Typology of Programming Languages

Error handling

May 2025

Section 1

Errors and Exceptions

Ariane flight V88

4 June 1996

Maiden flight of the Ariane 5 rocket.

Ariane flight V88

4 June 1996

Maiden flight of the Ariane 5 rocket.

Reused code from Arianne 4 resulted in an un-handled integer overflow.

Often deemed “the most expensive bug in history” at $307 million.

Between the chair and the keyboard…

- *To Err is Human*: developpers **will** make mistakes.
- *Murphy's Law*: things **will** break.

Between the chair and the keyboard…

- *To Err is Human*: developpers **will** make mistakes.
- *Murphy's Law*: things **will** break.

Most programs will fail at some point. We must expect errors to happen and plan around them.

…between the keyboard and the computer

Exercise

How can languages and tools help us prevent errors?

…between the keyboard and the computer

Exercise

How can languages and tools help us prevent errors?

- **Static** analysis: catch as many things as possible *before* running the code
  - Compiler errors.
  - Static analysis tools (*clang-tidy*, *astrée*, *infer*…).
- **Dynamic** analysis: catch bugs *during* program execution
  - Often better to abort than keep going in a failure state.
  - Coupled with extensive testing, should catch everything the compiler doesn't detect. Hopefully.
- **Formal verification**: prove the correctness of the program

Sources of errors

Exercise

What can cause a program to fail?

Sources of errors

Exercise

What can cause a program to fail?

- "User" errors
  - Bad inputs, wrong order of operations...
- "Environment" errors
  - Data corruption, network errors, hardware errors...
- "Developper" errors
  - Forgot to check if `ptr == NULL` :(
  - Did not check the syscall return :(
  - Casting to the wrong type :(

Sources of errors

Exercise

What can cause a program to fail?

- "User" errors
  - Bad inputs, wrong order of operations...
- "Environment" errors
  - Data corruption, network errors, hardware errors...
- "Developper" errors
  - Forgot to check if `ptr == NULL` :(
  - Did not check the syscall return :(
  - Casting to the wrong type :(
  - These depend a lot on the language used!

Sources of errors

Exercise

What can cause a program to fail?

- "User" errors
  - Bad inputs, wrong order of operations...
- "Environment" errors
  - Data corruption, network errors, hardware errors...
- "Developper" errors
  - Forgot to check if `ptr == NULL` :(
  - Did not check the syscall return :(
  - Casting to the wrong type :(
  - These depend a lot on the language used!

Errors can be classified as two main kinds: **recoverable** and **unrecoverable**.

Error handling in C

Exercise

How to handle errors in C?

Error handling in C

Exercise

How to handle errors in C?

- Recoverable: error values.
- Unrecoverable: exit program.

```c
// C
void* p = malloc(size);

if (p == NULL)
   err(EXIT_FAILURE, NULL);

int fd = open(file_name, O_RDONLY, 0);

if (fd == -1)
   err(EXIT_FAILURE, "%s", file_name);
```

Error handling in C

Exercise

What are the limits of these approaches?

Error handling in C

Exercise

What are the limits of these approaches?

```c
// C
int fd = open(file_name, O_RDONLY, 0);

char s[32];
read(fd, s, 32);
```

Error handling in C

Exercise

What are the limits of these approaches?

```c
// C
int fd = open(file_name, O_RDONLY, 0);

char s[32];
read(fd, s, 32);

// Oops forgot to check if fd is -1
```

Error handling in C

Exercise

What are the limits of these approaches?

```c
// C
int fd = open(file_name, O_RDONLY, 0);

char s[32];
read(fd, s, 32);

// Oops forgot to check if fd is -1

// Oops also forgot to check what read returns
```

Error handling in Go

Go has a dedicated error type, which is idiomatically handled through a return of multiple values.

```
// Go
import "errors"

func first_positive(l []int) (int, error) {
    if len(l) == 0 {
        return 42, errors.New("Empty slice")
    }

    for _, v := range l {
        if v >= 0 {
            return v, nil
        }
    }
    return 0, errors.New("No positive")
}
```

Error handling in Go

```go
// Go
func main() {
  first, err := first_positive(...)
  if err != nil {
    // Use first
  } else {
    // Error case
  }
}
```

Error handling in Go

```go
// Go
func main() {
  first, err := first_positive(...)
  if err != nil {
    // Use first
  } else {
    // Error case
  }
}
```

Exercise

Go's error handling is often criticized. Why?

Error handling in Go

```
// Go
func main() {
  first, err := first_positive(...)
  if err != nil {
    // Use first
  } else {
    // Error case
  }
}
```

- Always returning a value, even in error cases.
  - So the error can be ignored by the developper…
  - Remember sycalls returning `-1`?

Exercise

Go's error handling is often criticized. Why?

```
// Go
func main() {
  first, _ := first_positive(...)
  // Oops forgot to check the error
}
```

Optionals

```cpp
// C++
std::optional<int>
get_first(const std::vector<int>& v) {
  if (v.empty())
    return std::nullopt;
  else
    return { v[0] };
}
```

- Dedicated “null” value for every context
- Strongly typed, need to handle the null case explicitely
    - No risk of nullptr deref!
    - Cannot ignore the null value!

Optionals

Definition

An **option type** is a *sum type* representing either a value or its absence.

```ocaml
(* OCaml *)
type 'a option =
  | Some of 'a
  | None
```

```rust
// Rust
enum Option<T> {
  Some(T),
  None,
}
```

```cpp
// Not the way it's implemented in C++ but could be
template <typename T>
using std::optional<T> = std::variant<std::nullopt_t, T>;
```

Nullable types

Definition

A **nullable type** is a type which values can be the *null value*.

```kotlin
// Kotlin
var a: String = "abc"
a = null // KO: a must be a String
val la = a.length
print(la)
```

```kotlin
// Kotlin
var b: String? = "abc"
b = null // OK: b is of a nullable type
val lb = b.length // KO: b may be null
print(l)
```

Limits of optionals

Exercise

What are the limits of option types / nullable types?

Limits of optionals

Exercise

What are the limits of option types / nullable types?

- Only one “error” value!
- Not fine grained enough for cases like `get_first_positive(int[])`
  - Two error cases!

Result types

Definition

A **result type** is a *sum type* representing either the result of a computation or the error that occured during it.

- Generalization of option types.

```ocaml
(* OCaml *)
type ('a, 'b) result =
  | Ok of 'a
  | Error of 'b
```

```rust
// Rust
enum Result<T, Error> {
    Ok(T),
    Err(Error),
}
```

Result type in OCaml

```ocaml
(* OCaml *)
let first_positive l =
  let rec aux l =
    match l with
    | [] -> Error `No_positive
    | hd :: tl ->
      if hd >= 0
        then Ok hd
        else aux tl
  in
  match l with
  | [] -> Error `Empty_list
  | l -> aux l

(* first_positive : int list -> (int, [> `Empty_list | `No_positive ]) result *)
```

Result type in Rust

```rust
// Rust
enum MyError {
    EmptyList,
    NoPositive,
}

fn first_positive(l: Vec<i64>) -> Result<i64, MyError> {
    if l.is_empty() {
        Err(MyError::EmptyList)
    } else {
        for elt in l {
            if elt >= 0 {
                return Ok(elt);
            }
        }
        Err(MyError::NoPositive)
    }
}
```

Handling error types

Definition

What can be drawbacks of using error types?

Handling error types

Definition

What can be drawbacks of using error types?

- They're normal values, so we need to handle them.
- If we cannot proceed with a computation, we end up having to manually check for each possible error and exit the function early.

Handling error types

```ocaml
(* OCaml *)
(* val can_fail : unit
    -> (int, e) result *)

(* val can_also_fail : t1
    -> (string, e) result *)

let f () : (unit, e) result =
  let x = can_fail () in
  match x with
  | Error e -> Error e
  | Ok v -> (
    let x = can_also_fail v in
    match x with
    | Error e -> Error e
    | Ok v -> Ok (print_string v)
  )
```

```rust
// Rust
fn can_fail()
    -> Result<i64, E> { ... }

fn can_also_fail(_: i64)
    -> Result<String, E> { ... }

fn f() {
    let x = can_fail();
    match x {
        Err(e) => Err(e),
        Ok(v) => match v {
            Err(e) => Err(e),
            Ok(v) =>
              Ok(println!("{}", v)),
        }
    }
}
```

Handling error types

Most languages with error types provide handling functions and operators.

Using monadic operations:

```
(* OCaml *)
(* val can_fail : unit
    -> (int, e) result *)

(* val can_also_fail : t1
    -> (string, e) result *)

let f () : (unit, e) result =
  let x = can_fail () in
  let x = Result.bind x can_also_fail in
  Result.map print_string x
```

Using the ? “try” operator:

```
// Rust
fn can_fail()
    -> Result<i64, E> { ... }

fn can_also_fail(_: i64)
    -> Result<String, E> { ... }

fn f() {
    let x = can_fail()?;
    let x = can_also_fail(x)?;
    Ok(println!("{}", x))
}
```

Handling error types

Exercise

What if need to early exit from a whole branch of our call tree?

Handling error types

Exercise

What if need to early exit from a whole branch of our call tree?

Error types need to be handled *at every stage*.

- Good to make sure error cases are handled
- But heavy for the developper who needs to explicitly handle them

Exceptions

Definition

Exceptions are data structures encoding exceptionnal conditions.
They can be *thrown* where encountered, which unwinds the call-stack until the nearest appropriate *handler*.

- If no appropriate handler is encountered, the program crashes.
- Exceptions are a special kind of *value*, so they have a *type*.
- Function signatures may of may not indicate if the function can result in an exception
  - Java has *checked exceptions* and *unchecked exceptions*
  - C++ has `noexcept`
  - Most other languages do not have indications for exceptions

Exceptions in Ada

```ada
-- Ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Exceptions;  use Ada.Exceptions;

procedure Be_Careful is
   function Dangerous return Integer is
   begin
      raise Constraint_Error;
      return 42;
   end Dangerous;

begin
   declare
      A : Integer;
   begin
      A := Dangerous;
      Put_Line (Integer'Image (A));
   exception
      when Constraint_Error =>
         Put_Line ("error!");
   end;
end Be_Careful;
```

Exceptions in Java

```java
// Java
class MyException extends Exception {
  public MyException(String s) {
    super ("Message: " + s);
  }
}

class C {
  public void break() throws MyException {
    throw new MyException("broken!")
  }

  public int break_math() {
    return 42 / 0;
    // ArithmeticError, unchecked exception
  }
}
```

Exceptions as control flow

```ocaml
(* OCaml *)
let list_product l =
  let exception Zero in
  let rec aux = function
    | [] -> 1
    | 0 :: _ -> raise Zero
    | hd :: tl -> hd * aux tl
  in
  try aux l
  with Zero -> 0
```

Exceptions vs Errors

Exercise

What are drawbacks of exceptions? How do they compare to error types?

Exceptions vs Errors

Exercise

What are drawbacks of exceptions? How do they compare to error types?

Error types

- Explicit typing
- No impact on control flow
- Must be handled
- Can result in compilation errors

Exceptions

- No impact on typing (in most cases)
- More flexible control flow
- Should be handled
- Can result in runtime errors

Summary

Two main kinds of errors:

- **Unrecoverable errors** should halt the program. They must be detected and reported by the developper.
  - Assertions
  - Use of `err` in C
  - Use of `panic!` in Rust
  - …
- **Recoverable errors** should not halt the program. They must be detected and handled by the developper.
  - Specific values
  - Option types
  - Error types
  - Exceptions
  - …

Section 2

Design by contract

What is that?

> *"It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?*
>
> *– Charles Antony Richard Hoare*

Goals

In everyday life a service or a product typically comes with a contract or warranty: **an agreement in which one party promises to supply the service or product for the benefit of some other party**.

An effective contract for a service specifies requirements:

- Conditions that the consumer must meet in order for the service to be performed
  $\Rightarrow$ Preconditions
- Condition that the provider must meet in order for the service to be acceptable
  $\Rightarrow$ Postconditions

You’re already using contracts!

Exercise

What implicit contracts are you using all the time?

You're already using contracts!

Exercise

What implicit contracts are you using all the time?

- Deref $\Longrightarrow$ not nullptr
- Array access $\Longrightarrow$ array is big enough
- ...

Contracts

4 main types of contracts:

- Assertions
- Pre-conditions and postconditions of a method
- Class invariants
- Loop invariants

All introduced by Bertrand Meyer's Eiffel language in 1986.

Pre-conditions

Pre-conditions must be fulfill by the client, i.e. based on arguments

```
-- Eiffel
class SHAPE
feature
    xc, yc: INTEGER -- coordinates
    set_x_y(x, y: INTEGER)
        require
            x >= 0 and y >= 0
        do
            xc = x
            yc = y
        end
...
```

Post-conditions

Post-conditions must be fulfill by the provider, i.e. if the client fulfill preconditions, the provider will fulfill postcondiitons.

```
-- Eiffel
class SHAPE
feature
    ...
    set_x_y(x, y: INTEGER)
        require
            x >= 0 and y >= 0
        do
            xc := x
            yc := y
        ensure
            xc = x and yc = y
        end
```

Referencing previous version of an expression

`old x` reference the value of x before the execution of the method

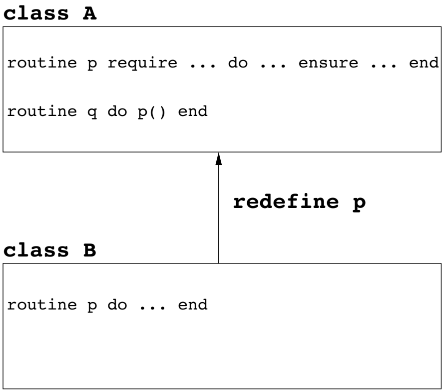
```
-- Eiffel
class RECTANGLE
feature
    width, height: INTEGER
    set_width(w: INTEGER)
        require
            w > 0
        do
            width := w
        ensure
            width = w and height = old height
        end
    ...
```

Stripping Objects

In a postcondition, `strip(x,y,..)` references an object where all attributes *x* and *y*, ... have been removed

```
-- Eiffel
class RECTANGLE
feature
    width, height: INTEGER
    set_width(w: INTEGER)
        -- change the width
        require
            w > 0
        do
            width := w
        ensure
            width = w and strip (width) = old strip (width)
        end
```

Redefinition (1/2)



No need to redefine `require` / `ensure`

$\Rightarrow$ Assertions are inherited.

Redefinition (2/2)

The redefined method must satisfy old assertions but can be more precise:

- Release some preconditions
- Add (Restrict) postconditions

```
-- Eiffel
class B

inherit
  A redefine p end ;

feature
  p is
    require else
      ... -- other restrictions for calls
    do
      ... -- new defintion
    ensure then
      ... -- additionnal postconditions
    end
end -- class
```

Class Invariants

A is an assertion attached to an object. The inherited class also inherits invariants.

```
-- Eiffel
class RECTANGLE
   ...

   invariant
         (xc < 0 implies width > -xc)
      and
         (yc < 0 implies height > -yy)
      and
         width >= 0
      and
         height >= 0

end -- class RECTANGLE
```

Assertions

Can be inserted anywhere in the code.

```
-- Eiffel
-- Code
check
    x > 0
    y < 0 implies width > -y
end
```

Loop (in)variants

Only one (complex) kind of loop in Eiffel

```
-- Eiffel
from
    ... -- initialization
invariant
    ... -- checked each iteration
variant
    ... -- positive and decreasing integer expression
until
    ... -- exit condition
loop
    ... -- loop body
end
```

Contracts in D

```d
// D
int fun(ref int a, int b)
in (a > 0)
in (b >= 0, "b cannot be negative!")
out (r; r > 0, "return must be positive")
out (; a != 0)
{
    // function body
}
```

```d
// D
struct Date
{
    this(int d, int h)
    {
        day = d;    // days are 1..31
        hour = h;   // hours are 0..23
    }

    invariant
    {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24);
    }

  private:
    int day;
    int hour;
}
```

Contracts in C++26 (WIP)

```cpp
// C++
int f(int i)
    pre (i >= 0)
    post (r: r > 0)
{
    contract_assert (i >= 0);
    return i+1;
}
```

Contracts in every programming language

- First-class support in Eiffel, D, C++26, Ada, Kotlin, Clojure, Racket, Scala…
- Libray support in many other
- Almost all languages have `assert`!

Contracts in every programming language

- First-class support in Eiffel, D, C++26, Ada, Kotlin, Clojure, Racket, Scala…
- Libray support in many other
- Almost all languages have `assert`!

Remember: it's better to fail *fast*

- It avoids running in a failure state!
- It makes problems explicit!
- It allows you to find bugs (and thus fix them) more easily!