Beyond Exception Handling

António Menezes Leitão

April, 2020

1 Introduction

An *exceptional situation* occurs when a program reaches a point in its execution where a planned operation cannot be done. For example, the program might be trying to read from a file but the file does not exist, or it might be trying to divide two numbers but the divisor is zero. Most programming languages consider these exceptional situations as *runtime errors* and they provide ways for the program to continue to execute from a safe point previously established by the programmer.

In most programming languages, dealing with exceptional situations comprises two different aspects:

Signaling When the program discovers that one operation cannot be done, it *signals* that it found an exceptional situation. Some languages, such as Java, do this by *throwing* an *exception*. The exception is a data structure that encodes not only the exceptional situation found but also relevant information needed to understand the situation. For example, in the case of a missing file, the exception might include the file's pathname.

Handling In order to handle the exceptional situation, it is necessary to transfer the execution to a different part of the program that was prepared in advance to deal with the situation. In the Java language, for example, this is implemented using a `try-catch` statement.

Signaling, in most programming languages, has the inconvenient side-effect of canceling the computation that was pending between the point where the exceptional situation is detected and the point where it is handled, but it does not need to be that way. The Common Lisp language, in 1988, introduced the idea of *restarting* where, besides signaling the exceptional situation, the program also presents different alternatives to fix the situation and proceed with the computation, called *restarts*. In the handling part of the process, either the computation is canceled, or one of the available restarts is selected and the computation proceeds with that restart without canceling the pending computation.

Many of the ideas introduced by the Common Lisp language were later implemented in other programming languages but *restarting* is not one of them. Until now.

2 Restarts in Julia

Julia is a recent language that is still being developed at MIT. Julia supports some of the most important features of Common Lisp, namely, multiple-dispatch methods and metaprogramming. Unfortunately, Julia does not provide anything even remotely similar to Common Lisp' restarts. In fact, exception handling in Julia is much more similar to the `try-catch` approach used in Java. Fortunately, the language is expressive enough to allow us to imagine a possible implementation of restarts. As an example, we will consider a function that computes the reciprocal of a number x: $1/x$. We will start with a simple definition:

```
reciprocal(x) =
  x == 0 ?
    error(DivisionByZero()) :
    1/x
```

Note that the function first checks if it is attempting to divide by zero. In that is not the case, it returns the reciprocal of the argument, otherwise, it signals the `DivisionByZero` exception. This exception can be defined as any other Julia exception, by inheriting from the `Exception` type:

```
struct DivisionByZero <: Exception end
```

When we use the function, we get the expected behavior:

```
julia> reciprocal(10)
0.1
julia> reciprocal(0)
ERROR: DivisionByZero() was not handled.
```

In order to be notified about an exceptional situation, we can use the `handler_bind` function. This function takes as arguments a function and an arbitrary sequence of pairs of the form *Exception Type* `=>` *handler function* which are treated as potential exception handlers. The `handler_bind` function calls the first argument in the context of these handlers. If during this call an exception is signaled, the first *Exception Type* that matches the signaled exception determines the handler to execute. Note that an exception type matches an exception if the exception is a direct or indirect instance of the type. When an exception handler is found, the corresponding function is called with the exception as argument.

Here is an example of the use of `handler_bind`:

```
handler_bind(()->reciprocal(0),
             DivisionByZero =>
               (c)->println("I saw a division by zero"))
```

Given that Julia treats the form

```
foo(args...) do x,y,...
  expr
end
```

as equivalent to

```
foo((x,y,...)->expr, args...)
```

from now on we will use the more palatable `do` syntax. Here is the equivalent code tested at the Julia REPL:

```
julia> handler_bind(DivisionByZero =>
                      (c)->println("I saw a division by zero")) do
         reciprocal(0)
       end
I saw a division by zero
ERROR: DivisionByZero() was not handled.
```

Note, in the previous example, that although the exception handler was called, it didn't stop the propagation of the signal. We can see this behavior in more detail by cascading multiple `handler_bind` forms:

```
julia> handler_bind(DivisionByZero =>
                      (c)->println("I saw it too")) do
         handler_bind(DivisionByZero =>
                        (c)->println("I saw a division by zero")) do
           reciprocal(0)
         end
       end
I saw a division by zero
I saw it too
ERROR: DivisionByZero() was not handled.
```

As can be seen, just calling the exception handler is not sufficient to stop the propagation of the signal. That behavior is intended because there are handlers that just want to know about a signal without actually taking any action to solve the problem. If a handler wants to stop the signal propagation, it must also make a *non-local transfer of control*, either by *escaping* from the `handler_bind` form or by going back to the signaling form. We will now discuss the former case.

A *non-local* transfer of control happens when execution jumps out from a given context and resumes at a different context. For example, a *non-local* exit happens when a function call returns to a point of the program that is not the one where the call was made. This is what happens, for example, when a Java program *throws* an exception and control is transferred to a *catch* clause that might be far away from the point where the exception was thrown. In the case of the C programming language, similar effects can be achieved using the `setjmp`/`longjmp` operations.

To use non-local transfers of control in the Julia language, we will assume the existence of two functions, called `block` and `return_from`, that have the same semantics as the identically-named forms of Common Lisp, i.e., `block` establishes a named exit point to which the execution can be transferred by calling the `return_from` function with the name of the exit point and the value that should be returned. Here is an example of the use of these functions:

```
mystery(n) =
  1 +
  block() do outer
    1 +
    block() do inner
      1 +
      if n == 0
        return_from(inner, 1)
      elseif n == 1
        return_from(outer, 1)
      else
        1
      end
    end
  end

julia> mystery(0)
3
julia> mystery(1)
2
julia> mystery(2)
4
```

With the `block` function, we can effectively handle an exceptional situation by *escaping* the `handler_bind` form, as follows:

```
julia> block() do escape
         handler_bind(DivisionByZero =>
                        (c)->(println("I saw it too");
                              return_from(escape, "Done"))) do
           handler_bind(DivisionByZero =>
                          (c)->println("I saw a division by zero")) do
             reciprocal(0)
           end
         end
       end
I saw a division by zero
I saw it too
"Done"
```

Naturally, it is possible to escape from multiple `handler_bind` forms:

```
julia> block() do escape
         handler_bind(DivisionByZero =>
                        (c)->println("I saw it too")) do
           handler_bind(DivisionByZero =>
                          (c)->(println("I saw a division by zero");
                                return_from(escape, "Done"))) do
             reciprocal(0)
           end
         end
       end
I saw a division by zero
"Done"
```

As is demonstrated by the previous examples, the rule for handling exceptional situations says that when an exceptional situation is signaled, if a handler is found, it may either handle the situation, by performing a non-local transfer of control, or decline to handle it, by failing to perform a non-local transfer of control. If it declines, other handlers are sought.

Handling an exceptional situation by escaping the `handler_bind` form is not exceptional (no pun intended). In fact, it is similar to what a traditional `try-catch` form can do. What is exceptional is the ability to go back to the place where the exceptional situation was signaled, a trick that very few programming languages can natively pull off. To that end, we need to use the `restart_bind` form. Here is one example:

```
reciprocal(value) =
  restart_bind(:return_zero => ()->0,
               :return_value => identity,
               :retry_using => reciprocal) do
    value == 0 ?
      error(DivisionByZero()) :
      1/value
  end
```

The `restart_bind` function takes any number of pairs of the form *keyword* => *restart function*, which are treated as named restarts, and then proceeds to evaluate its body. If, during the evaluation of the body, an exceptional situation is signaled, it is possible for an exception handler to invoke a restart. Here is an example:

```
julia> handler_bind(DivisionByZero =>
                      (c)->invoke_restart(:return_zero)) do
         reciprocal(0)
       end
0
```

It is also possible to pass arguments to a restart, as long as the restart function has the corresponding number of parameters. Here are two examples:

```
julia> handler_bind(DivisionByZero =>
                      (c)->invoke_restart(:return_value, 123)) do
         reciprocal(0)
       end
123
julia> handler_bind(DivisionByZero =>
                      (c)->invoke_restart(:retry_using, 10)) do
         reciprocal(0)
       end
0.1
```

Another powerful feature is the ability to know whether a restart is available. This is achieved by the predicate `available_restart`, that takes the restart name and returns true when that restart is available and false otherwise. This function is useful to *conditionally* invoke a restart, as the following example demonstrates:

```
handler_bind(DivisionByZero =>
               (c)-> for restart in (:return_one, :return_zero, :die_horribly)
                       if available_restart(restart)
                         invoke_restart(restart)
                       end
                     end) do
    reciprocal(0)
end
```

Finally, it is important to mention that the available restarts at any given moment include not only those that are typically located near the point where the exceptional situation is detected but also all others that were established along the call chain, as illustrated by the following function definition:

```
infinity() =
  restart_bind(:just_do_it => ()->1/0) do
    reciprocal(0)
  end
```

and by the following interaction:

```
julia> handler_bind(DivisionByZero =>
                      (c)->invoke_restart(:return_zero)) do
         infinity()
       end
0
julia> handler_bind(DivisionByZero =>
                      (c)->invoke_restart(:return_value, 1)) do
         infinity()
       end
1
julia> handler_bind(DivisionByZero =>
                      (c)->invoke_restart(:retry_using, 10)) do
         infinity()
       end
0.1
julia> handler_bind(DivisionByZero =>
                      (c)->invoke_restart(:just_do_it)) do
         infinity()
       end
Inf
```

3 Goals

The main goal of this project is the implementation, in the Julia programming language, of the operations for the signaling and handling of exceptional situations, including the use of restarts, as illustrated in the previous section.

Your implemention should use Julia's *exceptions* to describe exceptional situations. However, signaling and handling will require specific function definitions that you need to implement in a way that makes the previous examples run as expected. More specifically, you need to define the functions with the following signatures:

- `block(func)`
- `return_from(name, value=nothing)`
- `available_restart(name)`
- `invoke_restart(name, args...)`
- `restart_bind(func, restarts...)`
- `error(exception::Exception)`
- `handler_bind(func, handlers...)`

You must implement the required functionality in a single file named `Exceptional.jl`.

3.1 Extensions

You can extend your project to further increase your grade above 20. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Examples of interesting extensions include:

- Implementing the `signal` function that signals an exceptional situation without requiring the corresponding exception handling. The `error` can be easily implemented on top of this function.
- Implementing user-handling of restarts, as is possible in Common Lisp by interacting with the user, presenting the available restarts and allowing him to choose the preferred one.
- Implementing the Julia equivalent to the Common Lisp restart options `:test`, `:report`, and `:interactive`.
- Implementing the *macros* `handler_case` and `restart_case` to simplify the use of the functions `handler_bind` and `restart_bind`.

Be careful when implementing extensions, so that the extra functionality does not compromise the functionality asked in the previous sections. To ensure this behavior, you should implement all your extensions in a different file named `ExceptionalExtended.jl`.

4 Code

Your implementation must work in Julia 1.4.

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.

5 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.

6 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named as `project.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing:

- The source code, within subdirectory `/src/`

The only accepted format for the presentation slides is PDF. This PDF file must be submitted using the Fénix Portal separately from the ZIP file and should be named `presentation.pdf`.

7 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

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

8 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 as long as the proper attribution is provided.

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

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

9 Final Notes

Don't forget Murphy's Law.

10 Deadlines

The code must be submitted via Fénix, no later than 19:00 of **May, 15**. Similarly, the presentation must be submitted via Fénix, no later than 19:00 of **May, 15**.