Meta-Circular Evaluator for Julia

Pedro Lobo Rómulo Kaidussis Tomás Nascimento

Advanced Programming - 2023/2024

Overview

Environments

Implementation

- Stack of set of mappings.

Environments

Implementation

- Stack of set of mappings.
- Each mapping associates a name with a value.

Environments

Implementation

- Stack of set of mappings.
- Each mapping associates a name with a value.
- Names are represented with symbols.

Environments

Implementation

- Stack of set of mappings.
- Each mapping associates a name with a value.
- Names are represented with symbols.

Code

```
struct Frame
    bindings::Dict{Symbol, Any}
end

struct Env
    stack::Vector{Frame}
end
```

Environments

Initial Bindings

- Logical Operators

Environments

Initial Bindings

- Logical Operators
- Arithmetic Operators

Environments

Initial Bindings

- Logical Operators
- Arithmetic Operators
- Other functions

Environments

Initial Bindings

- Logical Operators
- Arithmetic Operators
- Other functions

Code

```
initial_bindings::Dict{Symbol, Any} = Dict(
    :+ => +, :- => -, :* => *, :/ => /, ...
    :! => !, :~ => ~, :& => &, :| => |, ...
    :(==) => ==, :(!=) => !=, :(<) => <, :(<=) => <=, ...
    :(println) => println, :(print) => print,
    :(gensym) => gensym
)
```

Let Expressions

Scheme

```
>> (let ((a 1) (b (+ a 2)))
       (+ a b))
;Unbound variable: a
```

Let Expressions

Scheme

```
>> (let ((a 1) (b (+ a 2)))
       (+ a b))
;Unbound variable: a
```

Julia

```
>> let a = 1, b = a + 2
      a + b
   end
4
```

Let Expressions

Let Expressions

- Implemented in a similar way to the Scheme evaluator.

Let Expressions

Let Expressions

- Implemented in a similar way to the Scheme evaluator.

Code

```
let new_env = extend_env(env, [], [])
    for (name, init) in zip(names(expr), inits(expr, new_env))
        add_binding!(new_env, name, eval(init, new_env))
    end
    eval(body(expr), new_env)
end
```

Let Expressions

Let Expressions

- Implemented in a similar way to the Scheme evaluator.
- Environment is extended with the new mappings, one at a time.

Code

```
let new_env = extend_env(env, [], [])
    for (name, init) in zip(names(expr), inits(expr, new_env))
        add_binding!(new_env, name, eval(init, new_env))
    end
    eval(body(expr), new_env)
end
```

Let Expressions

Let Expressions

- Implemented in a similar way to the Scheme evaluator.
- Environment is extended with the new mappings, one at a time.
- Allows the evaluation of future initialization forms to reference back to already evaluated forms.

Code

```
let new_env = extend_env(env, [], [])
    for (name, init) in zip(names(expr), inits(expr, new_env))
        add_binding!(new_env, name, eval(init, new_env))
    end
    eval(body(expr), new_env)
end
```

Functions

Functions

- Julia supports lexical scope.

Functions

Functions

- Julia supports lexical scope.
- Functions need to capture the environment.

Functions

Functions

- Julia supports lexical scope.
- Functions need to capture the environment.

Code

```
struct Function
    lambda
    env
end
```

Recursive Functions

Recursive Functions

- Create a new empty frame.

Recursive Functions

Recursive Functions

- Create a new empty frame.
- Initialize the let inits in the new frame.

Recursive Functions

Recursive Functions

- Create a new empty frame.
- Initialize the let inits in the new frame.
- Bind the names to the inits in the new frame.

Recursive Functions

Recursive Functions

- Create a new empty frame.
- Initialize the let inits in the new frame.
- Bind the names to the inits in the new frame.

Code

```
eval_let(expr, env) =
    let extended_env = extend_env(env, [], [])
        for (name, init) in zip(let_names(expr), let_inits(expr, extended_env))
            add_binding!(extended_env, name, eval(init, extended_env))
        end
        eval(let_body(expr), extended_env)
    end
```

Quasiquote

Quasiquote

- Powerful meta-programming mechanism.

Quasiquote

Code

```
eval_quasiquote(expr, env) = expand(quasiquoted_form(expr), env)

expand(form, env) =
    if is_unquote(form)
        eval(unquote_form(form), env)
    elseif is_quasiquote(form)
        expand(quasiquoted_form(form), env)
    elseif isa(form, Expr)
        form.args = map(i -> expand(i, env), form.args)
        form
    else
        form
    end
```

Fexprs

Fexprs

- Does not evaluate its arguments.

Fexprs

Fexprs

- Does not evaluate its arguments.

Code

```
struct Fexpr
    lambda
    env
end
```

Fexprs

Fexprs

- Does not evaluate its arguments.
- Allows for the use of `eval`.

Fexprs

Fexprs

- Does not evaluate its arguments.
- Allows for the use of `eval`.

Code

```
let (lambda, lambda_env) = (function_lambda(f), function_env(f))
    extend_env!(lambda_env, lambda_params(lambda), call_args(call))
    add_binding!(lambda_env, :eval, x -> eval(x, env))
    eval(lambda_body(lambda), lambda_env)
end
```

Macros

Code

```
struct Macro
    lambda
    env
end
```

Macros

Macro Calls

- Extend environment with unevaluated arguments.

Macros

Macro Calls

- Extend environment with unevaluated arguments.
- Macro expansion is computed.

Macros

Macro Calls

- Extend environment with unevaluated arguments.
- Macro expansion is computed.
- Macro expansion is evaluated.

Macros

Macro Calls

- Extend environment with unevaluated arguments.
- Macro expansion is computed.
- Macro expansion is evaluated.

Code

```
let (lambda, lambda_env) = (macro_lambda(f), macro_env(f))
    extend_env!(lambda_env, lambda_params(lambda), call_args(call))
    eval(eval(lambda_body(lambda), lambda_env), env)
end
```

Test Suite

Test Suite

- 102 tests.

Test Suite

Test Suite

- 102 tests.
- Located in the `tests.jl` file.