Documentation in Julia

Adin Schmahmann
Electrical Engineering and Computer Science
MIT
Cambridge, USA
adin@mit.edu

***Abstract*—The Julia Programming language currently lacks any sort of programmatic documentation; specifically it lacks function annotation and any way to define the proper use of an abstract type, or interface. However, Julia's macros can be used to operate on pre-parsed code and can therefore both augment and define new syntax to be used in Julia. Therefore, three macros, @interface, @implements and @doc have been implemented to create interfaces and enforce their implementations, as well to annotate function declarations. Additionally, functions have been created to search through function metadata to help programmers find the functionality available in the libraries they already have access to.**

Keywords—Julia; documentation; interface; macros

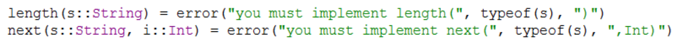
I. Introduction

Julia is a high-level, high-performance dynamic programming language (http://julialang.org/). It is syntactically similar to a number of dynamic programming languages, such as MATLAB®, Python and Lisp. Julia's features include optional typing and multiple dispatch, the ability to have multiple functions with the same name where the actual function called depends on both the function name and type signature. However, due to Julia's multiple dispatch system, there is no way to define a function as inherently related to a particular type or class, making it difficult to determine what code must be written to properly create a subtype of an existing type. Additionally, Julia does not have a built in way of associating comment text with a function, making it hard for a programmer to find if there exists a library function that can help him or her with a particular task. I have managed to use Julia's macro system, derived from Lisp, to create three macros, *@doc*, *@interface* and *@implements* to annotate functions, and declare what functions must be written to properly create a subtype of an existing type. Finally, I have created some useful sample functions for helping users search through the annotated code base.

II. Interfaces

Interfaces are an abstraction that describes how components interact with each other. In object-oriented programming languages, interfaces generally define a set of methods that need to be implemented by a given class in order for that class to be compatible with the interface. The usefulness of interfaces stems from the fact that they are very similar to types and in the same way that a variable can be of a type T, it can also implement an interface I. This means that programmers can declare a function such as *func(x::I)* that will take as an input any variable, *x*, that implements the interface I.

In Julia, the lack of interfaces means that a programmer must either look at external documentation or source code in order to determine how to use a function that operates on an abstract type. For example, *String*, is an abstract type and as a result all functions that take in arguments of type *String* implicitly rely on some definition of what a *String* is. Unfortunately, the only way to find out what is required for a type to properly sub-type *String* is by searching through source code until one finds the code segments in Figure 1.



```
length(s::String) = error("you must implement length(", typeof(s), ")")
next(s::String, i::Int) = error("you must implement next(", typeof(s), ",Int)")
```

Fig. 1. The *length(s:: SubStr)* and *next(s:: SubStr,i::Int)* functions are required to be implemented for the subtypes *SubStr* of the type *String*, but this is the only place in the code that it is documented.

As can be seen from Fig. 1, the two functions, *length(s::SubStr)* and *next(s::SubStr,i::Int)* must be defined in order for a sub-type, *SubStr*, of *String* to be truly treated as a *String*. Therefore, it would seem appropriate for there to be some documentation stating the dependence of *SubStr* on the functions *length(s::SubString)* and *next(s::SubString,i::Int)*. Additionally, since this dependence is actually a requirement for the code to be logically consistent it is reasonable that the language actually enforce this relationship – which is effectively the relationship of an interface (*String*) with another type (*SubStr*) (note: the interface name does not have to be the name of an abstract type, however once Julia implements multiple inheritance from abstract types this is the suggest method of use).

Therefore, I have implemented two macros *@interface* and *@implements* that deal with the declaration of interfaces and their enforcement. An example of the macros in use can be seen in Fig. 2. The main syntactic choice made in the macros was the decision to use * to denote the type that will be implementing the interface, all other parts of the signatures of the functions to be declared as part of the interface are as expected. In terms of the implementation, the major decision was to use an external dictionary to store all of the declared interfaces. This was chosen since there is currently no way in Julia to either augment existing types, nor to sub-type concrete types, this means that it is impossible to augment, for instance, the definition of a Type to include interface information.

```
abstract String

@interface String begin
next(s::*,i::Int)
length(s::*)
end

type UTF8String <: String
    data::Array{Uint8,1}
end

function next(s::UTF8String, i::Int)
    if !is_utf8_start(s.data[i])
        error("invalid UTF-8 character index")
    end
    trailing = _jl_utf8_trailing[s.data[i]+1]
    if length(s.data) < i + trailing
        error("premature end of UTF-8 data")
    end
    c = uint32(0)
    for j = 1:trailing
        c += s.data[i]
        c <<= 6
        i += 1
    end
    c += s.data[i]
    i += 1
    c -= _jl_utf8_offset[trailing+1]
    char(c), i
end

length(s::UTF8String) = length(s.data)
strlen(s::UTF8String) = ccall(:u8_strlen, Int, (Ptr{Uint8},), s.data)

@implements String UTF8String
```

Fig. 2. Example of how to declare the *String* abstract type/interface and how to implement it for the sub-type of *String* called *UTF8String*.

```
interface String
    nextA(s::*,a::Int)
    lengthA(s::*)
end
```

Fig. 3. How the *String* interface could be declared if interface declaration was built into Julia, note that the syntax is nearly identical to the current interface declaration syntax and analgous to the type declaration syntax.

There are a number of features that are yet to be introduced into Julia, but that are to be forthcoming that will make interface declaration and validation much cleaner. Specifically, when inheriting from multiple abstract classes is implemented then abstract types will be very similar to interfaces and could even be declared in a manner much like types (see Fig. 3). Additionally, once Julia additionally implements hooks for executing functions after a module loads then the interface validation will not require an additional line of code, but instead the interface can be validated by a programmatically inserted function hook that occurs after a module is loaded.

III. Documentation

In addition to the documentation that is implicit in the declaration of interfaces, Julia is severely lacking in methods to explicitly document code. Currently, Julia's methods of documentation include either inline comments, which the parser does not include as part of the code base, or a large "help" text file. However, having a system in place, like the "magic lines" at the beginning of a MATLAB function or the special block comment syntax before a C# or Java function declaration, would be very helpful for commenting Julia code. While the comments themselves are useful for those reading the source code, the goal is to allow the creation of tools to allow programmers to find what they are looking for in a library of source code, without having to read through all of it. In this fashion programmers can actually make full use of the abstractions provided by libraries of types and functions and focus on their own new creations.

I have made a simple macro, the *@doc* macro, which deals with annotating functions, and can easily be extended to annotate other code elements such as types. The *@doc* macro, like the *@interface* macro, also makes use of a dictionary to store its information (again, since types such as Function cannot be changed from within Julia to have additional fields). An example of how to use the *@doc* macro is given in Fig. 4.

```
@doc begin
"Add element to end of array."
end function push!(arr::Array,item)
        return push(arr,item)
    end
```

Fig. 4. Example of how to document a function

IV. Searching the Documentation

The above suggestions and associated macros include requiring relevant code to be close together via interface declaration and suggesting that functions (and other definitions, such as type definitions) have documentation near their declaration. However, one of the fundamental reasons for having different forms of documentation programmatically accessible is to allow programmatic sifting of the documentation. Below are some ways in which a programmer might want to search through the documentation code base, and these have all been implemented.

Scenarios

- Search for all functions with a particular signature
    - Ex: All functions of the form *func(a::Array,s::String)*
- Search for all functions that have a specific kind of signature
    - Ex: All functions that can use a *Float64*
- Search through all documentation for a particular clause
    - Ex: All annotated elements containing the text "SVD" or "Hash"

While these above scenarios have already been implemented in Julia, many more can be easily implemented as regular Julia functions by members of the community. Furthermore, many of these functions could be integrated into an IDE to allow for powerful code navigation that could parallel that of the IDEs of many object-oriented, statically typed, languages such as the combination of Java and Eclipse, or C# and Visual Studio.

V. Conclusion

While Julia is a very powerful and expressive language, it was lacking in documentation – especially documentation which was easily programmatically accessible. However, after implementing the *@doc*, *@interface,* and *@implements* macros this is no longer the case. Julia programmers can, and should, use the macros to annotate their code in order to make their code more easily navigable for both themselves and others. In the future hopefully the *@doc*, and *@interface* macros can be integrated more naturally into the language themselves and the *@implements* macro will at that time be able to be disposed of entirely. Additionally, some very useful sample search functions have been created to sift through the documentation made available by the macros. However, there are many more ways documentation could be searched and the accessibility of the annotations will allow any member of the Julia community to easily create a filter of his or her own.

ACKNOWLEDGMENTS

I would like to thank Prof. Alan Edelman for all of his assistance over the duration of this project. Additionally, I would like to give a tremendous thank you to Jeff Bezanson for all of his time and help discussing what improvements could be made to Julia and the best ways to proceed.