MATLAB-to-Julia Translator

Lydia A. Krasilnikova

December 23, 2013

Abstract

Some of the fields that could most benefit from parallelization primarily use programming languages that were not designed with parallel computing in mind. The MATLAB-to-Julia translator proposed here begins to approach this problem starting with MATLAB, which is syntactically close to Julia. The translator does much of the tedious work of converting source code from MATLAB to Julia, in hopes that a MATLAB user who is curious about Julia could then spend most of their first moments with the language exploring its capacity to improve their existing programs rather than wrangling with bugs or a new syntax. Hopefully with time and input from other Julia users this translator will become a powerful tool and perhaps lower the barrier to switching to Julia.

1 Background

Many fields, such as mechanical engineering, linear algebra, and computational biology, have massive potential for parallelization and a good fit with Julia's strengths. Unfortunately the existing code in these fields is already in other languages, and this on its own limits users' ability to try using Julia. There is first the problem of needing to learn and code in a new language, which is often intimidating. In addition the existing code, which could be chopped up and reused, cannot be copy-pasted into Julia like it could the original language.

There are currently no tools for translating source code into Julia from another language. The benefit of such tools could be vast: users could have more flexibility in their programming language, more users could be exposed to Julia, and fields for which Julia might be a good fit might be more likely to try it on.

2 Project Goals

The goal of this project is to build an easy-to-use tool for translating MATLAB source code into Julia. The translator does not need to be comprehensive, but it does need to be able to accurately translate enough of the most common statements that most of the tedious work of translating the code by hand is eliminated. The hope is that the user can then review the translated Julia code and perhaps make minor corrections, but be able to quickly move on to the more interesting task of parallelizing their code.

Because the focus of this project is to minimize barriers for potential Julia users, it is important that it be easy and pleasant to use. Therefore another goal of the project is to make the translator as accessible as possible, minimizing load time and barriers to access. In addition, it is important that the translated code be easy to read and that is feels like a different version of the user's own original code. It is important that the translation not introduce new bugs, if possible, and that any new bugs that are introduced be easily identified and resolved. It is also important that the personal style of the author who wrote the original code is preserved in the translation.

3 Implementation

The translator consists of two parts: the first is a front end user interface, written in Java; the second is the back end translator, written in Perl.

The Java user interface allows users to type in MATLAB source code or load it from a file and then manipulate it. When the MATLAB code is ready the user can press a button to translate the source code into Julia.

When the user initiates translation, the MATLAB source code is saved to a temporary file in the same directory as our program and control is handed over to the back end, the Perl translator script. The Perl translator script reads the temporary MATLAB file, translates it into Julia, and saves the translation to a second temporary file. Control is finally returned to the user interface, which reads in and displays the translated Julia code and deletes both of the temporary files.

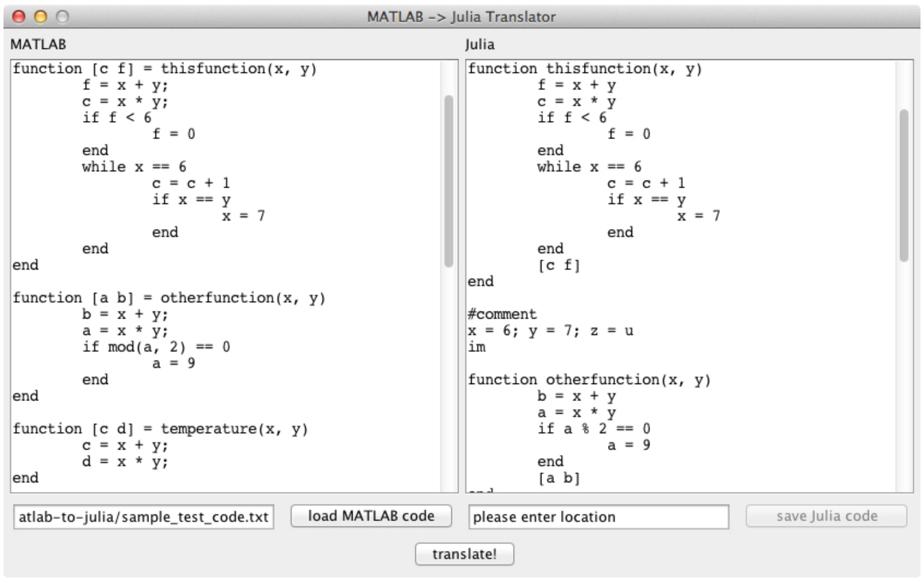
The user is then able to manipulate and save the resulting Julia code to a file, or continue editing and translating the MATLAB source code.

3.1 Front End (Java)

The front end of the translator is the user interface; its purpose is to communicate between the user and the back end. The fron end has two large text areas: the leftmost one takes in the MATLAB source code. This input can be typed in by the user directly into the text area. It can also be loaded into the program by entering the address in the text field below and pressing the "load MATLAB code" button directly to its right. (This button is disabled until the user enters a location.) The MATLAB text area will display any text that is loaded in from the input file. This text can be edited in the text area.

At the very bottom of the window is a button labelled "translate!" When the user presses this button, the interface saves the MATLAB source code that is currently in the leftmost text area to a temporary file and triggers the back end Perl script. When the Perl script is done translating and saving the translated Julia code to a file, the Java interface loads the contents of this file into the rightmost text area and deletes the temporary files.

This text area was disabled, but the translated text inside of it can now be edited and manipulated in order to correct any mistakes the translator made. The user may save the Julia code that is currently in the text area by entering a location in the field below the text area and pressing the button that says, "save Julia code."



```
function [c f] = thisfunction(x, y)
        f = x + y;
        c = x * y;
        if f < 6
                f = 0
        end
        while x == 6
                c = c + 1
                if x == y
                        x = 7
                end
        end
end

function [a b] = otherfunction(x, y)
        b = x + y;
        a = x * y;
        if mod(a, 2) == 0
                a = 9
        end
end

function [c d] = temperature(x, y)
        c = x + y;
        d = x * y;
end
```

```
function thisfunction(x, y)
        f = x + y
        c = x * y
        if f < 6
                f = 0
        end
        while x == 6
                c = c + 1
                if x == y
                        x = 7
                end
        end
        [c f]
end

#comment
x = 6; y = 7; z = u
im

function otherfunction(x, y)
        b = x + y
        a = x * y
        if a % 2 == 0
                a = 9
        end
        [a b]
```

Figure 1: A screenshot of the translator interface.

The user can continue editing both the MATLAB and the Julia code. Whenever a new translation is performed, the translated text replaces the previous contents of the Julia text area. Similarly, when new MATLAB code is loaded from a file, it replaces the previous MATLAB code and any changes the user may have made.

The front end Java interface waits for the Perl script to complete its job from a background thread. This allows the interface to remain responsive even while the Perl script is running. However, any changes the user might make would not be included in the current translation; therefore the fields and buttons of the interface are disabled until the Perl script is done running, and the background color of the interface changes to a dark grey. The interface is similarly disabled when there are error messages to the user.

3.2 Back End (Perl)

The Perl back end reads in the user's MATLAB code from a temporary file that was saved by the Java user interface. The back end's primary job is to translate the MATLAB source code into Julia. It then saves the Julia code to another temporary file.

Because MATLAB and Julia are syntactically very similar, most statements can be translated using regular expressions. When regular expressions fit a problem well, Perl can be a very powerful solution. In our case many of the statements are translated using just one line of code. Unfortunately the exceptions to this rule are much longer.

The lines translating each type of statement are separated from each other. Translation capability can be added simply by adding more lines of regular expressions between the existing groups of expressions. Because the front end and back end are separate, the back end can be edited and tested without restarting the front end or reloading the test code.

3.3 Supported Differences Between MATLAB and Julia

The following are the differences between MATLAB and Julia that the translator can currently translate.

	MATLAB code	Julia translation
COMMENTS	% comment	# comment

BLOCK COMMENTS	%{
	comment
	%}
		#
		# comment
		#

SEMICOLONS	a = 1 + 2; b = 4;	a = 1 + 2; b = 4
COMMAS	a = 1 + 2, b = 4,	a = 1 + 2; b = 4
QUOTES	‘hello’ ‘hi’ ‘h’	“hello” “hi” ‘h’

MODULUS	mod(a, b)
	mod(a + b, c + d)
		a % b
		(a + b) % (c + d)

BITWISE XOR	bitxor(a, b)
	bitxor(a + b, c + d)
		a $ b
		(a + b) $ (c + d)

BITWISE AND	bitand(a, b)
	bitand(a + b, c + d)
		a & b
		(a + b) & (c + d)

BITWISE OR	bitor(a, b)
	bitor(a + b, c + d)
		a | b
		(a + b) | (c + d)

IMAGINARY UNIT	sqrt(-1)	im
FORMATTED PRINTING	fprintf(‘%d + %.2f’, a, b)	@sprintf(“%d + %.2f”, a, b)

2D PLOTTING (example from MathWorks.com)	x = 0:0.05:5;
	y = sin(x.^2);
	plot(x, y);
	xlabel(‘Time’)
	ylabel(‘Amplitude’)
		using PyPlot
		x=0:0.05:5
		y=sin(x.^2)
		plot(x, y)
		xlabel(“Time”)
		ylabel(“Amplitude”)

	MATLAB code	Julia translation
In-Line Functions	h = @(x, y) x * y	h(x, y) = x * y
Functions	function [a b] = smprd(x, y)	function smprd(x, y)
	a = x + y;	a = x + y;
	b = x * y;	b = x * y;
		[a b]
	end (optional)	end
Nested Functions And Loops	function [a b] = smprd(x, y)	function smprd(x, y)
	while x <y	while x <y
	x = x + 2	x = x + 2
	end	end
	function [c] = fun(w, z)	function fun(w, z)
	c = mod(w, z)	c = w % z
		[c]
	end	end
	a = x + y;	a = x + y;
	if a == 5	if a == 5
	a = 0	a = 0
	end	end
	b = x * y;	b = x * y;
		[a b]
	end (optional)	end
Function Calls Without Parameters	f	f()
Array Indexing	M(a, b)	M[a, b]

3.4 Running the Program

The Java (.java) source code and the Perl (.pl) source code must be located in the same directory. If you are running the translator for the first time or after editing the Java source code, enter into the terminal `javac TranslatorGUI.java` from the directory in which they are located to build the class file. Enter `java TranslatorGUI` to launch the program. The Perl script will be run by the Java interface that has just been launched. Note that the Java interface and the Perl script will be saving and deleting temporary files within the folder that contains the source code.

If for some reason the Perl script freezes, the Java interface may continue to wait for it until it is terminated. Terminate the Perl script; you do not need to close or restart the Java interface.

4 Future Directions

I have posted the source code on GitHub. My hope is that other Julia users will find this project as interesting as I have and help to make it a truly powerful tool.

A good place to start improving the translator might be the list of noteworthy differences from MATLAB in the Julia docs. Some particularly useful areas for improvement include broadening the supported plotting functions, developing a more intelligent differentiation between matrix indexing and function calls, and incorporating the small differences between how the two languages process matrices.

It might also be interesting to compile a list of common statements and tools in MATLAB that do not translate to Julia and which other people might find interesting and implement.

4.1 Matrix Indexing

MATLAB unfortunately uses parentheses both to access elements of a matrix and to pass parameters to a function. Julia, on the other hand, uses parentheses to pass parameters to a function and brackets for matrix indexing. It is important that the translator be able to differentiate between function calls and matrix indexing and correctly assign brackets or parentheses. At the moment the translator locates function definitions and initialization of possible matrices in all of the surrounding code.

It would be helpful to incorporate the scope of the matrices and functions into the translator's decision, as well as the order in which the definitions appear. This would help deal with situations in which a function and a matrix might share a name.

It would also be helpful to collect a list of names of functions that are available to the program without having to be defined in the source code that is being translated, as well as those functions that output a matrix.

4.2 Plotting Functions

The plotting packages and functions are some of the most powerful tools in MATLAB. Currently the translator only supports 2D plotting. It would be wonderful to expand its reach to MATLAB's other plotting capabilities wherever they also exist in Julia.

4.3 Online Front End

In addition to improving the back end of the program, a lot of growth can be had by working on the front end. The primary goal of this program is to break apart some of the barriers to switching from MATLAB to Julia. The fewer barriers there are to running the program itself, to closer we can get to that goal.

My vision for this program is that it will be online, embedded in a web site, and that running it will not require the user to download anything. One way to achieve this is to upload the existing Java class file to a web site, either as a pop-up window or an applet. Unfortunately the time that Java programs usually take to load might be prohibitive. The wait time could become another barrier.

An alternative is to translate the Java interface into JavaScript. I have very little experience with JavaScript or with running Perl scripts online, but I hope that with generous help and time it could meld more seamlessly with the browser window and take fewer steps to run.

Once the front end has been revised and put online, it would also be helpful to the user if the program were to suggest code in the Julia translation that could be parallelized and link to appropriate Julia docs explaining how to do it. In addition, it would be useful if the program could highlight areas of code that either could not be translated at all or that could not be translated with confidence.

4.4 Other Languages

A broader, more long-term extension of this project could be to expand it to languages other than MATLAB. While MATLAB is very popular in some fields, such as mechanical engineering, some of the fields that are most desperate for parallelization use languages that are very syntactically different from Julia. Computational biology, for example, runs almost entirely on Perl. In this case the field is locked into Perl largely out of habit and inertia. While Perl is wonderful at pattern matching, which is a large part of the programming that computational biologists do, it is probably not the best language for sequencing or analyzing the human genome. However, Perl comes with many years of code that has already been written in the lab and which can very easily be chopped up and repurposed to save time. If these snippets of existing code could quickly be translated into Julia, it would be much more likely that computational biologists would be able to make the switch.

Translating into Julia from a language that is very different from it, like Perl, would be much more difficult than translating into Julia from MATLAB. In this particular case I was able to achieve a lot with regular expressions, but I think that as the source language gets farther from Julia, regular expressions will be able to translate less and less of the code. A better solution might be to use existing parser tools to translate the source code into an abstract syntax tree that can then be translated into Julia. One such tool is Java's ANTLR. If the front-end is kept in Java, ANTLR could be used to write the back-end. Unlike the current Perl script, this set-up would natually lend itself to adding new source languages, and it would be comparatively easy to continue to expand it.

5 Resources and Acknowledgements

This project gave me an exciting and unique opportunity to become closer acquainted with Julia, MATLAB, Perl, Java, and LaTeX.

I would like to thank Dr. Alan Edelman and Jeff Bezanson for teaching me about Julia and facilitating this project. I would also like to thank the Fall 2013 Parallel Computing class for their helpful suggestions toward the later stages of this part of the project, during my presentation.

I also found the following resources particularly helpful:

- The Julia docs have very detailed explanations in addition to code examples. I was able to get a good grasp of bigger pictures in addition to smaller details when I needed them.
- Similarly, the MathWorks website was extremely helpful for understanding MATLAB syntax through diverse code examples.
- I used the Eclipse Java IDE for writing the front end of the program.
- I used TextWrangler to write the Perl back end.
- **write**LaTeX greatly simplified LaTeX document creation and editing during the process of writing this paper.

Finally, thank you to anyone who contributes to this project in the future and helps it grow and improve. I am excited to see where you take it and I'm glad to have your invaluable help.