Principles of Software Design
Types, Paradigms

Robert Lukoťka
lukotka@dcs.fmph.uniba.sk
www.dcs.fmph.uniba.sk/~lukotka

M-255

Types

What is type system good for?

- `kill(int, int)`
  - `kill(SIGUSR1, pid)`
  - Compiles, runs. May cause an hard to track. Writing tests may be hard.
- `kill(Signal, ProcessId)`
  - `kill(Signal_SIGUSR1), ProcessId(pid))`
  - Compile-time error. Hard to make a bug.

Types

Strings + encoding:

- somewhere you should write a `="lala\aa"`
- somewhere else a `="lala\\aa"`, because `"\"` is a special character.
- We might want to make distinct types for these two to avoid bugs.
- One can also write unit tests to handle this, but if this issue spans multiple units, type checking is not only more reliable but also more efficient way to handle this.

When to check types

When type correctness can be checked

- While writing, your IDE may have this feature (requires static typing)
- Compile time (static typing)
- Run time (dynamic typing)

"Strong" vs "weak" typing, examples of weak typing may include

- "5"+3
- C pointers .. you cast `(void *)` to something all the time.
- many other shortcuts used in various programming languages

Type systems

The types may be very complex. Some type systems are NP-complete.

```
template <int N> struct Factorial
{
    enum { val = Factorial<N-1>::val * N };
};

template<>
struct Factorial<0>
{
    enum { val = 1 };
};
```

Thus types `std::array<int, 6>` and `std::array<int, Factorial<3>::val>` are the same.

Dependent types

Examples from Python:

- Any
- Union
- Tuple, List, Sequence, Iterable
- Optional
- Callable
- ...

You can define your own dependent types:

```
Tuple[int, str, List[int]]
Tuple[int, ...]
```

Type checks

Benefits of type checks:

- Compile time type checking provides faster feedback
- Help document the code.
- Enables better IDE support (autocomplete, even faster feedback).
- Help to maintain cleaner design.

Disadvantages:

- You have to write types.
- You have to create boundaries between typed and untyped code (e.g. using external libraries).

More info on typing in Python and on typing in general

If you are interested check this.

Programming paradigm

A programming paradigm is a style, or "way," of programming.

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

Selected Paradigms

From which elements we compose programs:

- Procedural
- Object oriented
- Functional

How we write program:

- Declarative
- Imperative

Programming Paradigms

Note that there are many other paradigms accounting for various concerns:

- Event-driven programming
- Aspect-oriented programming
- Generic programming
- ...

See e.g. Wikipedia for some random list.

Procedural programming

- Procedure perform operations over data.
- Procedures call other procedures or functions to perform partial tasks.

Object oriented programming

Principles:

- Program is divided into classes. Classes contain both data and behavior.
- Encapsulation
- Polymorphism
- Inheritance

Functional programming

Principles:

- Based on composing pure functions.
  - Same input gives the same output (no internal/global state).
  - No side-effects (throwing exceptions, i/o, ...)
- Higher order functions (functions that take functions as parameters)
- Referential transparency - Immutable data
- Pure functional languages are often accompanied by very strong and complex type systems.

Example - Relation on $\mathbb{R}$

Procedural (Python):

```
def relation_add(rel, el1, el2):
    if (el1, el2) in rel.elements:
        return
    rel.elements.add((el1, el2))
```

Example - Relation on $\mathbb{R}$

OOP (Python)

```
class Relation:
    ...
    def add(self, el1, el2):
        if (el1, el2) in self.elements
            return
        self.elements.add((el1, el2))
```

Example - Relation on $\mathbb{R}$

Functional (Python):

```
class Relation:
    ...
    def add(self, el1, el2):
        if (el1, el2) in self.elements
            return self
        return Relation(self.elements+(el1, el2))
```

Example - non-pure function

Eventually, you need a state. It can be done, but you should be careful with it in FP:

```
def give_counter():
    j=0
    def function_to_return():
        nonlocal j
        j+=1
        return j
    return function_to_return

counter=give_counter()
a=counter() %1
b=counter() %2
```

Example - server performs stuff on database

Procedural:

- You can perform a set of changes in several procedures

O-O:

- You read data, build object structure, then you do something and update what needs to be updated.

Functional:

- Read data, evaluate pure function, write data.
- You can return a procedure that changes the state as required.

Some stuff to be found in functional languages

```
fib = -> n { (n == 0 || n == 1) ? n :
             fib[n - 1] + fib[n - 2] }

s(x) = (1 to x) |> filter (x => x % 2 == 0)
                |> map (x => x * 2)

my_map_and_filter = filter(x => x % 2 == 0)
       . map (x => x * 2)
```

Some stuff to be found in functional languages

Check this (or if it is too abstract, just move on):

- Maybe Monad

Imperative vs declarative

- Imperative - you describe how to do something
- Declarative - you define what you want to get done

The boundary is fuzzy.

Examples: declarative approach

- SQL - you do not say how to join tables
- HTML - you do not say how to render stuff
- Regular expressions
- Constraint programming
- ...

Declarative programming

- You need something that translates what $\rightarrow$ how.
- It is really important that this "something" is very reliable.
- If you manage to do it, you will have a code that is much easier to read and write.
- Hard to do in general, however, it might be reasonable to do this if the scope is small.

Domain specific languages

Watch this:

- M. Fowler: Introduction to Domain Specific Languages

Resources I

- Wikipedia: Procedural programming (includes short description of OO and functional programming)
- M. Fowler: Introduction to Domain Specific Languages

References I

 Ray Toal: Programming Paradigms