A Meta-Circular Julia Evaluator

Group 14:

99172 - Adriana Nunes
99219 - Francisco Carvalho
99259 - José João Ferreira
99336 - Tiago Quinteiro

Introduction

In this presentation, the topics we're going to discuss are:

- General Architecture (Adriana)
- Evaluation (José)
- Reflection (Tiago)
- Metaprogramming (Francisco)
- Testing (Adriana)

General architecture

`Environment` - logic about the scopes

```
module Environment

export newEnv, addEnvBinding, hasEnvBinding,
getEnvBinding, createGenSym, addEnvBindingSym,
getEnvBindingSym

"""Creates a new environment inside the one passed as
argument"""
function newEnv(outer::Dict{String,Any})···
end

"""Adds or replaces a binding to a name in the
environment"""
function addEnvBinding(env::Dict, name::String, node::Any)·
end

"""Checks if a name is bound in the environment"""
function hasEnvBinding(env::Dict, name::String)···
end

"""Retrieves the node bound to a name in the environment"""
function getEnvBinding(env::Dict, name::String)···
end

"""Creates a new symbolic binding in the environment"""
function createGenSym(env::Dict, originalName::String,
newName::String)···
end
```

`MetaJuliaRepl` - meta-circular evaluator

```
module MetaJuliaREPL

include("Environment.jl")
using .Environment

# ------------------------------------------------
# Data Structures
# ------------------------------------------------
struct Function···
end

struct Fexpr···
end

struct Macro···
end

function Base.show(io::IO, x::Function)···
end

function Base.show(io::IO, x::Fexpr)···
end

function Base.show(io::IO, x::Macro)···
end


# ------------------------------------------------
# DEBUG functions
```

```
function parse_input()
    input = ""
    while input == ""
        print(">> ")
        input = readline()
    end
    node = Meta.parse(input)
    while node isa Expr && node.head == :incomplete
        print("   ")
        input *= "\n" * readline()
        node = Meta.parse(input)
    end
    Base.remove_linenums!(node)
    return node
end
function metajulia_repl()
    while true
        node = parse_input()
        DEBUG_NODE && debug_node(node)

        if node isa Expr || node isa Number || node isa String || node isa Symbol || node isa QuoteNode ⋯
        else
            error("Unsupported node type: $(typeof(node))")
        end
    end
end
```

Abstract Syntax Tree node

Multilining

Remove `LineNumberNode`'s

```julia
function metajulia_repl()
    while true
        node = parse_input()
        DEBUG_NODE && debug_node(node)

        if node isa Expr || node isa Number || node isa String || node isa Symbol || node isa QuoteNode

            # Evaluate the node and print the result
            result = evaluate(node)

            # no printing for null values
            if isnothing(result)
                continue
            end

            result isa String ? println("\"$(result)\"") : println(result)

        else
            error("Unsupported node type: $(typeof(node))")
        end
    end
end
```

It will use the

```julia
# global scope
global_env = Dict{String,Any}("#" => nothing)
```

Evaluation

- using a higher-order function as example, we look at what's happening when we call `Meta.parse`

Printing the AST Node

```
julia> metajulia_repl()
>> sum(f, a, b) =
   a > b ?
   0 :
   f(a) + sum(f, a + 1, b)
Any[:(sum(f, a, b)), quote
    #= none:1 =#
    if a > b
        0
    else
        f(a) + sum(f, a + 1, b)
    end
end]
```

Evaluation

Node Structure

```
julia> metajulia_repl()
>> sum(f, a, b) =
   a > b ?
   0 :
   f(a) + sum(f, a + 1, b)
[NODE]└── Expr(=): [sum(f, a, b), begin if a > b 0 else f(a) + sum(f, a + 1, b) end end]
[NODE]    ├── Expr(call): [sum, f, a, b]
[NODE]    │   ├── Symbol: sum
[NODE]    │   ├── Symbol: f
[NODE]    │   ├── Symbol: a
[NODE]    │   └── Symbol: b
[NODE]    └── Expr(block): [if a > b 0 else f(a) + sum(f, a + 1, b) end]
[NODE]        └── Expr(if): [a > b, 0, f(a) + sum(f, a + 1, b)]
[NODE]            ├── Expr(call): [>, a, b]
[NODE]            │   ├── Symbol: >
[NODE]            │   ├── Symbol: a
[NODE]            │   └── Symbol: b
[NODE]            ├── Int64: 0
[NODE]            └── Expr(call): [+, f(a), sum(f, a + 1, b)]
[NODE]                ├── Symbol: +
[NODE]                ├── Expr(call): [f, a]
[NODE]                │   ├── Symbol: f
[NODE]                │   └── Symbol: a
[NODE]                └── Expr(call): [sum, f, a + 1, b]
[NODE]                    ├── Symbol: sum
[NODE]                    ├── Symbol: f
[NODE]                    ├── Expr(call): [+, a, 1]
[NODE]                    │   ├── Symbol: +
[NODE]                    │   ├── Symbol: a
[NODE]                    │   └── Int64: 1
[NODE]                    └── Symbol: b
<function>
>>
```

Environment Structure

```
julia> metajulia_repl()
>> sum(f, a, b) =
   a > b ?
   0 :
   f(a) + sum(f, a + 1, b)
[ENV] sum => (f, a, b)->(quote if a > b 0 else f(a) + sum(f, a + 1, b) end end)
[ENV] # => nothing
<function>
>> triple(a) = a + a + a
[ENV] sum => (f, a, b)->(quote if a > b 0 else f(a) + sum(f, a + 1, b) end end)
[ENV] triple => (a)->(quote a + a + a end)
[ENV] # => nothing
<function>
>> sum(triple, 1, 10)
[ENV] f => (a)->(quote a + a + a end)
[ENV] b => 10
[ENV] # =>
      [ENV] sum => (f, a, b)->(quote if a > b 0 else f(a) + sum(f, a + 1, b) end end)
      [ENV] triple => (a)->(quote a + a + a end)
      [ENV] # => nothing
[ENV] a => 1
```

Evaluation

- numbers
- strings
- symbols
- expressions:
  - function calls:
    - +, -, *,^, /, %, <, <=, ==, >=, >, !=, !
    - defined functions
    - defined functions with captured environment:
      - global functions inside let blocks
      - functions with let blocks inside
  - ||, &&
  - if, elseif
  - block
  - let
  - global
  - ->
  - =

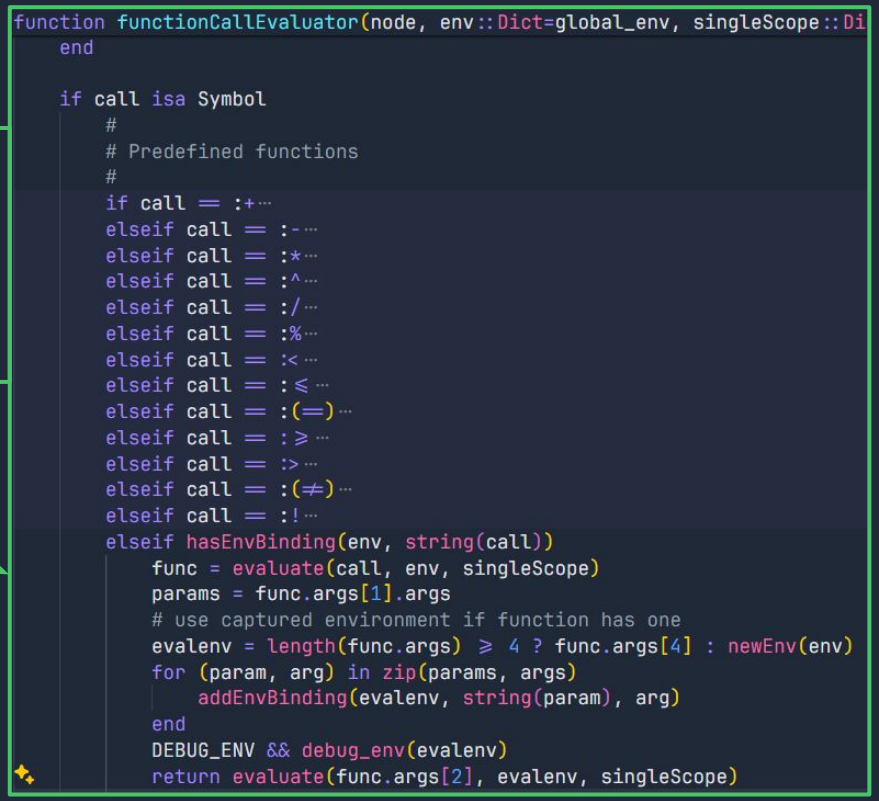
```
"""Main evaluator"""
function evaluate(node, env::Dict=global_env, singleScope::Dict
    if node isa Number || node isa String # Literals
        return node

    elseif node isa Symbol # Variables
        name = string(node)
        result = getEnvBinding(env, name)
        return result


    elseif node isa Expr # Expressions
```

Evaluation

- numbers
- strings
- symbols
- expressions:
  - function calls:
    - +, -, *,^, /, %, <, <=, ==, >=, >, !=, !
    - defined functions
    - defined functions with captured environment:
      - global functions inside let blocks
      - functions with let blocks inside
  - ||, &&
  - if, elseif
  - block
  - let
  - global
  - ->
  - =



Evaluation

- numbers
- strings
- symbols
- expressions:
  - function calls:
    - +, -, *,^, /, %, <, <=, ==, >=, >, !=, !
    - defined functions
    - defined functions with captured environment:
      - global functions inside let blocks
      - functions with let blocks inside
  - ||, &&
  - if, elseif
  - block
  - let
  - global
  - ->
  - =

```
>> 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
```

```
>> incr =
     let priv_counter = 0
       () -> priv_counter = priv_counter + 1
     end
<function>
>> incr()
1
>> incr()
2
>> incr()
3
```

Evaluation

- numbers
- strings
- symbols
- expressions:
  - function calls:
    - +, -, *,^, /, %, <, <=, ==, >=, >, !=, !
    - defined functions
    - defined functions with captured environment:
      - global functions inside let blocks
      - functions with let blocks inside
  - ||, &&
  - if, elseif
  - block
  - let
  - global
  - ->
  - =

```
"""Main evaluator"""
function evaluate(node, env::Dict=global_env, singleScope::Dict=Dict{Stri
    if node isa Number || node isa String # Literals…
    elseif node isa Symbol # Variables…
    elseif node isa QuoteNode # Reflection…
    elseif node isa Expr # Expressions
        if node.head == :call  # Function calls…
        elseif node.head == :|| …
        elseif node.head == :&& …
        elseif node.head == :if …
        elseif node.head == :elseif …
        elseif node.head == :block …
        elseif node.head == :let …
        elseif node.head == :global …
        elseif node.head == :-> # Anonymous functions passed as arguments
        elseif node.head == :(=) # Assignments…
```

Reflection

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

Isn’t an expression so the evaluator returns what was given by the parser

We need to evaluate expressions in greater detail because of the possibility of the $ operator appearing

Reflection

Reflect() recursively breaks down the expression passed into a String

If the $ operator comes up, if firstly evaluates that expression and then adds it to the string

After the reflection is done, we parse the returned string through Julia's native Meta.parser and return it as a Quote

```
"""Used for reflection to avoid evaluation of calls"""
function reflect(node, env::Dict, singleScope::Dict)
    # creates a string to be evaluated by the parser
    if node isa Symbol || node isa Number
        return string(node)

    elseif node isa String
        return "\"$(string(node))\""

    elseif node.head == :($)
        # evaluates the Expr inside $() and adds it to the string
        return string(evaluate(node.args[1], env, singleScope))

    elseif node isa Expr && node.head == :block
        final = string() * "("
        len = size(node.args, 1)
        for i in 1:len
            final = final * reflect(node.args[i], env, singleScope)
            if i == len
                break
            end
            final = final * ";"
        end
        final = final * ")"
        return final
```

```
value = Meta.parse(reflect(node.args[1], env, singleScope))
Base.remove_linenums!(value)

return Expr(:quote, value)
```

Reflection

```
"""This function evaluates the definition of fexpr"""
function fexprDefinitionEvaluator(node, env::Dict=global_env, singleScope:
    name = string(node.args[1].args[1])
    params = node.args[1].args[2:end]
    block = node.args[2]
    lambda = Expr(:->, Expr(:tuple, params...), block, "fexpr")
    addEnvBinding(env, name, lambda)
    DEBUG_ENV && debug_env(env)
    return Fexpr(name)
end
```

Definition is the same as in regular functions but we add a flag to distinguish fexprs

```
# needed to define fexpr args as quotes instead of calls
if func.args[3] == "fexpr"
    isFexpr = true
    len = size(node.args, 1)
    for i in 2:len
        if node.args[i] isa Expr && node.args[i].head != :quote
            node.args[i] = Expr(:quote, node.args[i])
        end
    end
```

When the fexpr is called, we change its arguments to quotes so they won't be evaluated

Reflection

```
"""Main evaluator"""
function evaluate(node, env::Dict=global_env, singleScope::Dict=Dict{String,Any}("#" => nothing))
```

```
function functionCallEvaluator
    call = node.args[1]
    global isMacro
    tempEnv = copy(env)
```

This is why, whenever we call a function, we save its environment in a variable called tempEnv, in case we might need that level of scope for an eval later

Because of eval() inside fexprs, we had to add the singleScope argument to the evaluate function

When called, eval's arguments are supposed to be evaluated in the scope of the call to the fexpr, not the scope where the function was defined nor the scope of the body of the function.

Reflection

```
>> 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
```

In this example what happens is:

- we define the function puzzle(x) as a fexpr
- the values 3 and 4 are assigned to vars “a” and “b”
- when the function is called, the current environment is saved
- inside the function’s body, we get the value for “a” from the scope of the global function and “b” from the local scope
- however, when the eval is called, the environment used inside is the one we saved earlier

Metaprogramming

```
repeat_until(condition, action) $=
  let loop = gensym()
    :(let ;
      $loop() = ($action; $condition ?
false : $loop())
      $loop()
    end)
end
```

1. It will store the macro in the current environment

```
"""This function evaluates the definition of macros"""
function macroDefinitionEvaluator(node, env::Dict=global_env,
singleScope::Dict=Dict{String,Any}("#" ⇒ nothing))
    name = string(node.args[1].args[1])
    params = node.args[1].args[2:end]
    body = node.args[2]

    lambda = Expr(:macro, Expr(:tuple, params...), body, "macro")

    addEnvBinding(env, name, lambda)
    DEBUG_ENV && debug_env(temp)

    return Macro(name)
end
```

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

1. When the evaluator sees the **`repeat_until`** func it will resolve its value from the environment and know it's a macro
2. Then it **will not** evaluate it's parameters and it will use the same env as the current block (so its the same as the macro code being pasted inside the block)
3. It evaluates the contents of the macro with the parameters

```
if occursin(r"macro\"\)\)$", string(getEnvBinding(env, string(node.
args[1]))))
    # macro params will only be evaluated when they are called
    args = map(x -> x, node.args[2:end])
    isMacro = true
else
    args = map(x -> evaluate(x, env, singleScope), node.args
    [2:end])
end
```

```
if (isMacro)
    # macros need to be executed in the same environment
    for (param, arg) in zip(params, args)
        addEnvBindingSym(env, string(param), arg)
    end
    val = evaluate(func.args[2], env, singleScope)
    isMacro = false
    return val
end
```

We are now in macro mode (the global variable is set to true and so some behaviour of the evaluator is going to be different)

```
repeat_until(condition, action) $=
  let loop = gensym()
    :(let ;
      $loop() = ($action; $condition ?
false : $loop())
      $loop()
    end)
end
```

```
if size(node.args, 1) ⩾ 2 && node.args[2] isa Expr && size
(node.args[2].args, 1) ⩾ 1 && node.args[2].args[1] == :gensym
    # special case where we need to change the env
    createGenSym(env, name, value)

    DEBUG_ENV && debug_env(env)
    return value
end
addEnvBindingSym(env, name, value)
DEBUG_ENV && debug_env(env)
return value
```

1. The gensym function will create a new name for all the loop variables, an association of the two names will be made using the createGenSym function and the environment will now store an entry like this one:
   “/loop” = “some gen value”
   “some gen value” = “ ”
2. There is now the function addEnvBindingSym and getEnvBindingSym that besides the normal behaviour will first check for the existence of a variable association and retrieve the the correct value

```
"""Retrieves the node bound to a name in the environment. If there is a
symbolic binding for the name, it uses the symbolic binding instead"""
function getEnvBindingSym(env::Dict, name::String)
    if hasEnvBinding(env, "/" * name)
        addr = getEnvBinding(env, "/" * name)
        return env[addr]
    else
        return getEnvBinding(env, name)
    end
end
```

```
repeat_until(condition, action) $=
  let loop = gensym()
    :(let ;
      $loop() = ($action; $condition ?
false : $loop())
      $loop()
    end)
end
```

1. Now the macro executes like a function
2. Variables are stored and retrieved from the env without the $
3. Function calls like $loop still execute in the same environment (which is the env of the block where the macro was called)

```
elseif call.head == :$
    func = getEnvBindingSym(env, string(call.args[1]))
    val = evaluate(func.args[2], env, singleScope)
    return val
```

```
elseif node.head == :$ # Macro variable resolver
    val = getEnvBindingSym(env, string(node.args[1]))
    return evaluate(val, env, singleScope)
```

```
# variables inside macros are defined with "$"
# at the begining which is not part of the name
if occursin(r"^\$", name)
    name = name[2:end]
end
```

Testing

```
function metajulia_eval(input)

    if input isa String
        return string(input)
    end

    node = Meta.parse(string(input))
    Base.remove_linenums!(node)

    if node isa Expr || node isa Number || node isa String || node isa Symbol || node isa QuoteNode
        # Evaluate the node and print the result
        result = evaluate(node)

        #
        # Pretty-Printing for unit testing
        #
        if result isa String…
        end

        if isnothing(result)…
        end

        if result isa QuoteNode…
        end

        if result isa Expr && result.head == :quote…
        end

        return result

    else…
    end
end
```

A Meta-Circular Julia Evaluator

Group 14:

99172 - Adriana Nunes
99219 - Francisco Carvalho
99259 - José João Ferreira
99336 - Tiago Quinteiro