trie 0.2.4, `https://hackage.haskell.org/package/bytestring-trie`
- **Go:** go-patricia 2.1.0, `https://github.com/tchap/go-patricia`

Approximately equivalent benchmarks were written in each language for each of the three main operations; get, insert and remove. Performance metrics were then recorded using two texts as input: *The Sun Rising* by John Donne and *Nineteen Eighty-Four* by George Orwell.

	Rust	**Haskell**	**Go**
get	0.113	0.115	0.061
insert	0.143	0.067	0.284
insert-remove	0.280	-	0.350
remove	0.137*	0.110	0.066*

Table 1: Results for *The Sun Rising* (226 words); times are in milliseconds.

	Rust	**Haskell**	**Go**
get	88	55	40
insert	93	409	57
insert-remove	175	-	97
remove	82*	387	38*

Table 2: Results for *Nineteen Eighty-Four* (103,366 words); times are in milliseconds.

The insert-remove benchmark for Go and Rust records the time to insert then remove each word of the entire text. This was done because both the Go and Rust tries lack deep-clone implementations. The starred (*) values indicate ones estimated by subtracting an insert benchmark from a remove benchmark (pre-rounding).

Although the Haskell library used for benchmarking (Criterion) outputs statistical information, no attempt has been made to determine statistical significance in the Rust and Go benchmarks. For this reason, these numbers serve only as rough estimates of performance. Most importantly these benchmarks indicate that my trie implementation achieves performance close to that of existing implementations. The Rust radix trie comes in second quite consistently, and with some further performance tweaks I'm confident it could challenge the Go implementation. Profiling revealed a lot of time spent in calls to `NibbleVec::split`, which could be reduced by using *slicing* instead of copying when recursing.

Further Work

There are few more things to do before the library is ready for consumption by the wider Rust community, but I'm confident I'll be able to accomplish these things in the coming months.

- Implement the "collection views" API, to allow efficient updating after node location [7].
- Improve performance by writing a slicing method for `NibbleVec`.
- Provide `TrieKey` implementations for more in-built Rust types (macros will probably be useful for the numeric types).
- Implement trie-specific methods for predecessor and successor location.
- Implement iterators.

Conclusion

Rust proved to be a very pleasant language for implementing data structures. Although some features of the language required conceptual re-adjustments, the safety guarantees provided

by the ownership system are worth the effort. As one of very few modern languages without garbage collection I hope to see Rust give C and C++ a run for their money. Rust's modern language features in combination with its robust build system should see it succeed even outside the systems programming space. Although the lack of parametric mutability makes some situations slightly awkward, for a language that hasn't reached 1.0 yet, I have very few complaints about Rust.

Acknowledgements

I'd like to thank Daniel Micay for the original implementation of `TrieMap` [3], from which I drew a lot of inspiration. I was encouraged to pursue further work on Rust tries by the helpful reviewing of Alexis Beingessner and Ben Striegel on my first major Rust pull request [6]. Other members of the Rust community, on Reddit, IRC, Github and at the San Francisco meet-ups, also helped fuel my passion for Rust. Lastly, I would like to thank Cormac Flanagan and Christopher Schuster for organising and running the wonderful course that was CMPS 112 at UCSC, which allowed me to complete this project.

Links

- Radix Trie: `https://github.com/michaelsproul/rust_radix_trie`
- Nibble Vec: `https://github.com/michaelsproul/rust_nibble_vec`
- Benchmarks: `https://github.com/michaelsproul/radix_trie_benchmarks`

References

[1] The Rust Programming Language (Homepage). `http://rust-lang.org`.

[2] Rust Issue #217 – Teach rustc to do tail calls. `https://github.com/rust-lang/rust/issues/217`, 2011.

[3] Daniel Micay's Trie Map. `https://github.com/rust-lang/rust/commit/a4d22635e1453d77e90c4335f5c2e1f62e3185eb`, 2013.

[4] Rust Issue #6001 – Represent enums with nullable pointers where possible. `https://github.com/rust-lang/rust/pull/6001`, 2013.

[5] Rust Internals – Parameterisation over mutability. `http://internals.rust-lang.org/t/parameterisation-over-mutability/235`, 2014.

[6] Rust Issue #18287 – Implement the collection views API for TrieMap. `https://github.com/rust-lang/rust/pull/18287`, 2014.

[7] Rust RFC #216 – Collection Views. `https://github.com/rust-lang/rfcs/blob/master/text/0216-collection-views.md`, 2014.

[8] NibbleVec split_odd bug fix. `https://github.com/michaelsproul/rust_nibble_vec/commit/69edc98774656ec0b44ede9b34fce0d4ddb57fbb`, 2015.

[9] Ownership, The Rust Programming Language. `http://doc.rust-lang.org/book/ownership.html`, 2015.

[10] Rust RFC #453 – Macro Reform. `https://github.com/rust-lang/rfcs/blob/master/text/0453-macro-reform.md`, 2015.

[11] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[12] Donald R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, October 1968.