Algebraic data types

The case against null

Motivation

- PL history (1970's)
- SML, OCaml, Haskell, Scala, F#, Rust, Swift
- Variants
- Null references

User-defined Types

- compound
  - ("Dunedin", 120_000) : string * int
- choice
  - Unemployed | Employee of string * int * string
- recursive and generic
  - type 'a list = Nil | Cons of 'a * 'a list

Why algebraic ? http://blog.lab49.com/archives/3011

Definition

```
type 'a tname = C1
              | C2 of string
              | C3 of 'a * 'a tname
```

OCaml syntax

Constructing values

```
# C1;;
- : 'a tname = C1

# C2 "car";;
- : 'a tname = C2 "car"

# C3 (4.5, C2 "red");;
- : float tname = C3 (4.5, C2 "red")
```

Deconstructing values

via pattern-matching:

```
match v with
  C1 -> ... (* do something for this case *)
| C2(s) -> ...    (* can use s here *)
| C3(x, y) -> ... (* can use x or y here *)
```

try to match v, from top to bottom, against a series of patterns. No fall-through !

Pattern-matching

Pattern matching allows both to peek inside data structures *and* branch depending on what matches

Example 1: cards

```
type color = Black | Red
type suit = Club | Diamond | Heart | Spade
type rank = Jack | Queen | King | Ace | Num of int
type card = rank * suit

let c = (Num 9, Club)
```

Example 1: cards

```
let card_value c =
    match c with
      (Ace, _)     -> 11
    | (Num(v), _) -> v
    | _            -> 10
```

Example 2: Syntax tree

```
type exp = Num of int
         | Plus of exp * exp
         | Mult of exp * exp
```

```
> Plus(Num 5, Mult(Num 4, Num 3))
```

Example 2: Syntax tree

```
let rec eval e =
  match e with
    Num n -> n
  | Plus(e1, e2) -> eval e1 + eval e2
  | Mult(e1, e2) -> eval e1 * eval e2
```

Example 2: Syntax tree

$$\begin{tikzcd}
 & & \text{Plus} \arrow[dl] \arrow[dr] & & \\
5 + 3 * 4 \arrow[r] & \text{Num } 5 & & \text{Mult} \arrow[dl] \arrow[dr] & \\
 & & \text{Num } 3 & & \text{Num } 4
\end{tikzcd}$$

Summary

- Compact notation
- Exhaustiveness check
- Easy to add operations (to existing code)

but…

- Hard to add new variants

OOP

- Easy to add new variants (subclasses)

but…

- Hard to add operations (methods)

Expression problem

http://channel9.msdn.com/Shows/Going+Deep/C9-Lectures-Dr-Ralf-Laemmel-Advanced-Functional-Programming-The-Expression-Problem

Encoding Variants

```
type colour = RGB of int * int * int
            | CMYK of int * int * int * int
```

Encoding Variants: 1

put all fields in one record and use an extra field as a tag

```
class Colour {
    int model; /* RGB = 0, CMYK = 1 */
    int r; int g; int b;
    int c; int m; int y; int k;
}
```

Encoding Variants: 1

Problem:

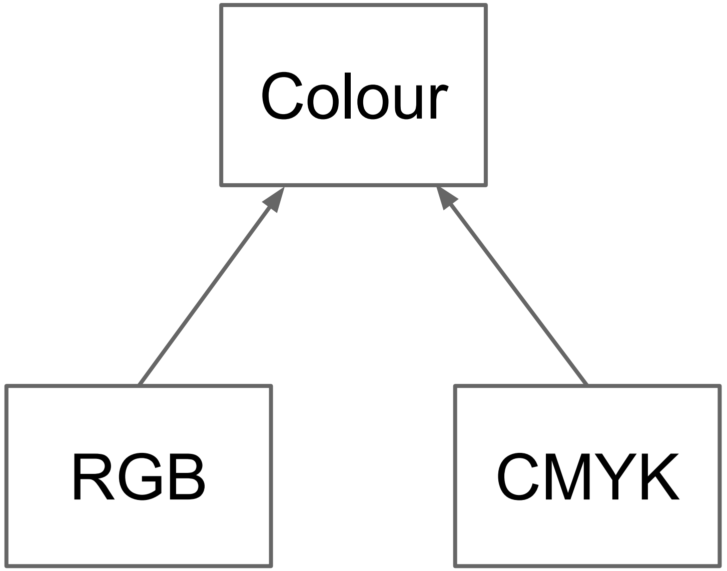
- not efficient (unused fields)
- illegal states are representable:

```
{ model = 0; /* RGB */
  r = 255; g = 0; b = 0;
  c = 255; m = 0; y = 0; k = 0; }
```

Encoding Variants: 2

a class for the type, a subclass for each variant

```
abstract class Colour {}
class RGB extends Colour {
  int r; int g; int b;
}
class CMYK extends Colour {
  int c; int m; int y; int k;
}
```



Null references

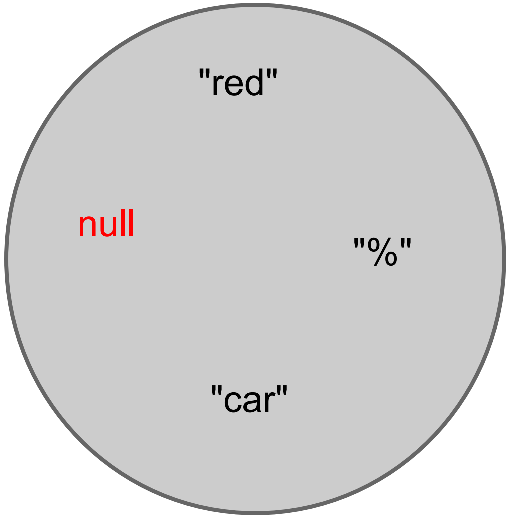
I call it my billion-dollar mistake. It was the invention of the null reference in 1965 [...]

This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

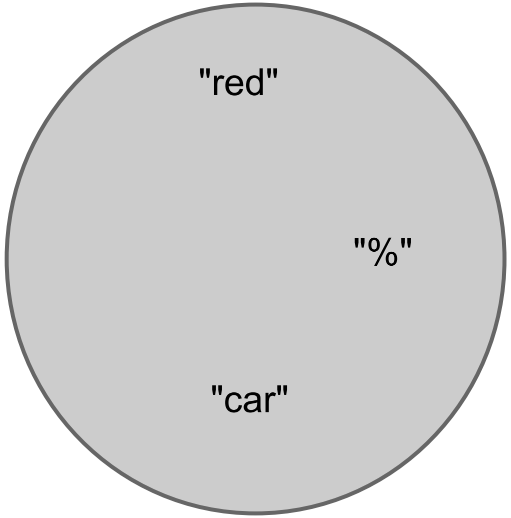
The problem: null is everywhere by default

String in Java



s.size() ?

string in OCaml



size s : int

The solution

Wait for it…

The solution

Yes, no null !

The solution

Tony Hoare, again:

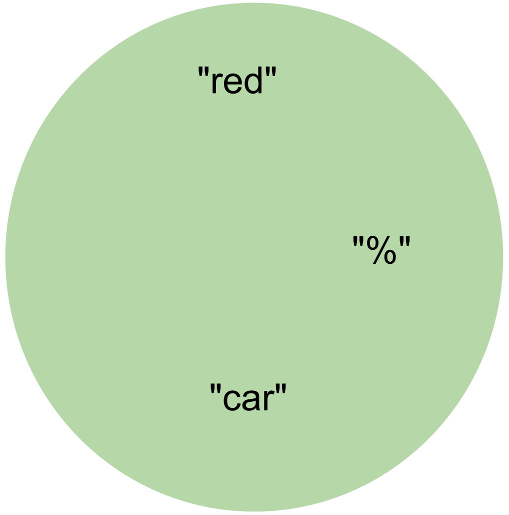
More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

The solution

We want *localized* nulls instead of *pervasive*

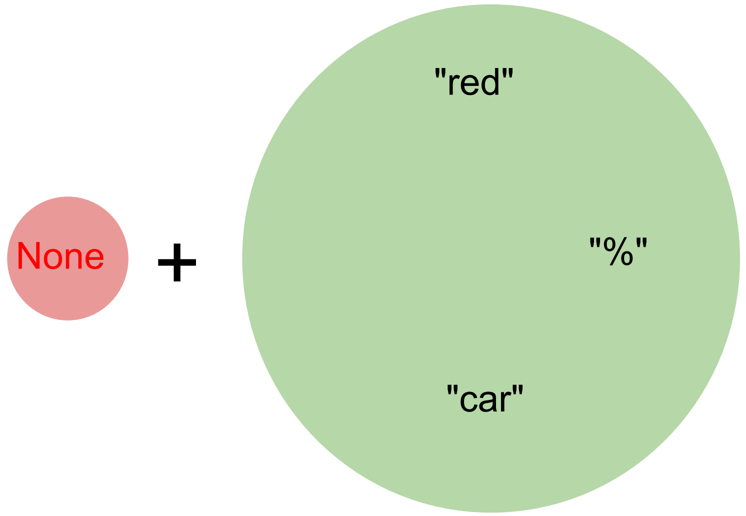
The solution

No null by default...



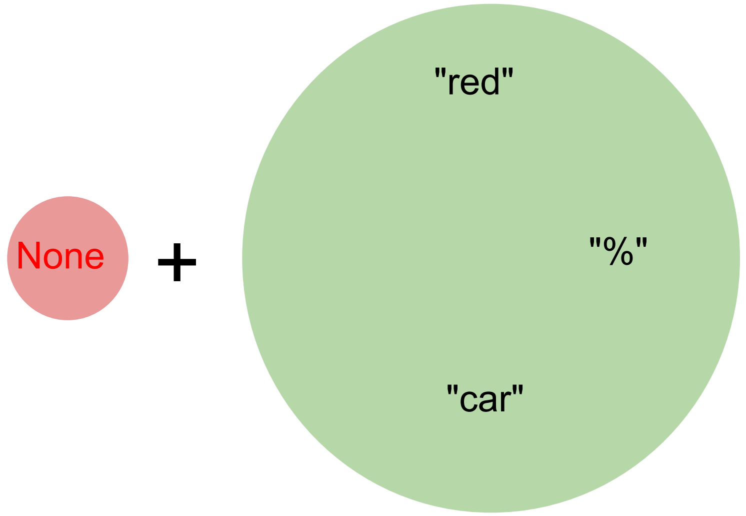
The solution

only add null to a specific type (as a variant), where needed



The solution

```
type maybe_str = None | Some of string
```



Option: a generic nullable type

```
type 'a option = None          (* failure *)
               | Some of 'a    (* success *)
```

```
# Some 5;;
- : int option = Some 5

# Some "alice";;
- : string option = Some "alice"

# None;;
- : 'a option = None
```

Using an optional value

```
match res with
  None    -> …  (* handle error *)
| Some v  -> …  (* do something with v *)
```

Ex 1: lookup

```
let l = [(1,"Joe"); (2, "Carmen"); ...]

let rec lookup(key, lst) =
  if END_OF_LIST then None       (* failure *)
  else if FOUND_V then Some v    (* success *)
  else CARRY_ON_LOOKING
```

Ex 2: List type

```
type 'a list = Nil | Cons of 'a * 'a list
```

Ex 2: List type

```
type 'a option = None | Some of 'a
type 'a list   = Nil  | Cons of 'a * 'a list
type bst       = Leaf | Node of bst * int * bst
```

Is it really better ?

1. when you use an option, you are forced to handle the None case*
2. No pervasive nulls. Once the value is extracted in the Some branch, it cannot be None. *No subsequent check needed.*

3. No more Null Pointer Exceptions* !