Enhancing RUSTDOC to Allow Search by Types

Mihnea DOBRESCU-BALAUR, Lorina NEGREANU

University POLITEHNICA of Bucharest,
313, Splaiul Independentei, Bucharest, 060042, Romania
mihnea@linux.com; lorina.negreanu@cs.pub.ro

Abstract: Programming languages have benefited from increased attention lately. With corporations like Google, Facebook and Mozilla investing in language design, there is a lot of activity in the mainstream languages domain. State of the art features like advanced type systems that were only available to more research-centered languages such as Haskell, are now making their way into the mainstream through languages like *Rust*. *Rust* is an up-and-coming system programming language that aims to fill the same role as *C* and *C++,* but in a much safer way. To achieve this, it brings a rich type system, alongside a pragmatic implementation of region based memory management, a feature that allows safe memory handling without the need of a garbage collector. Aiming for the mainstream, *Rust* comes with tooling to aid developers in their work. One such tool is *Rustdoc*, a program that automatically generates documentation based on type information and programmer comments. We aim to enhance *Rustdoc* to provide more advanced search support through features that are impossible to implement for dynamic languages, and even static languages with less advanced type systems, like *C* and *Go*.

1. Introduction

Progress in type inference algorithms [4] has led to statically typed languages that are easier to use and more appealing to programmers. Thanks to their performance compared to dynamic languages, programming languages such as Scala, Go and Hack (Facebook's version of typed PHP) are increasingly gaining popularity. Besides the mentioned ease of use and good performance, all these languages also come with great tooling.

While not specifically part of the language, accompanying tools are critical to a programming language's success. They represent the ways in which a programmer interacts with the code. For example, any language should have (or support) a good debugger to make finding bugs easier. Also, it should have an easy to use package management and build system. Many programmers find the XML configuration files of Java package systems cumbersome. To overcome this, Scala brings a simpler format for specifying dependencies.

Another important tool is the documentation generator. Good languages are used in large projects, and large projects need great documentation. Having a standard way of generating documentation pages is key to effortless knowledge sharing. Java has *Javadoc*, Scala has *Scaladoc* and Go has *Godoc*. They all do a good job of generating formatted, Web-accessible documentation from source code. However, they share a key shortcoming - they only support searching symbols by name. Search by name covers one aspect of programming - the situation where the programmer encounters a function and wants to know what it does. But there is also the case when the programmer needs to find a function that does a particular operation, and because of various reasons (new language, new naming standards, new codebase etc.) the name of that function is unknown. In this situation, the ability to search by type would be perfect. An important benefit of static typing, besides allowing for compile-time checking, is that the types act as documentation. So why shouldn't the documentation generators use this information?

Our project aims to add this exact functionality to *Rustdoc* - the documentation generator of the up-and-coming (currently in alpha) Rust [14] programming language. Rust has a rich type system, similar to that of Haskell and ML, which allows representing complex information through the type annotations. Because of this, we consider it a great target language for searching documentation by types.

The rest of this article is structured as follows: first, we will introduce Rust, its features and look at an existing implementation of searching documentation by type for Haskell. Then, we will discuss our implementation for *Rustdoc*. Following this, we will examine the usage as well as the performance of our solution. Finally, we will present our conclusions as well as possible future work for the project.

2. Background and Related Work

2.1 Rust and its type system

Rust is a systems programming language, like *C* and *C++*. While low level languages often have simple type systems that generally only describe how much space values need in order to perform memory allocation, Rust's type system supports features such as algebraic data types and traits, capabilities that we are used to see in higher level programming languages, such as ML and Scala.

First of all, the type system provides all the basic features that one might expect from languages such as *C* and *C++* - primitive types, enums, structs, methods etc. While Rust does not have support for inheritance (like in Object Oriented Programming), it makes up for it with a similarly powerful construct: traits.

Traits [11] are a set of methods that one uses to add functionality to a struct (Rust does not have classes - structs are used instead). They can either provide only method prototypes, or full methods with default implementations. In terms of Object Oriented Programming, a trait is similar to an interface. Making a reference to the Haskell ecosystem, a trait is like a type class. Using traits, the programmer can extend their own structs with new functionality, as well as enhance the built-in types, if needed.

Another important feature of the Rust type system is having algebraic data types [5] (ADT). ADTs are compound data types, useful for representing abstract concepts without reasoning about the implementation details. A common example is representing the natural numbers using ADTs:

```
type nat :=
| Zero : nat
| Succ : nat -> nat
```

Thus, `0` would be *Zero*, `1` would be *Succ*(*Zero*), `2` would be *Succ*(*Succ*(*Zero*)) and so on.

Having these abstract representations, one can describe properties and transformations by focusing on the actual algorithm, rather than on the representation of data. A function defined on Nat will perform the same transformation regardless of the representation we choose to use for natural numbers, and this is very powerful, because ADTs are a better fit in formal proofs [8]. Thanks to the support for ADTs, the programmer is allowed to code implementations that are close to the corresponding formal specification.

To easily work with abstract data types, Rust provides pattern matching through the match expression. This construct enables more expressive control flow, in ways that the regular if and switch statements cannot achieve. In its simplest form, the syntax looks as shown in Listing 1.

```
let x = 5;
match x {
   1 => println!("one"),
   2 => println!("two"),
   3 => println!("three"),
   _ => println!("something
                  else"),
}
```

Listing 1. Match syntax in Rust

However, one can use pattern matching to interpret (destructure) abstract data types. In Listing 2, we declare the described type *Nat* and show how we can use pattern matching to interpret it as a regular, integer (32 bits) value. Note that because *Nat* as *Succ*(*Nat*) would lead to an infinite recursive expansion, we wrap the *Nat* within *Succ* inside a box (Rust's version of a smart pointer [2]).

```
enum Nat {
  Zero,
  Succ (Box<Nat>)
}

fn interpret_nat (n: Nat) -> u32
{
  match n {
     Nat::Zero => 0,
     Nat::Succ(box x) => {
            1 + interpret_nat(x)
     }
  }
}
```

Listing 2. Working with *Nat* in Rust

Having mentioned pointers, Rust's approach to memory management is novel as well. Being a systems language, it does not afford the comfort of garbage collection that other, higher level languages (such as Haskell and Scala) have. However, staying true to its *safety first* philosophy, the C approach of manual memory management via `malloc` and `free` would not be acceptable either, since it would be the developer's responsibility to not forget to call `free`, and to not call it more than once on the same pointer. To get around this, Rust implements a variant of Region Based Memory Management [3]. This is similar to what happens to variables on the stack - when they are declared, memory is reserved for them; when the function ends (they go out of scope), memory is freed. Rust takes this simple idea and extends it to heap allocated objects. This is in no way a trivial task, because, to be useful, heap-allocated objects are often returned from functions and used in other lexical scopes than the one they were created in.

To keep track of dynamic memory, Rust has a complex ownership and lifetime system that determines who owns every variable (i.e. who would be responsible for calling `free`), as well as how long is a variable supposed to live (i.e. when should free be called). This information is inferred by the compiler automatically, so the programmer does not have to do any extra work. However, if she tries to take ownership over an already owned value, the compiler's borrow checker will statically detect this error and report it. This prevents an entire class of bugs from showing up at runtime [1] [12].

By using Region Based Memory Management, Rust allows developers to write code as they would in a garbage collected language, without worrying about explicit memory allocation, while ensuring that there are no leaks and that the memory is deterministically cleared, with no global pauses (as is the case when using garbage collectors).

Besides keeping track of ownership information, the Rust compiler is also aware of the mutability status of all values. By default, everything is immutable (e.g. `let x = 5`;). However, sharing a pragmatic perspective, Rust allows mutable values (let mut x = 5;). Having this information encoded in the type of a binding, the borrow checker can perform even more advanced checks. For example, a value can be borrowed any number of times through an immutable reference (i.e. no mutation is performed). Meanwhile, once a value is borrowed through a mutable reference (i.e. mutations can be performed), it can no longer be borrowed until the first borrow ends. This static check prevents potential data races that would otherwise be difficult to debug.

2.2 Documentation generators

1) Haddock and Hoogle: Haskell [13] is a programming language that is very popular in the research community. It is very advanced and actively developed as the state of the art in programming language design progresses. Its type system is even more complex than that of Rust. However, Haskell uses garbage collection and has no borrow checker.

Haddock is the standard documentation generator for Haskell. As all documentation generators, it crawls the source code of a project and creates a formatted representation of the documentation annotations. Haddock is aware of the type system annotations as well, and it uses them to build a database of the found symbols and their types.

Hoogle [9] (taking its name from Haskell and Google) is a search engine that uses the Haddock-generated database. Using Hoogle, a user can search for `(a -> b) -> [a] -> [b]` and find the documentation of map. This is the missing piece that mainstream languages like Scala and Go lack, as we have described in the Section 1.

Hoogle is a popular tool in the Haskell community, constantly helping programmers find functions that do what they need. This saves the time of reimplementing existing functions.

2) Rustdoc: Rustdoc is Rust's documentation generator. As its host language, it is in the early stages of development (Rust is currently at its second alpha version), but it already supports the same set of features as other established generators, such as godoc.

Rustdoc generates HTML files, making the documentation easy to browse. It also supports searching symbols by name, as we will describe in the following section. Thanks to the expressivity of the Rust type system, we feel that adding by type search support to *Rustdoc* will yield great added value to the Rust community.

Table 1 shows the search capabilities of the two documentation generators, along with the more popular Javadoc.

Table 1. Search capabilities of documentation generators

	By name	By type
Javadoc	X	
Rustdoc	X	
Haddock/Hoogle	X	X

3. Building a Searchable Index Based on Rust Types

The intention of our project is to enhance *Rustdoc's* existing search functionality in order to allow more complex queries dealing with types. This need comes from a set of situations that all programmers face, especially when working in a new language or when using a new library. To give a few examples:

- Finding a helper function that checks if a character is alphanumeric.
- Finding a method that gives the length or size of a vector.
- Finding a method that returns a substring of a string.

Usually, these problems would be solved by manual, exhaustive searches in the documentation. However, in statically typed programming languages, there is a better way. Consider that the programmer knows what types she is working with - in the above examples: char, bool, Vec, usize (unsigned int), str. Having this knowledge and a type-aware documentation generator, the programmer could simply search for:

```
char -> bool
Vec -> usize
str, usize, usize -> str
```

This is exactly what our project sets out to achieve. However, because of the way that *Rustdoc* generates the documentation, this is not a trivial task. We will describe the challenges that we have faced in the next section.

3.1 Challenges

In order to make it easy for anybody to self-host documentation for their Rust library or project, *Rustdoc* generates static, HTML files. This means that in order to publicly offer documentation for a project, the author only needs to serve the HTML from a web server. There is no need for a custom framework, a database server or executing PHP or other similar server-side dynamic language. In addition the documentation can be easily made available offline: one can simply download the files locally and then one can browse the documentation even without an Internet connection.

Deploying new changes to the public is easy as well. After modifications are made in the project code, one can execute *Rustdoc* and the new HTML files will be generated. From here, publishing the updated version is only a matter of copying the files to the remote server hosting the documentation.

While having the advantages described above, the static nature of *Rustdoc* has its shortcomings as well, and they mainly show up when looking at the search problem.

Generally, the search flow obeys the following structure: there is a front-end, where a user enters a search query. The front-end sends the query to the search back-end. The back-end parses the query and matches it against a data store (generally, a database) using the data store's query language (e.g. SQL for relational databases). Then, the back-end assembles a response, which it sends to the front-end. The front-end formats the results contained by the response and the user finds the information she searched for.

Because *Rustdoc* deals with static files, there can't be a separate back-end with its own data store. Thus, we have to rely on the dynamic facilities that browsers provide in order to implement our search functionality. This means that the search engine actually has to be JavaScript code, running client-side in the user's browser. Having no database server, the data store also needs to be loaded client-side, making it available for the JavaScript code to query. The constraints described above impose the following architecture for supporting search. Generating the static files (as shown in Figure 1):

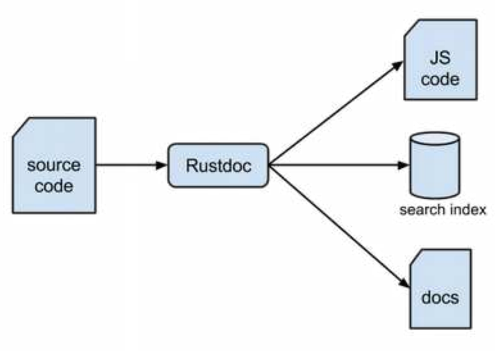


Figure 1. Generating the static files

- Walk the source code of the project.
- Generate the formatted documentation.

– For every symbol, add it to the search index.
– Write the search index to a separate file.
– Provide the JavaScript file that loads the index and processes queries

Client-side handling of the query in JavaScript (as shown in Figure 2):

– Parse the query
– Filter the symbols matching the given query
– Display the results

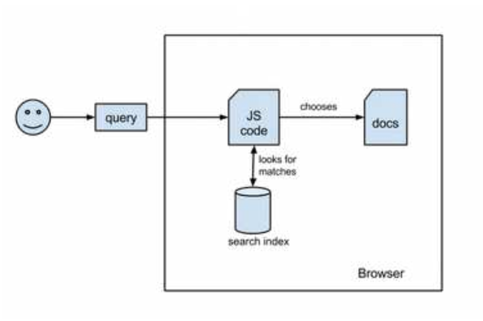


Figure2. Responding to a query

3.2 Making use of the compiler data

Rustdoc already supports searching by name and it achieves this by implementing the previously described architecture. To support searching by type, we needed to make two main additions:

1. Enrich the search index with type data.
2. Enhance the JavaScript query handler to filter by type.

This subsection describes the first part, while the following subsection will describe the second.

The current, upstream implementation of *Rustdoc* generates data that does not contain type information. It uses the compiler to identify symbols (functions, structures, methods etc.) in the project, and then it formats the annotations found on the respective declarations. To facilitate searching, it also builds a search index alongside the whole process, and then writes it to a separate file. The format of the existing search index is the following:

```
[item_type; name; path; description; parent_path]
```

`item_type` is not the full type system annotation of the symbol, but something simpler: one of `function`, `module`, `struct`, `enum`, `trait`, `typedef`. It helps narrowing down result sets by their "general" type. The other important fields are `name`, the actual name of the symbol and `description`, a short explanation of the symbol. The former is used when searching, while the latter is used when displaying results. The `path` and `parent_path` fields are not relevant for searching by type. They are useful for ranking results when searching by name, because results closer to the module that the user is exploring should be prioritized.

To implement searching by type, we needed to add another field to the index item format - the actual type of the symbol. To give an example, for the `is_alphanumeric` function, the type is `char -> bool`.

We decided to use a simple representation that can later be extended to support more advanced features that we will describe in Section 5. Listing 3 provides an example for the `is_alphanumeric` function mentioned above.

```
{
  ″inputs″ : [
      { ″name″ : ″char″ }
  ] ,
  ″output″ : {
     ″name″ : ″char″ ,
  }
}
```

Listing 3. Type representation

The choice of JSON [6] as the index format allows easy interaction with the index from JavaScript. The `inputs` key has a *list* value, allowing for multiple function arguments. The `output` key has an object value, allowing for void functions (represented as `"output": null`).

Because *Rustdoc* uses the Rust compiler to detect symbols in a project, it also has access to the type system information. This is how we extract the data in order to add the function types to the search index. The only custom aspect that we had to cover was finding the type of the `self` argument of methods. This is non-standard because in Rust, the first argument of a method (`self`) does not have an explicit type annotation, since the information would be redundant - the type is always the structure or trait to which the method belongs. However, to provide support for meaningful search queries, we had to include this type in the argument list.

Without the `self` type, searching for the `len` method of a vector would need a "`-> usize`" query. But this is too vague, since there are many functions that return an unsigned integer. Adding the type of self, one can query against "`Vec-> usize`", leading to a small, relevant set of results.

Finding the type of `self` is not trivial, because that information is not contained in the method annotations. When generating data for methods, our implementation finds the parent symbol of the method (i.e. the structure or trait that the method belongs to) and it uses that type. Because of the language's syntax, methods can only be declared within structs or traits, so the type of `self` can always be found this way. However, for traits this means that the searchable type will contain the trait's name and not the one of the struct implementing it. To give an example, `to_lowercase: CharExt -> char` is a method that the `CharExt` trait declares and `char` implements. When searching for `to_lowercase`, one would have to query for `CharExt -> char`. With the current implementation, `char -> char` would not find `to_lowercase`.

This way, Rustdoc has all the information needed to build the enhanced search index. Then, it is up to the JavaScript code to query that index.

3.3 Searching through types

Rustdoc provides a JavaScript file that executes search queries against the generated search index. Although it only supports searching by name, it is a complex piece of software:

- It allows filtering the results by general types (functions, structures etc.).
- It allows searching by exact name, in addition to matching against subsets of keywords.
- It ranks results based on a complex set of rules:
    - Levenshtein distance [7]
    - Module - results in the selected module are ranked higher.
    - Description - results with description are preferred.
    - General type - functions are ranked higher than constants.

We wanted to extend the search engine to support type search while keeping all the existing functionality. Thus, besides the existing keyword and exact match modes, our implementation adds another mode - *by type*.

Whenever the search query contains the "->" string, the search engine enters the new mode. This is safe to do because no valid Rust identifier can contain "->", so there is no interference with the existing by name search modes. Searching for non-functional types is still handled by the existing search logic. This is natural, since the only way to search for a nonfunctional type (e.g. a struct) is by its name.

Having detected a by type search query, we parse the input in order to find what type to match against. There are two main components: the argument types (the inputs) and the return type (if any). Using string manipulation operations, the respective types are identified. Then, because of the way the search index is structured, the query types are easily compared to the search index information.

To make the search more user-friendly, we ignore the order of the provided input types. To achieve this, both the provided input types and the sets of input types from the index are sorted lexicographically. This way, the user does not need to worry about the correct order of function arguments.

4. Evaluation

Having compiled *Rustdoc* with our changes, we have regenerated the documentation for the entire standard library. This is the largest set of documentation that is currently available for a Rust project, and we decided to use it to test our implementation. Measurements show that our changes do not affect the duration of generating documentation using *Rustdoc*.

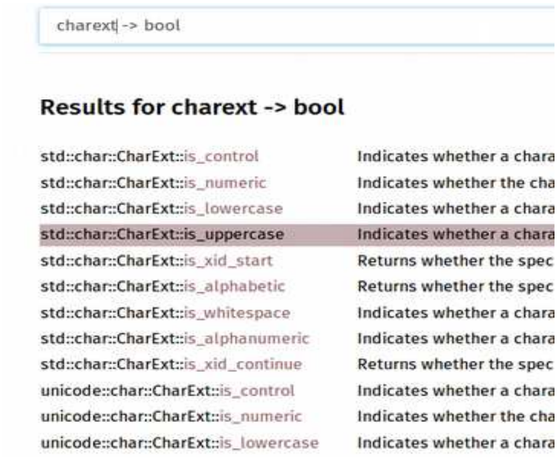
We used the same set of queries from Section 3:

```
char -> bool
Vec -> usize
str, usize, usize -> str
```

Besides making sure the results were correct, we also inspected the performance of the search code.We did this using the standard performance API [10], which provides accurate timestamps down to the millisecond.

The average response time for the queries above was 10 ms. In comparison, we found the average response time for by name queries to be 25 ms. The main reason for the difference is that the by type search has a smaller search space, since it only has to look at functions (and methods, which can also be considered functions for our purpose).

Overall, adding the new query type does not show a performance cost from a time perspective. It does, however add size to the search index, because of the new type annotations. This means that the browser has to load more data over the network. We have not found this to be an issue in our tests. However, should the need arise, compression can be used when generating the search index.



```
charext -> bool

Results for charext -> bool

std::char::CharExt::is_control         Indicates whether a chara
std::char::CharExt::is_numeric         Indicates whether the cha
std::char::CharExt::is_lowercase       Indicates whether a chara
std::char::CharExt::is_uppercase       Indicates whether a chara
std::char::CharExt::is_xid_start       Returns whether the spec
std::char::CharExt::is_alphabetic      Returns whether the spec
std::char::CharExt::is_whitespace      Indicates whether a chara
std::char::CharExt::is_alphanumeric    Indicates whether a chara
std::char::CharExt::is_xid_continue    Returns whether the spec
unicode::char::CharExt::is_control     Indicates whether a chara
unicode::char::CharExt::is_numeric     Indicates whether the cha
unicode::char::CharExt::is_lowercase   Indicates whether a chara
```

Figure 4. Screenshot of a query

5. Conclusion and Future Work

Documentation generators for statically typed languages like Rust can be very powerful. Our implementation shows that it is possible to use the type annotation that the compiler provides in order to build a search index for the types in an entire project. Then, that search index can be queried client-side, from JavaScript, without the need of a database server.

Our solution also shows that there is no performance impact for adding by type search support, other than increasing the actual size of the index.

While covering the basic needs of type search, the implementation is definitely not complete. There are some other features that can be added:

- Support for generics - `char -> char` should match `t -> t`.
- Support for trait implementors - `char -> char` should match `CharExt -> char`.
- Support for paths - the ability to either write `char` or `std::char`.
- Ranking - more concrete types should rank higher than matching generic types.
- Support for references.
- Support for lifetimes.

The design of our implementation allows for all the above features to be incrementally added. This way, after the proposed changes get merged in the upstream Rust repository, other contributors from the Open Source community can take on implementing any feature they find necessary.

REFERENCES

1. BOYAPATI, C., R. LEE, M. RINARD, **Ownership Types for Safe Programming: Preventing Data Races and Deadlocks,** in Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ser. OOPSLA '02, (New York, USA, 2002), pp. 211-230.
2. EDELSON, D., I. POHL, **Smart Pointers: They're Smart, But They're Not Pointers,** Citeseer, 1992.
3. GROSSMAN, D., D. MORRISET, T. JIM, M. W. HICKS, Y. WANG and J. CHENEY, **Region-based Memory Management in Cyclone**, in PLDI, J. Knoop and L. J. Hendren, (ed.) ACM, 2002, pp. 282-293.
4. JONES, S. P., D. VYTINIOTIS, S. WEIRICH, M. SHIELDS, **Practical Type Inference for Arbitrary-rank Types**, Journal of functional programming, vol. 17, no. 01, 2007, pp. 1-82.
5. JOUNNAUD, J.-P. and M. OKADA, **Abstract Data Type Systems**, Theoretical Computer Science, vol. 173, no. 2, 1997, pp. 349-391.
6. **Javascript Object Notation**, http://www.json.org (ac. 26th March 2015).
7. LEVENSHTEIN, V. I., **Binary Codes Capable of Correcting Deletions, Insertions, and Reversals**, Soviet physics doklady, vol. 10, no. 8, 1966, pp. 707-710.
8. LISKOV, B., S. ZILLES, **Programming with Abstract Data Types**, in Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, New York, NY, USA: ACM, 1974, pp. 50-59.
9. MITCHELL, N., **Hoogle Overview**, The Monad. Reader, vol. 12, 2008, pp. 27-35

10. **Performance webapi**, https://developer.mozilla.org/en-US/docs/Web/API/Performance/now (accessed 26th March 2015).

11. SOZEAU, M. and N. OURY, **First-class Type Classes**, in Theorem Proving in Higher Order Logics, Springer, 2008, pp. 278-293.

12. STROUSTRUP, B., **Exception Safety: Concepts and Techniques**, in A. Romanovsky, C. Dony, J. Knudsen, and A. Tripathi, (ed.), Advances in Exception Handling Techniques, ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, vol. 2022, 2001, pp. 60-76.

13. **The Haskell Programming Language**, https://www.haskell.org/ (accessed 26th March 2015)

14. **The Rust Programming Language**, https://www.rust-lang.org/ (accessed 26th March 2015)