Beyond Exception Handling

Implementing restarts in Julia

Alexandre Faria
97005

Rafael Pinto
97153

Instituto Superior Técnico

May 2020

Data Structures and Global Variables

- ReturnFromException is used to transfer control between return_from to block
- Handlers are stored in a global stack
- A counter is used to keep block names unique
- Group all the handlers from a handler_bind form and store those groups as a list in current_available_handlers

```
# exception used in block and return_from
struct ReturnFromException <: Exception
    name    # block name
    value   # return value
end

# global variables
current_available_restarts = []     # stack of restarts
current_available_handlers = []     # stack of handler groups
current_block_id = 0                # block name counter
```

Handler Bind

1. Push handlers onto the start of the current_available_handlers stack
2. Call the function
3. Pop the handlers from the stack

```
function handler_bind(func, handlers...)
    global current_available_handlers

    pushfirst!(current_available_handlers, handlers)

    try
        func()
    finally
        popfirst!(current_available_handlers)
    end
end
```

Error Handling

- The handler corresponding to the first matched exception is executed
- Otherwise, the error is not caught and is thrown normally

```
function error(exception::Exception)
    for handler_group in current_available_handlers
        for handler in handler_group
            if exception isa handler[1]
                handler[2](exception)
                break
            end
        end
    end
    throw(exception)
end
```

Block

- Use the name in the exception to know which block is supposed to handle it
- If this block doesnt match the name, propagate the exception upwards in the chain call to another block

```
function block(func)
    global current_block_id
    current_block_id += 1
    name = current_block_id
    try
        func(name)
    catch e
        if e isa ReturnFromException && e.name == name
            return e.value
        end
        rethrow(e)
    end
end
```

Return From

- Throw a ReturnFromException exception with the block name and the value to return
- The block form will use this information to gain the control flow and return

```
function return_from(name, value = nothing)
    throw(ReturnFromException(name, value))
end
```

Restart Binding

- Before calling the function, we add each restart to the stack so that it's available later to be invoked
- The context of execution is inserted into a block form and the name 'rb_block' is passed into the restart function by wrapping it in an anonymous functions that capturesthe value of rb_block

```
function restart_bind(func, restarts...)
    block() do rb_block
        for r in restarts
            pushfirst!(current_available_restarts,
                (r[1] => (args...) -> return_from(rb_block, r[2](args...))))
        end
        ...
    end
end
```

Restart Binding

- After establishing the restarts, the execution can then be passed to the given function
- Once the block gets the control of the execution back, the restarts established for that block must be removed
- The finally block guarantees that the restarts are always removed from the stack, even if an exceptional situation occurs

```
function restart_bind(func, restarts...)
    block() do rb_block
        ...
        try
            func()
        finally
            for i in restarts
                popfirst!(current_available_restarts)
            end
        ...
```

Available Restarts

- The ability to know whether a restart is available is useful to conditionally invoke a restart
- The function looks for any restart, with the given name, in the current available restarts
- The current available restarts contains the restarts that were established along the call chain

```
function available_restart(name)
    global current_available_restarts
    any(r -> r[1] == name, current_available_restarts)
end
```

Invoking Restarts

- In order to invoke a restart, the program evaluates if there are currently available restarts that match the name provided by the handler
- If so, it will invoke the restart with the given arguments which will resume the execution in some pre-established point

```
function invoke_restart(name, args...)
    global current_available_restarts
    i = findfirst(r -> r[1] == name, current_available_restarts)
    func = current_available_restarts[i][2]
    func(args...)
end
```

Signal Extension

- The signal function is simply the error function except it doesn't throw the exception if there are no available handlers

```
function signal(exception::Exception)
    global current_available_handlers

    for handler_group in current_available_handlers
        for handler in handler_group
            if exception isa handler[1]
                handler[2](exception)
                break
            end
        end
    end
end
```

Handler Case Macro

- This macro wraps the handler in a block and return_from form

```
macro handler_case(func, handlers...)
    let
        escape_block = Symbol("escape_block")
        handler_func = func

        for handler in handlers
            handler_body = handler.args[3].args[2]
            handler.args[3].args[2] = :(return_from($(escape_block), $handler_body))
        end

# hack in order to work with two macro calling syntax
# @handler_case(reciprocal(0), DivisionByZero => (c) -> println("zero!"))
# @handler_case(DivisionByZero => (c) -> print("zero!")) do reciprocal(0) end
        if func.head == :call
            handler_func = :(() -> begin $func end)
        end
        quote
            block() do $(escape_block)
                handler_bind($handler_func, $(handlers...))
            end
        end
    end
end
```

Restart Case Macro

- This macro simply calls restart_bind since it's already in the same form as restart_case

```
macro restart_case(func, restarts...)
    let
        quote
            restart_bind($func, $(restarts...))
        end
    end
end
```

Interactively Invoking a Restart

- Example of the interactive restarts extensions

```
struct DivisionByZero <: Exception end

reciprocal(value) =
    restart_bind(
        Restart(:return_zero => (args...) -> 0, report="Return Zero"),
        Restart(:return_value => identity, report="Return Value",
                                           interactive=true,
                                           test = () -> false),
        Restart(:retry_using => reciprocal, report="Retry with another value",
                                            interactive=true)
    ) do
        value == 0 ? error(DivisionByZero()) : 1 / value
    end
```

Interactively Invoking a Restart

- Only the functions which 'test' returns true are shown
- The user handling of restarts is only run when no handlers execute a non-local transfer control

```
julia> reciprocal(0)

#<DivisionByZero()>#
 [Condition of type DivisionByZero]
Restarts:
 1: [RETRY_USING] Retry with another parameter
 2: [RETURN_ZERO] Return Zero
Pick: 2
0
```

Interactively Invoking a Restart

- Only the functions which 'test' returns true are shown
- There's also the possibility to ask the user for input by setting 'interactive' to true
- The 'interactive' variable was simplified to be easier to implement interactive restarting

```
julia> reciprocal(0)

#<DivisionByZero()>#
 [Condition of type DivisionByZero]
Restarts:
 1: [RETRY_USING] Retry with another parameter
 2: [RETURN_ZERO] Return Zero
Pick: 1
Input: 10
0.1
```

Interactively Invoking a Restart

- Limited backwards compatibility with normal restarts
- The interactive handling also works with normal restarts but no input is asked from the user, therefore it only works with restarts which take no parameters

```
julia> reciprocal(0)

#<DivisionByZero()>#
 [Condition of type DivisionByZero]
Restarts:
 1: [RETRY_USING] retry_using
 2: [RETURN_VALUE] return_value
 3: [RETURN_ZERO] return_zero
Pick: 3
0
```

Restart Struct

- This structure holds the values for an Extended Restart
- This is used to be able to print the restart with a name, test if its available, and get input from the user

```
mutable struct Restart
    restart     # restart pair :return_zero => () -> 0
    test        # test if the restart is available
    interactive # simplified to be simply a boolean
                # instead of a custom function that asks the user for parameters
    report      # name of the restart that appears in the prompt

    function Restart(restart; test= ()->true, interactive=false, report=nothing)
        report = report == nothing ? string(name) : report
        new(restart, test, interactive, report)
    end
end
```

Interactively Invoking a Restart

- This method is added to the global stack of handlers so that its the last to be reached
- This method is only called if no handlers execute a non-local transfer of control
- It searches for restarts in the global stack and prints them

```
current_available_handlers = push!(current_available_handlers,
                [Exception => (c) -> picking_interactive_restart_handler(c)])
function picking_interactive_restart_handler(exception)
    global current_available_restarts
    restarts = filter(r -> (!(r[2].r isa Restart) ||
                            (r[2].r isa Restart && r[2].r.test())),
                      current_available_restarts)
    if length(restarts) > 0
        println()
        println("#<$(exception)>#")
        println(" [Condition of type $(typeof(exception))]")
        println("Restarts:")
        ...
    end
end
```

Interactively Invoking a Restart

- The 'if' inside the for loop makes it 'backwards' compatible with non-extended restarts (:name => func)

```
current_available_handlers = push!(current_available_handlers, [Exception => (c) ->
picking_interactive_restart_handler(c)])
function picking_interactive_restart_handler(exception)
    ...
    if length(restarts) > 0
        ...
        for r in restarts
            if r[2].r isa Restart
                println(" $(i): [$(uppercase(String(r[1])))] $(r[2].r.report)")
            else
                println(" $(i): [$(uppercase(String(r[1])))] $(r[1])")
            end
            i += 1
        end
        print("Pick: ")
        restart = readline()
        i = tryparse(Int, restart)
        restart = restarts[i]
        ...
end
```

Interactively Invoking a Restart

- It then selects the restart and uses invoke_restart
- Passing parameters to non-extended restarts was not implemented, nor was any type of type checking for parameters which would increase the complexity of this extension

```
function picking_interactive_restart_handler(exception)
    ...
    if length(restarts) > 0
        ...
        value = nothing
        if restart[2].r isa Restart && restart[2].r.interactive
            print("Input: ")
            value = Meta.parse(readline())
            invoke_restart(restart[1], value)
        else
            invoke_restart(restart[1])
        end
    end
end
```

Interactively Invoking a Restart

- In order to support this, restart_bind was also extended by adding another generic function which receives the new extended restarts
- This was necessary in order to convert them into the normal, un-extended, pair format (:name => func)

```
function restart_bind(func, restarts::Restart...)
    global current_available_restarts
    block() do rb_block
        for r in restarts
            # wrap inside an anonymous function that
            # captures the value of rb_block
            pushfirst!(current_available_restarts,
               (r.restart[1] => (args...) -> return_from(rb_block,
                                                r.restart[2](args...))))
        end
        try
            func()
        finally
            for i in restarts
                popfirst!(current_available_restarts)
            end
end end end
```