Category Theory for Programmers

Homework (Session 2)

Bruno Vandekerkhove

Contents

Challenges

Chapter 3

What Kind of Order?

For a set of sets with an *inclusion* relation it holds that $A \leq B, B \leq A \Rightarrow A = B$. If two sets do not share any elements, then they are not related. Which means we're dealing with a partial order.

In `C++` you can use template functions to define functions that take a pointer to any type. A template function `min` that takes two pointers to scalar types, for example. You could pass that function a pointer to an `int` or a `long`, the compiler won't complain.

```
#include <iostream>

template <class T, class U>
auto min(T *a, U *b) {
    return (*a < *b ? *a : *b);
}

int main(int argc, const char * argv[]) {
    int a = 1;
    long b = 2;
    printf("%i", (int)min(&a, &b));
    return 0;
}
```

This does not work in general ; because of the strong typing system, any function that explicitly asks for a pointer to an `int` will not accept a pointer to a `long` unless you typecast. `void` is one exception. If you consider `void` a type, then any other type is one of its subtypes. Inheritance also makes it possible to substitute one type for another, presuming that the substituted class is a parent class and its substitute one of its subclasses (or subtypes). But in no case can you go the other way around ; you cannot hand a pointer to `void` to a function expecting a pointer to an `int`, for example.

All of this implies that - if we go by *naming equivalence* - then we appear to be dealing with a partial order. Naming equivalence means that two `structs` with the exact same structure are considered different types simply because of having a different name (in contrast with *structural equivalence*).

Addition Modulo 3

Adding 0 corresponds to the identity function, adding 1 or 2 are other possible morphisms, and any composition of these leads to the same set of morphisms (for example, composing the addition of 1 and 2 gives the identity function).

$$\begin{tikzcd}
\mathbb{N} \arrow[loop, "id"] \arrow[loop, "add1mod3"] \arrow[loop, "add2mod3"]
\end{tikzcd}$$

Chapter 4

It is asked to construct and implement the Kleiski category for partial functions.

Construction

The objects of the category are the types of the programming language.

The (partial) functions take a value of a given class and return an optional. Composition of functions works as follows. The first function is called on the input. If the result is valid (i.e. a valid optional is returned) then the second function is called on the value of the returned optional. Finally, the result of this second function call is returned. If the result of the first function call wasn't valid, an invalid optional is returned. The output class of the first function has to be the same as the input class of the second one.

The identity morphism for an optional of a given class returns the optional itself.

Implementation

Below you can find an implementation in Haskell.

```haskell
data Optional a = Valid a | Invalid deriving Show

(>=>) :: (a -> Optional b) -> (b -> Optional c) -> (a -> Optional c)
(>=>) f g = \x ->
    let y = f x
    in  case y of
        Valid z -> g z
        _ -> Invalid

return :: a -> Optional a
return x = Valid x
```

Composition

In the implementation hereafter, calling `safe_root_reciprocal` will result in an `Invalid` value if the input argument is either negative or equal to zero.

```haskell
safe_root :: Double -> Optional Double
safe_root x = if (x >= 0) then Valid (sqrt(x)) else Invalid

safe_reciprocal :: Double -> Optional Double
safe_reciprocal x = if (x /= 0) then Valid (1/x) else Invalid

safe_root_reciprocal = safe_reciprocal >=> safe_root
```

Chapter 5

Product & Coproduct in Posets

We start from any given poset with a relation denoted with 'less than'. While a preorder is a 'thin' category (there's at most one morphism from any object to an other one), a partial order also imposes the absence of cycles. Let's consider two objects A and B. We want a product C which has projections to A and B (this means it is related to them or 'less than' them). Let's say there is an other object C' which also has such projections.

$$\begin{tikzcd}
 & C' \arrow[d, "m"'] \arrow[ddl, bend right] \arrow[ddr, bend left] & \\
 & C \arrow[dl] \arrow[dr] & \\
A & & B
\end{tikzcd}$$

Then by the definition of a product there must be a unique morphism from C' to C (C' is 'less than' C). This functions as a factorising function m. Meaning that C is a 'better' fit for a product than C'. Or in general, the product of two objects in a poset is the 'largest' object that is smaller than both of these objects. Because the product in categories is unique up to isomorphism, the product in a poset must be unique.

If we were to deal with a total order (in which any two objects including A and B are related), then the product of two objects would be the 'smallest' of the two.

$$\begin{tikzcd}
C' \arrow[d, "m"'] \arrow[rd] & \\
A \arrow[r] \arrow[loop left, "id"] & B
\end{tikzcd}$$

The same reasoning can be applied to the coproduct, where the resulting object is the ‘smallest‘ of all objects that is ‘larger’ than the two given objects. Or the supremum, if you will.

Better Coproducts (`Either` versus `int`)

It’s easy to use Milewski’s own factoriser function to show that you can factorise the injections to `int` :

```
-- Create factorizer

i :: Int -> Int
i = id

j :: Bool -> Int
j b = if b then 1 else 0

factorizer :: (a -> c) -> (b -> c) -> Either a b -> c
factorizer i j (Left a) = i a
factorizer i j (Right b) = j b

m = factorizer i j

-- GHCI
-- m (Left 1) --> 1
-- m (Right True) --> 1
-- m (Right False) --> 0
```

Because of the existence of such a factoriser it’s clear that `Either` is a better coproduct. You cannot do the same the other way round, because 0 and 1 represent both integers and booleans (you can’t factorise). I’d argue that since `int` has limited range, adding 2 to positive integers wouldn’t solve this problem ; it may lead to an overflow so the injections aren’t actually functions.