Beyond Exception Handling

Pedro Mendes, Ricardo Pereira

Instituto Superior Técnico

May 22, 2020

Overview

1. `block & return_from`
2. `handler_bind`
   - `error` function
3. `restart_bind`
4. `invoke_restart`
   - `error` function
5. Extensions
   - `signal`
   - Interactive restarts
   - `@handler_case`
   - `@fn_types`
   - `@restart_case`
6. Extra remarks

Functions `block` & `return_from`

These two functions were implemented using the try-catch mechanism present in Julia.

`block`

To distinguish different blocks we decided to attribute a unique id to each block, to achieve this the `block` function uses a global counter to generate the token that will be assigned to the block, and then used by the `return_from` function. If the block function catches an exception that is not of type `ReturnFrom`, it simply rethrows it so it doesn't intervene in the rest of the program's control flow.

`return_from`

The `return_from` function then throws a known exception called `ReturnFrom` containing the token and the return value to return in the block.

handler_bind

Handler Stack

To make the handler bind work, we used a global handler stack. This stack has a Vector of handlers, each of them belonging to a separate `handler_bind` call.
Using a stack of vectors instead of a flat vector of handlers allows us to nest `handler_bind` calls and still distinguish to which call the handler belongs to.

handler_bind

This function then simply pushes its passed handlers to the stack and calls the wrapped function normally.

The `error` function when combined with `handler_bind`

Important Note

The error function is the function that makes the whole architecture tick. As such we will be returning to it latter in the presentation.

To make the `handler_bind` work, the error function tries to find a suitable handler in the Handler Stack, for each set of handler it tries to call all that match the type of the passed error.
Throwing in a handler will result in no more handlers being called as this counts as a *non-local transfer of control*. As such a simple implementation of it would look like this.

error example

```julia
function error(err)
    while (hs = peek(HANDLER_STACK)) != nothing
        for h in filter(h -> err isa h.first, hs)
            h.second(err)
        end
        pop!(HANDLER_STACK)
    end
    throw(err)
end
```

restart_bind

restart_bind

`restart_bind` is very similar to `handler_bind`, all it does is collect the restarts it receives and stores them in a stack (a separate one from the one used by `handler_bind`), so that they can be referenced later. After doing so it simply calls the wrapped function.

Restart Stack

We use a stack here for the same reason we used on for the handlers, this way we can pop the restarts in logical blocks so, this also allows nesting of restarts with the same name.

invoke_restart

invoke_restart

Invoke restart is designed to works a possible handler of handler bind. If it is called outside this context it's behaviour is undefined.
Given this constraint we know that handlers are only called by the error function, which means that all calls made to this function are done indirectly by `error`.

Finding a restart

This function will then pop restarts from the restart stack until it finds one that matches the given name, when it does, it calls the restart and throws another known exception called `RestartResult` that contains the return value of the called restart. This exception is then caught by `error` which will halt the handler lookup to return the value wrapped in the `RestartResult`

The new `error` function

Changes to the `error` function

The `error` function had to be changed to accommodate for this feature, a try catch was added to check if the handler threw a `RestartResult` and handle it accordingly as explained before.

error example

```
function error(err)
  while (hs = peek(HANDLER_STACK)) != nothing
    for h in filter(h -> err isa h.first, hs)
      try
        h.second(err)
      catch e # added
        # if the handler `h.second` invoked a restart
        if e isa RestartResult
          return e.result # return it
        else
          # otherwise maintain the behavior that throwing is a
          # non-local transfer of control
          rethrow(e)
    end; end; end
    pop!(HANDLER_STACK)
  end
  throw(err)
end
```

`signal`

The signal function simply makes use of the `Base.error` function to abort the program no matter what is happening, throwing an `ErrorException`.

Interactive restarts

invoke_restart_interactive

This function can be used anywhere a normal `invoke_restart` function can be used. It then will, when called, interrupt the program to provide an interactive prompt for the user to pick a restart or cancel to return without invoking any restarts.

Example

Given this function,

```
reciprocal_restart(v) =
    restart_bind(:return_zero => () -> 0,
                 :return_value => identity,
                 :retry_using => (f) -> f()) do
    reciprocal(0)
end
```

if an interactive restart is used a prompt like this will show up:

Interactive restarts

Example

```
Choose a restart:
0: cancel :: !
1: return_zero :: Any
2: retry_using :: Any
3: return_value :: Any
restart>
```

Example

If there were other restarts in previous functions they can also be called here and are distinguished by their level on indentation.

```
Choose a restart:
0: cancel :: !
1: return_zero :: Any
  2: just_do_it :: Any
restart>
```

Parameterized restarts

Some restarts require parameters to be passed. Because julia doesn't let you reflect on the types of lambdas at runtime the programmer needs to supply the parameters by hand. For the example above this will look like so

Example

```
reciprocal_restart(v) =
  restart_bind(:r_zero => () -> 0,
               :r_value => identity => (Float64,),
               :r_using => ((f) -> f()) => (Function,)) do
  reciprocal(0)
end
```

Parameterized restarts

Example

```
0: cancel :: !
1: return_zero :: Any
2: retry_using :: Function -> Any
3: return_value :: Float64 -> Any
restart> 3
Input Float64: 4.2
```

Example

```
Choose a restart:
0: cancel :: !
1: return_zero :: Any
2: retry_using :: Function -> Any
3: return_value :: Float64 -> Any
restart> 2
Input Function: () -> 8.4 / 2
```

@handler_case

Abbreviating common idioms

This macro is a shortcut for combining `block`, `return_from` and `handler_bind`.

Example

Instead of writing

```
block() do token
  handler_bind(Except => (c) -> return_from(token, some_val)) do
    some_function()
  end
end
```

One can simply write

```
@handler_case some_function() begin
    c::Except = some_val
end
```

Nesting

Example

It can also be nested in arbitrary ways without causing conflicts.

```
@handler_case e() begin
    c::Except1 = 1
    c::Except2 = @handler_case e() begin
            d::Except1 = c.n
            d::Except2 = d.n * 2
        end
end
```

And even a default exception can be used much like the language's try catch expression can.

```
@handler_case e() begin
    c = 1
end
```

@fn_types

To make this process more pleasant a new `@fn_types` macro was implemented. It can be applied to lambdas whose type arguments are explicitly typed and it will generate the type tuple for the user.

Example

This code is equivalent to the example in Slide 14.

```
reciprocal_macro(v) = restart_bind(
        :return_zero => () -> 0,
        :return_value => (@fn_types (c::Float64) -> c),
        :retry_using => @fn_types (f::Function) -> f()) do
    reciprocal(0)
end
```

This is a convenience macro for defining restart cases. Instead of writing

```
some_restartable(v) =
restart_bind(:return_zero => () -> 0,
             :return_value => ((c) -> c) => (Float64,),
             :retry_using => ((f) -> f()) => (Function,)) do
    some_function(v)
end
```

One can simply write

```
some_restartable(v) = @restart_case some_function(b) begin
    :return_zero => () -> 0
    :return_value => (c::Float64) -> c
    :retry_using => (f::Function) -> f()
end
```

This macro will also make sure to include the types of the lambdas when they are typed so that `invoke_restart_interactive` can then ask for the parameters for parsing.

Pattern matching

@match macro

During the development of the project lots of if/else-if/else chains where used theses were very verbose but Julia does not have a builtin switch/case or pattern match mechanism.
To compensate for this a simple @match macro was written to alleviate this pain point.

Example

```
@match parse(Int, some_string) begin
    0   => println("Got a 0")
    1   => println("Got a 1")
    2:6 => println("Got a number between 2 and 6")
    _   => println("Got something else: \$_")
end
```

Pattern matching

This macro supports matching:

- on constants;
- on ranges of values
- default cases (using the _ character)

It was heavily inspired by the macro implemented by kmsquire and *hosted on github*.