Category Theory for Programmers

Homework (Session 1)

Bruno Vandekerkhove

Contents

Definition of a Category

Formal Definition

Milewski summarises his informal definition of a category as follows :

> *A category consists of objects and arrows (morphisms). Arrows can be composed, and the composition is associative. Every object has an identity arrow that serves as a unit under composition.*

Let's compare the statement with the original definition of Mac Lane and his colleague Eilenberg [**?**, **?**]:

A category $C = \{A, \alpha\}$ is an aggregate of abstract elements A called the *objects* of the category, and abstract elements α, called *mappings* of the category. Certain pairs of mappings α_1, α_2 uniquely determine a product mapping $\alpha_1\alpha_2$ subject to the first three axioms listed below. Each object A is associated with a unique mapping e_A for which axioms 4-5 have to hold.

1. The triple product $\alpha_3(\alpha_2\alpha_1)$ is defined if and only if $(\alpha_3\alpha_2)\alpha_1$ is defined. When either is defined, the associative law $\alpha_3(\alpha_2\alpha_1) = (\alpha_3\alpha_2)\alpha_1$ holds. This triple product will be written as $\alpha_3\alpha_2\alpha_1$.

2. The triple product $\alpha_3\alpha_2\alpha_1$ is defined whenever both products $\alpha_3\alpha_2$ and $\alpha_2\alpha_1$ are defined.

   Definition : A mapping $e \in C$ will be called an *identity* of C if and only if the existence of any product $e\alpha$ or βe implies that $e\alpha = \alpha$ and $\beta e = \beta$.

3. For each mapping $a \in C$ there is at least one identity $e_1 \in C$ such that αe_1 is defined, and at least one identity $e_2 \in C$ such that $e_2\alpha$ is defined.

4. The mapping e_A corresponding to each object A is an identity.

5. For each identity e of C there is a unique object A of C such that $e_A = \mathrm{e}$.

Comparison

Aside from the naming, the definition also differs in that it is more abstract. The first of the axioms corresponds to associativity of the composition function. The fact that mappings can be composed if the range of one mapping is equal to the domain of the other is stated by a lemma that follows from the axioms. The fact that each object is associated with just one identity follows from axiom 3-5.

A few decades after the formal definition given here, Mac Lane published a book in which an other definition is given [**?**]. There he starts by defining a (meta)graph, builds on that to define metacategories (for which he does use the term morphism for denoting the arrows). Finally he introduces the term category as an interpretation of the axioms of a metacategory within set theory. Because a category is defined as a special kind of directed graph, this automatically answers the last challenge of chapter 1. Bartosz mentions the book in his foreword making it likely that he based his informal definition on this one.

In both of Mac Lane's referenced works he points out that since there's a one-to-one correspondence between objects and identities, one could disregard objects altogether and deal with arrows (morphisms or mappings) only. The resulting axioms are equivalent.

Challenges

Memoization

In `C` macros can be used to define memoized functions, but things quickly get complicated, so I limit myself to functions with one integer argument :

```c
#include <stdio.h>

#define MEMOIZE(return_type, function, argument)                    \
    return_type function_internal(int argument);                    \
    return_type function(int argument) {                            \
        static return_type *results[50];                            \
        if (results[argument] != NULL)                              \
            return *results[argument];                              \
        results[argument] = malloc(sizeof(return_type));            \
        *results[argument] = function_internal(argument);           \
        return *results[argument];                                  \
    }                                                               \
    return_type function_internal(argument)

int fib(int n) {
    return (n <= 2 ? 1 : fib(n-2) + fib(n-1));
}

MEMOIZE(int, memoized_fib, n) {
    return (n <= 2 ? 1 : memoized_fib(n-2) + memoized_fib(n-1));
}

int main(int argc, const char * argv[]) {
    printf("Hello, World!\n");
    printf("%i\n", fib(45));
    printf("%i\n", memoized_fib(45));
    return 0;
}
```

In languages that provide generics and higher-order functions it's easier to do it (here in Swift, not the preferred language for this) :

```
func memoize<I:Hashable,O>(_ function: @escaping ((I) -> O, I) -> O) -> (I) -> O {
    var map = Dictionary<I,O>()
    func internal_function(input: I) -> O {
        print(map)
        if  let value = map[input] {
            return value
        }
        let result = function(internal_function, input)
        map[input] = result
        return result
    }
    return internal_function
}

func fib(_ n: Int) -> Int {
    return (n <= 2 ? 1 : fib(n-1) + fib(n-2))
}

let memoized_fib = memoize {
    memoized_fib, n in
    return (n <= 2 ? 1 : memoized_fib(n-1) + memoized_fib(n-2))
}

print(fib(25)) // 75025, slow
print(memoized_fib(25)) // 75025, fast

import Foundation

func random(_ seeded: Bool) -> Double {
    if  seeded {
        srand48(0)
    }
    return drand48()
}

let memoized_rand = memoize { _, flag in return random(flag) }
memoized_rand(true)
memoized_rand(false) // 0.8404853694114252

print(random(false)) // 0.09637165562356742
print(memoized_rand(false)) // 0.8404853694114252
print(random(true)) // 0.17082803610628972
print(memoized_rand(true)) // 0.17082803610628972
```

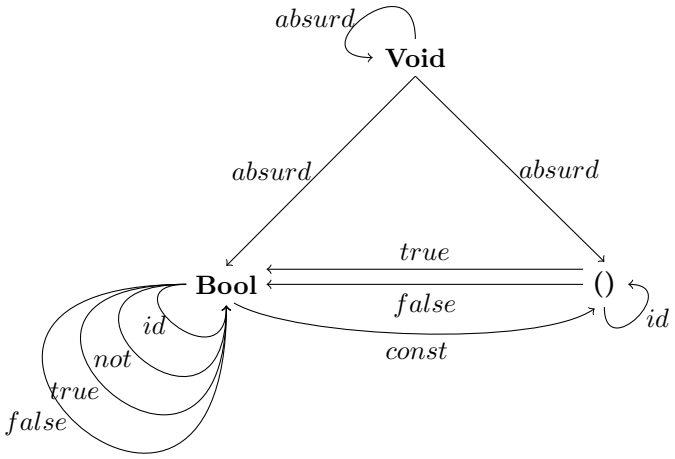
Memoizing a random number generator doesn't work because such a generator is a dirty function (its output varies when the input value doesn't). Making use of a seed which is the basis of the random number generation results in a pure function, for which memoization *does* work.

Pure Functions

The factorial function produces no side effects and returns the same output for a given output at all times. `std::getchar` will return a different output depending on the user's behaviour. `std::cout` prints to the screen which is a side effect. Finally, the use of a static variable means that memory shared between function calls is altered (which is a side effect) and the output happens to depend on the value of the very variable, not just on the input.
In other words, only the factorial function is pure.

Void, Unit & Bool

The category with objects *Void*, *Unit* and *Bool* could be displayed as follows :



Void cannot be returned by any function. It is an initial object so there's only one morphism from **Void** to every object. **Unit** is a terminal object so there's exactly one morphism for every object that points to it.

The Haskell types include the bottom $\perp$ (even **Void**), and the corresponding functions are missing from the diagram. For example, there are 11 functions going from **Bool** to **Bool** (you cannot pattern match the bottom but the `True` and `False` functions may take it as an argument.

The absurd function cannot be called. You can return the bottom with *undefined*.