Adventures in data types : benefits of the comparative approach in computer science education.

Camille Akmut

June 17, 2019

Abstract

The transition from 'structured programming' to 'object-oriented programming' is a canon of the computer science curriculum; traditionally presented, or tacitly acknowledged, as the transition from beginner to intermediate programmer, the passage from one programming language to multiple ones (covering CS1 and CS2, in various ways). In this other addition to computer science education, we defend the benefits of a comparative approach : knowledge in one language gains the student access to a world of other languages, and ways to model reality. This goes contrary to prevalent methods of focusing on one language to introduce these topics, as common with mainstream, "pure" computer scientists.

Introduction

The merits of the comparative approach are undoubted : from history to anthropology to linguistics and finally computer science, its results are too numerous to ignore. Its great power : helping us break with the taken-for-granted assumptions that underlie our various subjects.

No different in teaching and inversely learning to program,.

In the following, we present an attempt to introduce the topics of 'object-oriented programming' and data modeling more generally :

data types (`Haskell`), struct's (`C`, `Rust`), records (`Pascal`), classes (`Python`, `Ruby`, `Java/Scala`) ...

We take a single, identical and simple example, and model it in a dozen of both common and uncommon programming languages[1], making appear similarities and dissimilarities; but most importantly enabling any student with knowledge in any of these a rapid access to ways of modeling and thinking in others.

In doing so, we took great care in writing *legal* code that was *actually* validated (be it succesfully compiled, or interpreted).[2]

This is one attempt, we hope others will improve on it.

[1] One of them, in fact, `PureScript`, a dialect of `Haskell` for the Web, is still undergoing major revisions at the time of writing.

[2] Such redundant assurances would be unnecessary if not for one too many a lazy computer science professors writing inept code on boards, or in their presentations, let alone their books (and not because they identified with the original hacker tradition); adding great confusion to already considerable one; then earnestly wondering what went wrong. (They were the bug.)

adventures with data types

Data types, classes, struct's, records, interfaces … in various languages

Goals :

1. Print out the whole structure
2. Print out select attributes

Model :

Student

- name (e.g. "Rachel")
- age (e.g. "20")
- grade (e.g. "4.0")

C

```
#include <stdio.h>
#include<string.h>

struct Student {
    char name[50];
    int age;
    float grade;
};

struct Student rachel = {"Rachel", 20, 4.0};

int main(void) {
    printf("%s \n",rachel.name);
    printf("%i \n",rachel.age);
    printf("%f \n",rachel.grade);
}
```

Output :

```
Rachel
20
4.000000
```

Rust

```rust
fn main() {

    #[derive(Debug)]
    struct Student {
        name: String,
        age: u64,
        grade: f64
    }

    let rachel = Student{name: "Rachel".to_string(), age: 10, grade: 4.0};

    println!("{:?}", rachel);

}
```

Output :

```
Student { name: "Rachel", age: 10, grade: 4.0 }
```

Python

```python
class Student():
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

Rachel = Student("Rachel", 20, 4.0)
```

```
print(vars(Rachel))
print(Rachel.name)
```

Output :

```
{'name': 'Rachel', 'age': 20, 'grade': 4.0}
Rachel
```

Ruby

```ruby
class Student
  attr_accessor :name, :age, :grade
  def initialize(name, age, grade)
    @name = name
    @age = age
    @grade = grade
  end
end

rachel = Student.new('Rachel', 20, 4.0)

# rachel.inspect
p rachel
puts rachel.name
```

Output :

```
#<Student:... @name="Rachel", @age=20, @grade=4.0>
Rachel
```

Javascript

```javascript
var rachel = {name: 'Rachel', age: 20, grade: 4.0};
```

```
console.log(rachel);
console.log(rachel.name);
```

Output :

```
{ name: 'Rachel', age: 20, grade: 4 }
Rachel
```

Typescript

(interfaces)

```
interface Student {
    name: String;
    age: Number;
    grade: Number;
}

var rachel: Student = {
  name: "Rachel",
  age: 20,
  grade: 4.0
}

console.log(rachel.name);
console.log(rachel.grade);
```

Output :

```
Rachel
4
```

Haskell

```
data Student = Student String Int Float
```

```
  deriving Show

rachel = Student "Rachel" 20 4.0

main = print(rachel)
```

Output :

```
Student "Rachel" 20 4.0
```

Scala

```
class Student (var name:String, var age:Int, var grade:Double){}

val rachel = new Student("Rachel", 20, 4.0)

println(rachel.name)
```

Output :

```
Rachel
```

Pascal

(record) (and addressing / and its instance)

```
type
    Student = record
        name: string;
        age: integer;
        grade: real;
    end;

var
    rachel: Student;
```

```
begin

rachel.name := 'Rachel';
rachel.age := 40;
rachel.grade := 2.0;

writeln ('Student name : ', rachel.name);

end.
```

Output (Free Pascal Compiler 3.0) :

```
Student name : Rachel
```

Elm

```
student = { name = "Rachel", age = 20, grade = 4.0 }
```

Output :

```
{ age = 20, grade = 4, name = "Rachel" }
    : { age : number, grade : Float, name : String }
```

PureScript

(possibly incomplete)

```
-- (import Prelude) ?

data Student = Student { name :: String, age :: Int, grade :: Number }

showStudent :: Student -> String
showStudent (Student o) = o.name <> ", aged " <> show o.age
```

```
rachel :: Student
rachel = Student { name: "Rachel", age: 20, grade: 1.0 }
```