The Bridge between Mathematical Models of Physics and Generic Simulations

Dániel Berényi[1], **Gábor Lehel**[2]
[1]Wigner RCP, Budapest, [2]Eötvös University, Budapest

Abstract
We would like to draw attention to the fact that abstractions related to logic and mathematical models of physics soon going to be necessary to the development of generic high-performance simulations to advance computational physics. The common language to describe and formulate these is already available in some high-level languages and the main cornerstones are rooting in category theory that in turn again related to the basic foundations of mathematics and physics. The types that many people thought are just used for differentiating integers from floating point numbers in computer programs have grown not just to give the main structure of modern computer programs but recent research is focused on founding mathematics and physics on them.

1. Introduction

Axiomatic foundations of mathematics on set theory was a great program started at the beginning of the last century by David Hilbert. Despite being proved to be impossible by Gödel it had a great influence on the later development of mathematical logic, proof theory, computer science and later type theory. The latter started to develop in parallel from avoiding the Russell paradox in logic systems. Hilbert's student Haskell Curry come to the observation that propositions correspond to types and proofs of those propositions to programs inhabiting the corresponding types; logical connectives between propositions (for example, logical conjunction $A \wedge B$) to type formers (for example, the pair type `(A, B)`) [1]. Roughly speaking this establishes a direct relation between verification of a constructive logical proof and successful compilation of a computer program written in a strongly typed programming language. Type theory then continued to investigate more and more complex type systems, their expressive power, and their connection to (higher-order) logic. Types slowly started to diffuse into mainstream programming languages during the dawn of the digital computer era, and with the spread of computer programming the evolution of programming languages advanced and research in functional programming soon started to realize such type systems in the form of compilers. However performance characteristics, division between academic and industrial attitudes and implementation maturity polarized the programming community: production and performance oriented programmers traditionally developed in lower level and less typed languages, while stronger type systems were mainly remained in the realm of academic interest were more focused on advanced optimization and generic (reusable) development techniques. However in recent years high profile use cases showed that functional programming is necessary and more scalable than traditional solutions because of its more natural composability. This was followed by the gradual adaptation of features from high-level languages like lambda functions (anonymous functions), generics (templates) and more recently traits (concepts).

Meanwhile, Category Theory appeared at the end of the 1940-ties, as a continuation of the work of Emmy Noether, but was first investigated in the context of algebraic topology and homological algebra [2]. Later connections to deductive systems and type theory was established, and the concept of toposes paved the way to the formalization of set theory and foundations of mathematics via categories. This line continued to evolve, and we only note [3] as recent research direction into founding physics on this concept. More recently category theory found application to functional programming, and started to play an essential role in designing advanced functional libraries [4, 5].

Yet another direction is intuitionistic type theory (Martin-Löf type theory [6]) which was introduced in the 70ties as a constructive alternative foundation of mathematics, building on top of the Curry-Howard correspondence, introducing dependent types and universal quantification amongst other things; a less ambitious construction known as System F (Girard–Reynolds polymorphic lambda calculus) is at the core of most of today's functional programming languages, such as Haskell and the ML family. It was shortly understood in category theory via locally cartesian closed categories [7]. Further developments resulted in computer proof assistants and languages that are widely utilized today even in industrial applications. Investigation of the decidable intensional type theory lead to the advent of homotopy type theory, and recently the univalent foundations project [8, 9].

From this introduction it is evident, that logic, programming, and mathematics are deeply related, and useful abstractions discovered in one or another can be applied in all of them to advance thinking and development. In the following sections we would like to present a particular example from different viewpoints to show how these ideas can be utilized to advance the development of generic simulation tools of computation physics.

2. Category Theory

The concept is usually attributed to Samuel Eilenberg and Saunders Mac Lane, who developed the most important concepts in the field. For mathematicians the traditional reference is [10], otherwise we recommend [11, 12]. Here we only review the most important basic properties of categories.

A category is a collection of objects and a collection of morphisms that take objects into objects. We may denote morphisms with an arrow between objects: $x \to y$. Also, a category is equipped with a binary operation called composition ($\circ$) of morphisms that obeys two rules:

- Associativity: if $f: x \to y,\ g: y \to z,\ h: z \to w$, then $h \circ (g \circ f) = (h \circ g) \circ f$,
- Identity: for each object x, there exist a morphism $1_x: x \to x$ such that: $1_x \circ f = f$ and $g \circ 1_x = g$, for any morphisms f and g.

For our purposes one more construction is needed, called a functor, that describes mapping a category to another. Let C and D denote two categories. The functor F from C to D associates each object $x \in C$ to an object $F(x) \in D$, and each morphism $f: x \to y \in C$ to a morphism $F(f): F(x) \to F(y) \in D$ such that it preserves the identity and the composition property.

The properties above are so general, that is not surprising, that they show up in many different scientific disciplines. In fact, one can identify the elements of the Curry-Howard correspondence within category theory, and then recognize the same features in Quantum Mechanics (Feynman diagrams) and Topology (cobordisms). For an enlightening review and introduction we recommend [13].

3. Types and Programming Languages

In this section we introduce some aspects of types in programming languages and explain the syntax that we'll use later to express program code examples. We will provide most examples in Haskell due to its concise and close-to-mathematics syntax, and in C++ due to its abundance in modern scientific computing.

From the practical point of view, types were added to the early programming languages to distinguish different variable types: some functions are not valid on some type of variables, for example it is not meaningful to divide a number by a character string. Thus, type annotations were added to variables.

It is immediately evident that functions are something special, so they need a special pattern to distinguish them from ordinary variables. For example a simple squaring function would look like:

In Haskell:

```
sq :: int -> int

sq x = x*x
```

In C++:

```
int sq(int x){ return x*x; }
```

In Haskell the pattern `int -> int` is called the signature (return type is the rightmost type) of the function: it takes an integer and produces an integer. In C++, signatures appear in the form: `int(int)`, where the return type is the leftmost type.

Many languages allow abstractions over types. If we would like to abstract our square function over types we'd write the function declarations as:

In Haskell: `sq :: a -> a`

In C++: `template<typename A> A sq(A);`

'`a`' or '`A`' is now a type parameter, that must be substituted with a known type before using the function. This construction may be called a generic function. In mathematical notation this means a universal quantification: the function may accept all possible types. Types can be also parametrized over types:

In Haskell: `data Vector a = ...`

In C++: `template<typename A> class Vector { ... };`

The same capability exist in other languages as well: Java/C#/Rust calls these generic types.

These constructions can be understood one level higher. The type of a type is a kind. Haskell denotes kinds with `*`. Now one may ask, what is the type of Vector? Vector takes a type (`a`) and creates a new type (`Vector a`), so it is a `*` → `*`. In C++ the nearest syntax is `template<typename>`.

However, while simple type-parameterization is sufficient for expressing functions whose behaviour is completely uniform over all possible types (parametric polymorphism), further tools are required if we wish to express functions whose behaviour depends on type-specific structure, such as an equivalence relation, ordering, or numerical operations on that type (ad-hoc polymorphism). The Haskell language adopted type classes, a form of limited user-controlled overloading, by which a set of functions (in this context: *methods*) with specified type signatures can be associated with types claiming membership in the class, for this purpose. (The `class` specifies the signatures; `instances` for particular types provide the implementations). C++ has the reverse problem of completely *unrestricted* overloading, which makes it difficult to precisely specify the interface of a function template within C++ code itself, and leads to inscrutable error messages when its implicit requirements are not met; it is however planning to adopt a similar solution in the form of *concepts* [14]. Similar ideas have also made their way to Java and C# (`interfaces`) as well as Rust (`traits`).

Type system constructs can be also understood in the framework of category theory: the types expressible in a language are the objects, and form a category. In the literature, short names are given to these, for example the category of Haskell types is known as “Hask”. The morphisms on the types are the functions, and the binary operation is the function composition. We will see at the end of the next section how functors appear in Haskell.

4. Covariance and Contravariance

We would like to illustrate the connection between mathematics, physics and functional programming by demonstrating how the concept of covariance and contravariance shows up in these subjects connected by category theory.

Changing a base in a vector space is an important transformation in virtually all areas in physics and related mathematics. For a given vector space V over field F, one can choose a Basis B that is represented by an ordered set of basis vectors $\{e_{ij}\}$. Then all vectors $\vec{v} \in V$ can be represented by their components in the respective basis: $\vec{v} = e_{ji} v_i$. Now if we would like to change the basis via a transformation $T: B_1 \rightarrow B_2$ such that $e^2_{ji} = e^1_{jk} M_{ki}$, but keep the represented vector $\vec{v}$ the same we obtain the identity:

$$e^1_{jk} v^1_k = \vec{v} = e^2_{ji} v^2_i = e^1_{jk} M_{ki} v^2_i$$

Assuming that M_{ki} is invertible, we can express the transformation of the coordinates of $\vec{v}$:

$$v^2_i = {M_{ij}}^{-1} v^1_j$$

This transformation property is called contravariant, because the components transform by the inverse of the transformation matrix. On the other hand, if one considers linear functionals on the vector space V over field F, $\varphi: V \rightarrow F$, they can be represented by their components also: φ_i and the inner product can be written: $\varphi(\vec{v}) = \varphi_i\, v_i$. The inner product to be invariant under the basis change: $\varphi(\vec{v}) = \varphi^1_i\, v^1_i = \varphi^2_i\, v^2_i$ we need $\varphi^2_i = \varphi^1_k M_{ki}$ to cancel the effects of the coordinate components coefficient matrix: $\varphi(\vec{v}) = \varphi^1_k M_{ki} {M_{ij}}^{-1} v^1_j$. This transformation property is called covariant.

In category theory functors are distinguished based on how they transform compositions of morphisms. Let A, B, C denote different objects, $f: A \to B$ and $g: B \to C$, denote morphisms and F, G denote two functors. Then F is called *covariant* if

$$F(g \circ f) = F(g) \circ F(f)$$

holds, and G is called *contravariant* if

$$G(g \circ f) = G(f) \circ G(g)$$

holds.

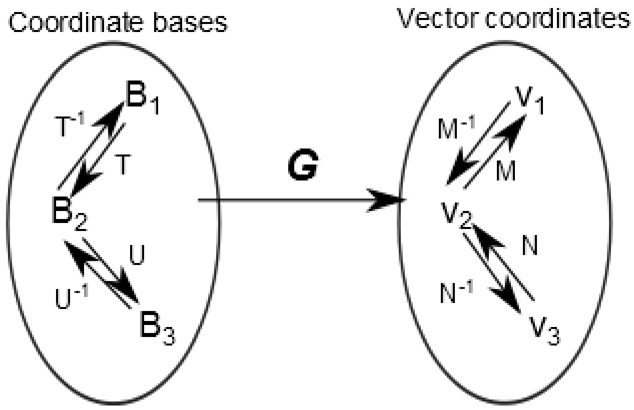

Figure 1. The functor G maps from the category of coordinate bases to the category of vector coordinates, reversing the directions of arrows.

Let's take basis change transformations as morphisms on bases, let the functors map the bases to their coordinate representations and consider the compositions of successive basis changes. Let the basis change $T: B_1 \to B_2$ be represented by the matrix M, and a successive change $U: B_2 \to B_3$ represented by N. For coordinate representations of bases the transformations act as:

$e_{ji}^2 = e_{jk}^1 M_{ki}$ and $e_{ji}^3 = e_{jk}^2 N_{ki}$ and finally: $e_{ji}^3 = e_{jk}^1 M_{kl} N_{li}$.

The same can be shown for linear functionals. For these the functor F is covariant:

$$F(U \circ T) = F(U) \circ F(T) \longrightarrow MN$$

However for coordinate vector components:

$$v_i^2 = M_{ij}{}^{-1} v_j^1 \quad \text{and} \quad v_i^3 = N_{ij}{}^{-1} v_j^2 \text{ and finally } v_i^3 = N_{ij}{}^{-1} M_{jk}{}^{-1} v_k^1.$$

So the functor G mapping from bases to coordinate representations of vectors is contravariant since:

$$G(U \circ T) = G(T) \circ G(U) \longrightarrow N^{-1} M^{-1}$$

In other words, the representation of the composition of basis changes accumulating in the reverse order in the matrix product that will transform vector components. Also, contravariant functors reverse the direction of morphisms that manifests here in the inversion of the transformation matrices, see Figure 1.

The maps represented by functors are quite often encountered in functional programming. Let's consider the category Hask: in the Haskell standard library we have[1]:

[1] actual Haskell syntax slightly differs: type variables have lowercase names.

```
class Functor G where
      fmap :: (B1 → B2) → (G B1 → G B2)
```

Which is the type class of endofunctors (a functor from a category to that same category) on Hask. Here a `Functor G` takes types `B1` to types `G B1`, and its method `fmap` takes morphisms from `B1` → `B2` to morphisms from `G B1` → `G B2`.

We may also have

```
class Contravariant F where
      contramap :: (B2 → B1) → (F B1 → F B2)
```

In the context of the above example this means that an `fmap` takes an abstract basis change function `(B1 → B2)` and then it can produce a coordinate representation of it in the form of `G B1 → G B2`, for instance the matrix M.

For a contravariant functor like the one mentioned above the `contramap` takes the inverse of the original basis change function `(B1 → B2)` that is: `(B2 → B1)` and can use it to produce the transformation of vector components `(F B1 → F B2)`, for instance the matrix M^{-1}.

In a more general programming view covariant functors can be seen as parameterisable producers: the user can choose a type and provide a function that produces that type of output, and then can use fmap to apply this function on a functor instance. Contravariants are the opposite: they are parameterisable consumers, now the user can parametrize what values should the functor instance accept and work on.

5. Application to programming: the case of a linear algebra library

Categories and category theoretic design also started to dominate in the development of functional programming libraries. To see how they help to guide abstraction, we go through a series of programming tasks that eventually lead to the need of functors and show how the abstraction provided by the `fmap` method is a valuable building block of today's parallel algorithms.

As a very basic example, consider the problem of implementing the multiplication of a vector or matrix by a scalar. The traditional solution in a performance critical scientific context was to develop the code in FORTRAN or C. In the latter an implementation would look something like the following[2]:

```
void mulByScalar(double scalar, double* vct, int size)
{
    for(int i=0; i<size; i=i+1)
    {
        vct[i] = vct[i] * scalar;
    }
}
```

[2] In the following we will not elaborate on different memory management techniques and choose the syntactically simplest representation.

The function receives the value of the scalar represented by a double precision floating point number, a pointer to the beginning of a (double precision) data in memory that is the vector argument and an integral value that represents the number of components of the vector. The body of the function then proceeds with a loop that for each component of the vector performs the multiplication by the scalar. While similar behaviour can be implemented in many slightly different ways, we would like to draw attention to a few important observations. If one considers some more use cases, namely division of a vector by a scalar and both of them for different types of components (integral or floating point) it is easy to see that almost the same code will be written over and over again to provide the necessary functions. Unfortunately the C language does not flexible enough to solve these problems, so we turn to C++.

For the sake of compactness of the next examples let's assume that a `Vector` object exists, that takes care of the memory management (allocates an integer number of entries on construction), and provides an array subscript operator `[•]` and a `size()` member function that returns the number of components. At this point, let's assume it stores some kind of fixed type, say `double`. Moreover let's change the semantics, so that our function creates a new `Vector` instance:

```
Vector mulByScalar(double scalar, Vector v)
{
    Vector r(v.size());
    for(int i=0; i<r.size(); i=i+1)
    {
        r[i] = v[i] * scalar;
    }
    return r;
}
```

At first the new `Vector` instance is allocated to the same size as the input vector, the `for` loop is just like the previous case, the only difference is at the end, where the new `Vector` object `r` is returned.

Now let's abstract over the division and multiplication part of the problem. In C++ we can generalize a function over types and this construct is called function templates in the language. Consider the following:

```
template<typename F>
Vector binaryOpOnVector(F f, double scalar, Vector v)
{
    Vector r(v.size());
    for(int i=0; i<r.size(); i=i+1)
    {
        r[i] = f(v[i], scalar);
    }
    return r;
}
```

The function now receives a new argument `f`, that has type `F`. `f` must be a function that can be called with two arguments otherwise the above code would be incorrect. We can also observe, that we can achieve the same functionality by applying a one argument function to the components that we got from a partial application of a binary operator one step before, e. g. (scalar * •) is such a partially applied operator. We can take advantage of lambda functions introduced in C++11 to express such functions in a short way. They can capture variables, and use them later, when called, so we can reduce the function invocation as follows:

```
Vector mulByScalar(double scalar, Vector v)
{
    return almost_fmap( [=](double x){ return scalar * x; }, v );
}
```

where `[=](double x){ return scalar * x; }` is a lambda function that is equivalent to the partially applied operator (scalar * •) still waiting an argument before it can be evaluated. The looping implementation is now the application of a single argument function:

```
template<typename F>
Vector almost_fmap(F f, Vector v)
{
    Vector r(v.size());
    for(int i=0; i<r.size(); i=i+1)
    {
        r[i] = f(v[i]);
    }
    return r;
}
```

As C++ language implements argument type deduction for function template arguments, so we don't have to provide the `F` template argument explicitly to `almost_fmap`[3]. Easy to see, that for division only the lambda inner operator needs to be changed and there is no need to rewrite the complete implementation of `almost_fmap` for both of them.

The next step is to abstract over the scalar and vector component types: we would like to use the same code for integer (`int`), single precision floating point (`float`) and double precision floating point (`double`) arguments. This need change to the `Vector` class so that it will be dependent on a template parameter of the stored type: `Vector<T>` as well as for the `almost_fmap`. Note, that we can now also express a change in the return type that may be induced by the function `f:T->R`, that's why another template parameter R is being introduced[4]:

```
template<typename R, typename F, typename T>
Vector<R> fmap(F f, Vector<T> v)
{
    Vector<R> r(v.size());
    for(int i=0; i<r.size(); i=i+1)
    {
        r[i] = f(v[i]);
    }
    return r;
}
```

[3] In fact, for lambda functions we couldn't have provided that.

[4] We could calculate the return type of the function via `decltype` since C++11, but we didn't want to complicate the example.

Now, the function can accept any type instances, as far as the function call is compatible with them. The interface implementation now has to be changed accordingly:

```
template<typename R, typename T>
Vector<R> mulByScalar(T scalar, Vector<T> v)
{
    return fmap<R>( [=](T x){ return scalar * x; }, v );
}
```

At this point it should be clear: what we just implemented is the `fmap` method described at the end of the last section. So `Vector` is a (covariant-, endo-)functor mapping from the category of the C++ types to the C++ types, and fmap is capable of applying a simple function over such types to the mapped category, that now represents the elements of the `Vector`. In a more general view one may see a functor as a type embedded inside a context, and fmap can apply a plain function inside that context. In Haskell one would make `Vector` to be an instance of the `Functor` typeclass. In C++ this semantic is not yet available, but with the expected adaptation of the C++17 standard where concepts are expected to provide exactly this behaviour there will be a standard way to express this pattern [14].

It is hard to overemphasize the importance of the concept: it took many years for developers to realize the power of abstractions like this, but fmap (usually known as simply map, or transform) became one of the most important building blocks in programming ranging from simple applications to distributed computation systems. If someone was coming from category theory or functional programming it might have been evident to start with such representation immediately saving hundreds of development hours spent on rewriting and designing. It is instructive to note, that all the usual vector-matrix operations can be described with three functional building blocks: fmap, fold and zipWith, where fold generalizes the concept of summarizing a group of elements into one representative with a binary operation and zipWith captures the element wise application of n-ary functions.

Computational physics is growing to be one of the most important applications of programming in physics. Rapid parametrization and investigation of theories and simulations demand highly generic codes and expressive programming languages. Category theoretical thinking is already a huge advantage and soon will be unavoidable to understand and design such programs.

6. Conclusion

In this work we gave a simple example of how category theory appears in linear algebra that is one of the main pillars of physics, and then showed how successive generalizations a computer program motivated by reducing code duplication and improving flexibility resulted in the manifestation of the same category theoretical concept in a high-level programming language that is widely used to develop performance oriented physics simulations.

While we could only barely scratch the tip of the iceberg with these examples due to the limited scope of this article but it must be noted that category theoretical concepts in programming are being applied in way more advanced and sophisticated use cases. We can only give some references to the interested reader [4, 5].

While the demand for developers and scientists with these kind of skills is rapidly increasing, natural science courses yet to integrate category theoretical attitudes.

7. Acknowledgements

Parts of this work was done in the Wigner GPU Laboratory. D. B. acknowledges the support of the Hungarian OTKA Grant No.: NK 106119.

8. References

[1] P. Wadler, Propositions as Types, http://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/propositions-as-types.pdf

[2] J.-P. Marquis, Category Theory, The Stanford Encyclopedia of Philosophy (Winter 2014 Edition), Edward N. Zalta (ed.), http://plato.stanford.edu/archives/win2014/entries/category-theory

[3] U. Schreiber, Differential cohomology in a cohesive infinity-topos, http://arxiv.org/abs/1310.7930

[4] E. Meijer, M. Fokkinga, R. Paterson, Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, In Conf. Functional Programming and Computer Architecture, 124-144, Springer-Verlag, 1991.

[5] R. Hinze, N. Wu, J. Gibbons, Unifying structured recursion schemes, ACM SIGPLAN Notices - ICFP '13, Vol. 48 Issue 9, pp. 209-220, 2013.

[6] P. Martin-Löf, Intuitionistic Type Theory, Bibliopolis, 1984, ISBN 88-7088-105-9

[7] R. A. G. Seely, Locally cartesian closed categories and type theory, Math. Proc. Camb. Phil. Soc. (1984) 95.

[8] The Univalent Foundations Program, Homotopy Type Theory: Univalent Foundations of Mathematics, Institute for Advanced Study, 2013, http://homotopytypetheory.org/book

[9] V. Voevodsky, Univalent Foundations Project, http://www.math.ias.edu/vladimir/files/univalent_foundations_project.pdf

[10] S. Mac Lane, Categories for the working mathematician, Springer, 1972, ISBN 0-387-98403-8

[11] B. Coecke, Introducing categories to the practicing physicist, 2008, http://arxiv.org/abs/0808.1032

[12] B. Coecke, E. O. Paquette, Categories for the practising physicist, 2009, http://arxiv.org/abs/0905.3010

[13] J. C. Baez, M. Stay, Physics, Topology, Logic and Computation: A Rosetta Stone, New Structures for Physics, ed. Bob Coecke, Lecture Notes in Physics vol. 813, Springer, Berlin, 2011, pp. 95-174. http://arxiv.org/abs/0903.0340

[14] B. Stroustrup, A. Sutton, A Concept Design for the STL, 2012, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf