A Meta-Circular Evaluator for Julia

António Menezes Leitão

March, 2024

1 Introduction

A meta-circular evaluator is an excellent tool for experimentation in language design. In its simplest form, a meta-circular evaluator for language L is a program written in language L that takes an expression written in language L and computes its meaning.

During the Advanced Programming course, we studied the implementation of a meta-circular evaluator for the programming language Scheme, which we then easily extended to experiment with different evaluation rules, including dynamic scope, lexical scope, short-circuit evaluation, and non-deterministic evaluation, among others.

2 Goals

The goal of this project is to develop a meta-circular evaluator for a subset of the Julia programming language named MetaJulia. More specifically, the evaluator needs to support the evaluation of untyped but correct Julia programs, i.e., Julia programs that do not include type declarations and that do not throw exceptions, with all the syntatical forms demonstrated below.

You must also implement a Julia function named `metajulia_repl` that starts a Julia's REPL (with a `>>` prompt) that allows testing of the evaluator. To simplify your task, you might want to use the `Meta.parse` function but, obviously, you cannot use Julia's predefined `eval` function.

Below, we present examples of evaluations using the MetaJulia language that you might want to use as tests of your own evaluator.

```
julia> metajulia_repl()
>> 1
1
>> "Hello, World!"
"Hello, World!"
>> 1 + 2
3
>> (2 + 3)*(4 + 5)
45
>> 3 > 2
true
>> 3 < 2
false
>> 3 > 2 && 3 < 2
false
>> 3 > 2 || 3 < 2
true
```

Boolean values are useful in conditional expressions, which happen to have two different syntaxes:

```
>> 3 > 2 ? 1 : 0
1
>> 3 < 2 ? 1 : 0
0
>> if 3 > 2
     1
   else
     0
   end
1
>> if 3 < 2
     1
   elseif 2 > 3
     2
   else
     0
   end
0
```

The language also supports blocks, again, with two different syntaxes:

```
>> (1+2; 2*3; 3/4)
0.75
>> begin 1+2; 2*3; 3/4 end
0.75
```

MetaJulia supports the `let` form, with lexical scope:

```
>> let x = 1; x end
1
>> let x = 2; x*3 end
6
>> let a = 1, b = 2; let a = 3; a+b end end
5
>> let a = 1
     a + 2
   end
3
```

The `let` form also allows function definitions:

```
>> let x(y) = y+1; x(1) end
2
>> let x(y,z) = y+z; x(1,2) end
3
>> let x = 1, y(x) = x+1; y(x+1) end
3
```

MetaJulia also supports assignments, which implicitly creates definitions if necessary:

```
>> x = 1+2
3
>> x+2
5
>> triple(a) = a + a + a
<function>
>> triple(x+3)
18
>> baz = 3
3
>> let x = 0
     baz = 5
   end + baz
8
>> let ; baz = 6 end + baz
9
```

MetaJulia also deals with higher-order functions:

```
>> sum(f, a, b) =
     a > b ?
       0 :
       f(a) + sum(f, a + 1, b)
<function>
>> sum(triple, 1, 10)
165
```

Obviously, it also supports anonymous functions:

```
>> (x -> x + 1)(2)
3
>> (() -> 5)()
5
>> ((x, y) -> x + y)(1, 2)
3
>> sum(x -> x*x, 1, 10)
385
>> incr =
     let priv_counter = 0
       () -> priv_counter = priv_counter + 1
     end
<function>
>> incr()
1
>> incr()
2
>> incr()
3
```

Special care must be taken to support the following semantics, which illustrates the use of the `global` keyword to create (or update) binding on the global scope:

```
>> let secret = 1234; global show_secret() = secret end
<function>
>> show_secret()
1234
>> let priv_balance = 0
     global deposit = quantity -> priv_balance = priv_balance + quantity
     global withdraw = quantity -> priv_balance = priv_balance - quantity
   end
<function>
>> deposit(200)
200
>> withdraw(50)
150
```

Logical operators have the typical short-circuit evaluation semantics:

```
>> quotient_or_false(a, b) = !(b == 0) && a/b
<function>
>> quotient_or_false(6, 2)
3.0
>> quotient_or_false(6, 0)
false
```

So far, MetaJulia is a strict subset of Julia. What makes MetaJulia different from Julia (and from most other languages) is its different take on reflection and metaprogramming.

2.1 Reflection

The `eval` function is usually considered an important feature in languages that have reflective capabilities. This function takes an expression as argument and computes its value. This means that the use of `eval` requires the ability to reify the expression that will be evaluated.

Like Julia, MetaJulia has native support for reifying expressions. That is achieved using the *quote* form and its support for interpolation, as demonstrated in the following examples:

```
>> :foo
:foo
>> :(foo + bar)
:(foo + bar)
>> :((1 + 2) * $(1 + 2))
:((1 + 2) * 3)
```

Regarding the implementation of `eval`, it is important to note that it is difficult to support `eval` in lexically scoped languages, as the evaluation of an expression typicaly depends on free variables that are not accessible from the scope where `eval` is being used. There are different ways to solve this problem: one is to restrict `eval` so that it can only use the global scope, as this scope is always accessible, and another is to also reify scopes, so that we can explicitly provide the `eval` function with the scope where a given expression should be evaluated.

MetaJulia does not use any of these approaches. Instead, MetaJulia only makes `eval` accessible inside a *fexpr*. A *fexpr* is a function that does not evaluate its arguments, meaning that, when called, the function receives the actual expressions that were provided as arguments. In MetaJulia, a *fexpr* is defined just like a normal function but using the `:=` operator instead of the `=` operator used to define functions. Here is an example that compares a function with a *fexpr*:

```
>> identity_function(x) = x
 <function>
 >> identity_function(1+2)
 3
 >> identity_fexpr(x) := x
 <fexpr>
 >> identity_fexpr(1+2)
 :(1 + 2)
 >> identity_fexpr(1+2) == :(1+2)
 true
```

As can be seen in the final expression, instead of receiving the value of the expression `1 + 2` as argument, `identity_fexpr` received the actual expression `1 + 2` as argument and then returned it.

Just like it happens in languages that support `eval`, when a *fexpr* wants to evaluate an expression, it can use the `eval` function, which is lexically bound inside the *fexpr*. This behavior can be seen in the following interaction:

```
>> debug(expr) :=
     let r = eval(expr)
       println(expr, " => ", r)
       r
     end
<fexpr>
>> let x = 1
     1 + debug(x + 1)
   end
x + 1 => 2
3
```

Note an important detail in the use of `eval` by a *fexpr*: its argument is evaluated in the scope of the call to the *fexpr* and not in the scope where the *fexpr* was defined nor in the scope of the body of the *fexpr* itself. Here is an example that demonstrates this:

```
>> let a = 1
     global puzzle(x) :=
       let b = 2
         eval(x) + a + b
       end
   end
<fexpr>
>> let a = 3, b = 4
     puzzle(a + b)
   end
10
```

For a more extreme example, consider the following:

```
>> let eval = 123
     puzzle(eval)
   end
126
```

Historically, the *fexpr* idea became popular because it allowed programmers to extend the programming language with new control structures that were harder to achieve using functions. For example:

```
>> when(condition, action) := eval(condition) ? eval(action) : false
<fexpr>
>> show_sign(n) =
     begin
       when(n > 0, println("Positive"))
       when(n < 0, println("Negative"))
       n
     end
<function>
>> show_sign(3)
Positive
3
>> show_sign(-3)
Negative
-3
>> show_sign(0)
0
```

For a more sophisticated example, consider:

```
>> repeat_until(condition, action) :=
     let ;
       loop() = (eval(action); eval(condition) ? false : loop())
       loop()
     end
<fexpr>
>> let n = 4
     repeat_until(n == 0, (println(n); n = n - 1))
   end
4
3
2
1
false
```

Using *fexprs* and `eval` allows us to access local scopes without requiring their reification, as demonstrated by the following example:

```
>> mystery() := eval
<fexpr>
>> let a = 1, b = 2
     global eval_here = mystery()
   end
<function>
>> let a = 3, b = 4
     global eval_there = mystery()
   end
<function>
>> eval_here(:(a + b)) + eval_there(:(a + b))
10
```

Despite all their expressiveness, *fexprs* made compilation very difficult and they ended up being replaced by *macros*, a metaprogramming-based approach that, despite being harder to master, allowed compilers to generate very efficient code.

2.2 Metaprogramming

MetaJulia also supports metaprogramming through the use of *macros*. Differently from Julia, these *macros* are not defined using Julia's `macro` form and they are not called using Julia's `@` syntax. Instead, they look like functions but are defined using the `$=` operator. Here is an example:

```
>> when(condition, action) $= :($condition ? $action : false)
<macro>
```

As an example of use, consider the following definition of the `abs` function written in an imperative style, using `when` and assignment:

```
>> abs(x) = (when(x < 0, (x = -x;)); x)
<function>
>> abs(-5)
5
>> abs(5)
5
```

As *macros* can replace most of the uses of *fexprs* and `eval`, it is not surprising to verify that they can implement `repeat_until`:

```
>> repeat_until(condition, action) $=
     :(let ;
         loop() = ($action; $condition ? false : loop())
         loop()
       end)
<macro>
>> let n = 4
     repeat_until(n == 0, (println(n); n = n - 1))
   end
4
3
2
1
false
```

It is important to note that the previous *macro* definition is not entirely correct, as is visible in the following example:

```
let loop = "I'm looping!", i = 3
     repeat_until(i == 0, (println(loop); i = i - 1))
   end
<function>
<function>
<function>
false
```

This strange behavior is the result of a name conflict involving the `loop` local variable in the above expression, and the `loop` local function that results from the macro expansion, which shadows the previous variable. To fix the problem, we need to use the `gensym` function, which generates a symbol that is unique, in the sense that it does not clash with any other symbol:

```
>> repeat_until(condition, action) $=
     let loop = gensym()
       :(let ;
           $loop() = ($action; $condition ? false : $loop())
           $loop()
         end)
     end
<macro>
>> let loop = "I'm looping!", i = 3
     repeat_until(i == 0, (println(loop); i = i - 1))
   end
I'm looping!
I'm looping!
I'm looping!
false
```

2.3 Testing

In order to facilitate the evaluation of your project, you need to define a Julia function named `metajulia_eval` that takes one MetaJulia expression and **returns** its value. Depending on your implementation, this might be the function that you use in your MetaJulia REPL or it might be a different function. The goal is that this function is callable from Julia as follows:

```
julia> metajulia_eval(:(1 + 2))
3

julia> metajulia_eval(:(foo(x) = x + 1))
<function>

julia> metajulia_eval(:(foo(2)))
3

julia> metajulia_eval(:(foo(2))) == 3
true
```

2.4 Extensions

You can extend your project to further increase your final grade. Note that this increase will not exceed **two** points.

Examples of interesting extensions include:

- Type declarations (`struct`, `abstract type`, etc).
- Methods.
- Exceptions (`throw`, `try-catch`, etc).

Be careful when implementing extensions, so that the extra functionality does not compromise the functionality asked in the previous sections.

3 Code

Your implementation must work in Julia 1.10 but you can also use more recent versions.

The written code should have the best possible style, should allow easy reading, and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided into functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

4 Presentation

For this project, a full report is not required. Instead, a presentation is required. This presentation should be prepared for a 10-minute slot, should be centered on the architectural decisions taken, and may include all the details that you consider relevant. You should be able to "sell" your solution to your colleagues and teachers.

Note that the presentation needs to be recorded and the recording must be submitted along with the presentation slides.

5 Format

Each project must be submitted by electronic means using the Fénix Portal. Each student must submit a single compressed file in ZIP format, named `project.zip`. Decompressing this ZIP file must generate a folder containing:

- The source code, within subdirectory `/src/`.
- The presentation, in a file named `presentation.pdf`.
- The recording of the presentation, in a file named `recording.mp4`.

The only accepted format for the presentation slides is PDF. This PDF file must be named `presentation.pdf`. The only accepted format for the recording of the presentation is MPEG-4. This MPEG-4 file must be named `recording.mp4`.

6 Evaluation

The evaluation criteria include:

- The completeness of the developed programs (60%).
- The quality of the developed programs (20%).
- The quality of the presentation (20%).

In case of doubt, the teacher might request explanations about the inner workings of the developed project, including demonstrations.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues (or ChatGPT) as long as the proper attribution is provided.

This course has very strict rules regarding plagiarism. Any project where plagiarism is detected will receive a grade of zero.

These rules should not prevent the healthy exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code must be submitted via Fénix, no later than 23:00 of **March**, **27**. Similarly, the presentation must be submitted via Fénix, no later than 23:00 of **March**, **27**.