Categorical models for BigData

Laurent Thiry, Heng Zhao, Michel Hassenforder
IRIMAS
Université de Haute Alsace
Mulhouse, France
$\{$*laurent.thiry, heng.zhao, michel.hassenforder*$\}$*@uha.fr*

***Abstract*—This paper shows how concepts coming from category theory associated to a functional programming language can help to formalize and reason about data and get efficient programs in a BigData context. More precisely, it shows how data structures can be modeled by functors related by natural transformations (and isomorphisms). The transformation functions can then serve to shift a data structure and then get another program (eventually reducing time complexity). The paper then explains the main concepts of the theory, how to apply them and gives an application to a concrete database and the performances obtained.**

***Keywords*-Category theory, BigData, modeling, performance**

I. Introduction

BigData is centered on large amount of data what directly impacts the performances of the programs (e.g. to query a specific information) and then requires specific architectures to improve them [1], e.g. use of graph databases or distributed concurrent computations. If a lot of technologies are available today to put BigData into practice, theories usable to well understand the benefits/limitations of each architecture, to identify possible improvements or means to combine them are more rare [2]. In this context, the paper presents the capabilities offered by category theory together with a functional programming language (to easily implement the concepts and facilitate experimentations) to solve this limitation. In particular, it explains how functors can serve to model data structures (e.g. various representations of graphs) and natural transformations to both change data structures and shift programs applicable to a particular data structure to another program for an other data structure. The concept of natural isomorphism is then used to prove that two data structures represent the same information, or that two programs are equivalent. Finally, the equations representing the programs can serve to calculate computation steps (time complexity) and compare the performances of two equivalent programs, then show that a natural transformation is an optimization.

The paper is divided into 3 parts. Part II presents the main elements from category theory and discusses about its possible interests in a BigData context (in particular to study performances and optimizations). Part III proposes a concrete application on how to apply the theory and translate the models into a functional programming language. This part also develops the example of various representations of a graph database, then the various representations of a query and analyzes the performances obtained for each ones. Part IV concludes by summarizing the main elements presented and gives some of the perspectives considered.

II. Categories, BigData and performance

Category theory is a field of mathematics that deals with "structures". More precisely, a category is defined by a set of objects, a set of morphisms between this objects, a composition operator written $(\circ)$ for morphisms and an identity morphism for each object [3]. The composition is associative and has id as neutral element. A concrete example is given by the category $\mathcal{S}et = (X_i, f_j : X_k \to X_l, \circ, id_i)$ where objects correspond to sets $(X_i)_{i \in I}$, and morphisms to functions $(f_j)_{j \in J}$. This category can be easily related to (functional) programs by considering that sets model basic datatypes (e.g. boolean, integer, etc.), and morphisms (e.g. f_j) correspond to programs with a parameter X_k and a result X_l, [4].

A functor F is a structure preserving map between two categories (i.e. it preserves composition $F(f_1 \circ f_2) = F(f_1) \circ F(f_2)$, and identities $F(id_{X_i}) = id_{F(X_i)}$). An example of a functor is the powerset $\mathcal{P} : \mathcal{S}et \to \mathcal{S}et$ with $\mathcal{P}(X_i)$ the set of subsets of X_i and $\mathcal{P}(f_i)\{x_1, ..., x_n\} = \{f_i(x_1), ..., f_i(x_n)\}$. By considering programs, this one can serve to model collections and simple transformations ($\mathcal{P}(f_i)$ being viewed as a loop applying f_i to the elements of a set). Another example is the product (bi)functor $X_i \times X_j$ with $(f_i \times f_j)(x_i, x_j) = (f_i(x_i), f_j(x_j))$. By considering programs, this one can serve to model records.

The preceding functors can be composed to model more complex data structures. For instance, a directed unlabeled graph can be represented by a set of edges and a functor $G(N) = \mathcal{P}(N \times N)$ where N represents a set of nodes and $N \times N$ edges. With a function $f : N \to N'$, we can define a graph morphism $m(f) = \mathcal{P}(f \times f)$ that changes the nodes by preserving the structure of the graph (i.e. if (x, y) is an edge of g then $(f(x), f(y))$ is an edge of $m(g)$). $m(id_N)$ is a identity morphism, morphisms are composable (i.e. $m(f \circ g) = m(f) \circ m(g)$) what makes the set of graphs and morphisms another example of a category $\mathcal{G}raph$.

Graphs play an important role in BigData community and have many applications (see, for instance, Neo4J toolkit[1]), [5]. Querying a particular information then consists in finding a morphism from the graph representing a query to a graph database [6]. Indeed, a query such as "(X isa man) and (X worksfor irimas)" can be viewed as a labeled graph with two edges $\{isa : X \to man, worksfor : X \to irimas\}$, where X denotes a variable as illustrated in Figure 1. By forgetting labels, the query is also represented by $\{(X, man), (X, irimas)\} : G(N \cup X)$. The result is then a set of sets of pairs $\{\{(X, laurent)\}, \{(X, zhao)\}\}$ and the program finding the possible morphisms can be formalized by $query : G(N') \times G(N) \to \mathcal{P}(\mathcal{P}(X \times N))$ if $N' = N \cup X$ the union of constant nodes N plus variables X; $\mathcal{P}(X \times N)$ is here a shortcut for a mapping function $f : X \to N$. Such a program has been already detailed in the literature with in particular [7] and [8].

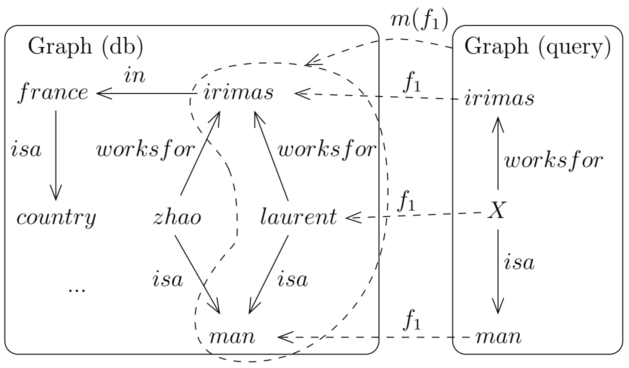

Figure 1. Sample graphs and query/morphism.

Now, other representations of graphs are possible, e.g. $G'(N) = \mathcal{P}(N \times \mathcal{P}(N))$ that associates adjacent nodes to each node. The relation between the two functors can be represented by a natural transformation $\eta : G'(N) \to G(N)$ with $\eta(g') = \{(x, y) \,|\, (x, ys) \in g, y \in ys\}$. This transformation is invertible and the functors/datatypes are then said to be naturally isomorphic $G'(N) \cong_\eta G(N)$. If the two structures represent a "same" information, the performance of a program depends on the structure selected. As an example, a function/program to get the adjacent nodes, i.e. $g(n) : G(N) \to \mathcal{P}(N)$, will have complexity $\mathcal{O}(n)$ where n is the number of edges when using G, and $\mathcal{O}(m)$ where m is the number of nodes when using G', and $m \leq n$. So, $g'(n) : G'(N) \to \mathcal{P}(N)$ is "faster" than $g(n)$.

The change from G to G' can be viewed as an optimization technique called "memorization" in the sense that G' memorizes the result (i.e. adjacent node) for each input node and then eliminates extra computations [9]. The optimized version of the program will be obtained with $g'(n) = g(n) \circ \eta^{-1}$ that can be simplified by using the definitions of g and η^{-1} (and is known as short-cut fusion optimization [10]). Another common optimization technique consists in splitting data and use parallel computations. In the example of graphs and by considering a pair of computers, this can be modeled with $G''(N) = G(N) \times G(N)$. The function to get the adjacent nodes will be now $g''(n)(g_1, g_2) = \cup \circ (g(n)(g_1) \times g(n)(g_2))$ with a complexity $\mathcal{O}(max(n_1, n_2))$ where n_i is the size of g_i. And $\mathcal{O}(g'') \leq \mathcal{O}(g') \leq \mathcal{O}(g)$.

III. Categories for BigData

The elements presented have been translated in the functional programming language `Haskell` [11]. Products are then interpreted as pairs `(x,y)` and are associated to the higher-order function `mult f g (x,y) = (f x,g y)`. Powersets are replaced by lists `[x]` inductively defined by the empty list `[]` and a binary operator `(:)` to add an element to a list. $\mathcal{P}(f)$ is then represented by the `map f` function.

The various representations of graphs (functors) are encoded as type synonyms: `[(x,x)]`, `[(x,[x])]`, etc. Natural transformations simply correspond to functions and the following code gives the example of the η (eta) transformation detailed previously. An expression such as `\x->y` denotes an anonymous function $f(x) = y$, the `concat`[2] function concatenates a list of lists, and the dot `(.)` represents function composition. The function `get` returns the adjacent nodes, and `eta'` is the inverse of `eta` (the extra parameter `xs` represent the nodes' list in the graph `g`).

```
type G  x = [(x,x)]
type G' x = [(x,[x])]

eta :: G' x -> G x
eta =
 concat.(map (\(x,ys)->map (\y->(x,y)) ys))

get :: x -> G x -> [x]
get n =
 concat.(map (\(x,y)->
  if (x==n) then [y] else []))

eta':: [x] -> G x -> G' x
eta' xs g = map (\x->(x,get x g)) xs

get':: x -> G' x -> [x]
get' n []          = []
get' n ((x,ys):xs) =
 if (x==n) then ys else get' n xs
```

The following program can now be used to get the adjacent nodes of a node and compare the performances for the various representations of data and programs (i.e. does

[1]https://neo4j.com/

[2]As another example of the cut-fusion optimization, by having the definitions of *concat* and *map*, one can check that the complexity of $concat \circ map$ is $\mathcal{O}(n^2)$. Now, the definitions can be used to simplify the expression and obtain the definition of another well-known function called $filter$ with a $\mathcal{O}(n)$ complexity.

get better than get' ?). In this program, the graph is stored in a text file "graph.txt" that is loaded with the `readFile` function, and parsed with the `read` function. The function get (resp. get') is then called and the result r displayed with the `print` function; the operator `(!!)` returns the n-th element of a list and is used here to get the node we are looking for adjacent from command line arguments (see `getArgs`). The code is then compiled by using the Glasgow Haskell Compiler (see `ghc` below), and execution time is measured with the shell command `time`.

```
-- Usage: ghc --make BigData.hs
-- time ./BigData Zulu
main = do
 xs <- getArgs
 f  <- readFile "graph.txt"
 let g = read f :: G String
 let r = get (xs!!0) g
 print r
```

The dataset[3] considered in `graph.txt` describes the relation between the concepts in Wikepedia pages, e.g. $(19th\_century, Telephone)$, and has 119767 edges (3.8Mo for file size). The computer(s) considered for performances' measurement is an EliteBook(s) 820 with processor(s) Intel i5 with 4 threads at 2.3GHz, and 16Go RAM plus 16 Go of Swap.

The time required by each steps of the preceding program are presented in Figure 2. Thus, the whole program takes 3.112s to get the concepts linked to "Zulu" (that is the last concept in the data list and the worst case to query), and most of the time is dedicated to load/parse the data file, i.e. 3.092s. The use of G' and get' will reduce the time by 0.020s = $\mathcal{O}(get) - \mathcal{O}(get')$. As a remark, if $\eta(g)$ is saved in "graph2.txt" then the size of the file is 1.7Mo (smaller than the original file) and read/parse will only required 1.268s. The change of the data structure (see `eta`) requires 0.088s.

Program	Time
`read`	0m3.092s
`get`	0m0.020s
`eta`	0m0.088s
`get'`	0m0.000s
$10^9\times$`get'`	0m15.480s

Figure 2. Performances measured.

To eliminate the file loading/parsing, the program can be transformed into a service (what is also the first step to parallel and distributed computations - see G'' in the preceding section) as follow. In the code, `slave` will load a data `file` and listen to a given `port` of localhost. It continuously waits for a node's to query, then calls the get function and sends the result. The program `query` simply opens a connection to a host `h` at a port `p` and transmit a node to query `q`. The `main` program is adapted to either start a slave or to send a query.

```
slave file port = withSocketsDo $ do
 f <- readFile file
 let g = read f :: G String
 sock <- listenOn $ PortNumber port
 forever $ do
  (handle, host, port) <- accept sock
  query <- hGetLine handle
  let r = get query g
  hPutStrLn handle (show r)

query h p q = withSocketsDo $ do
 handle <- connectTo h (PortNumber p)
 hPutStrLn handle q
 r <- hGetLine handle
 putStrLn r

main = do
 a <- getArgs
 case a of
  ["slave",f,p]   -> slave f (read p)
  ["query",h,p,q] -> query h (read port) q
```

The distributed version of the application is then used as follow. In case 1, a single slave with the whole database is considered. In case 2, two slaves are considered with half of the database. In case 3, the request is directly sent to Wikepedia.

```
# case 1
./bigdata slave graph.txt 9000 &
time (./bigdata query localhost 9000 Zulu)

#case 2
./bigdata slave graph-part1.txt 9000 &
./bigdata slave graph-part2.txt 9001 &
time (./bigdata query localhost 9000 Zulu &
      ./bigdata query localhost 9001 Zulu)

#case 3
time (curl https://en.wikipedia.org/wiki/Spe-
 cial:Search?search=Zulu)
```

The time required by each of the preceding cases is presented in Figure 3. As expected, the time required to load the file database is suppressed and the gain is about 3.092s (case 1). The split of the database (case 2) brings a gain of only 0.005s, and this may be explained by transmission time introduced when using a network. A comparison with a direct query to Wikipedia (case 3) shows that the application presented is 50 times faster than this later. Many reasons can explain this difference: the route to attain the Wikepedia server is longer, the server is queried by many simultaneous clients and this one returns a HTML page (36.32Ko transmitted at 867Kb/s) and not a simple lists of adjacent concepts (approx. 222o).

Finally, despite its apparent simplicity, the application proposed as an illustration well shows that theoretical concepts such as functors (G,G',G'',etc.) and natural transfor-

[3]https://snap.stanford.edu/data/wikispeedia.html

case 1	0m0.020s
case 2	0m0.015s
case 3	0m0.754s

Figure 3. Performances using network.

mations/isomorphisms (η) can be easily implemented in a functional programming language (e.g. `Haskell`), and conduce to efficient programs and optimizations of them. In particular, the attentive reader can see that the distributed version of G used in the definition of G'' can be again improved by using get' (faster than get as shown in the last line of Figure 2).

As mentioned before, the application can be easily extended to other data structures, functors and natural transformations. For instance, labeled graphs can be modeled by $G_l(N, L) = \mathcal{P}(N \times (L \times N))$ where L corresponds to labels, or $G'_l(N, L) = \mathcal{P}(N \times \mathcal{P}(L \times N))$, or $G''_l(N, L) = \mathcal{P}(L \times \mathcal{P}(N \times N))$, etc. This is currently under study and a discussion about the benefits of each representation and the performances when querying a specific information will be presented in the future. The application can also be easily generalized to other subjects such as the search of documents containing a keyword. Indeed, a set of documents having each one a set of words can be modeled by $\mathcal{P}(Doc \times \mathcal{P}(Word))$ that can be changed to $\mathcal{P}(Doc \times Word)$ with η, that can be transformed to $\mathcal{P}(Word \times Doc)$ with $\mathcal{P}(\backslash(x, y) \to (y, x))$, and finally to $\mathcal{P}(Doc \times \mathcal{P}(Word))$ with η^{-1}. The get function then serves to find a specific word to return a set of documents.

IV. Conclusion

This article describes on how category theory combined with a functional programming language can be interesting in a BigData context. It explains how the concepts of functors and natural transformations can serve to represent data structures and data transformations. It then develops the example of querying a specific information in various representations of a graph database based on a real life and large dataset (Wikipedia) to compare the performances of various programs[4] returning a same result. If the work presented is still in progress, the results already obtained are encouraging.

The perspectives considered now will consist in studying more complex models and their relations to the standards (relational databases, tree structures such as XML documents, document-oriented models such as JSON files or MongoDB databases, or again labeled graph as found in Neo4J). The perspectives will also study queries and the graph morphism problem (i.e. finding a map form a subgraph representing a query to another graph corresponding to a database, and get answers for a precise query) that is known as a complex problem but important in a semantic web context [8].

[4]Available at `https://github.com/thiry/bigdata`

References

[1] M. Chen, S. Mao, and Y. Liu, “Big data: A survey”, *Mob. Netw. Appl.*, vol. 19, no. 2, pp. 171–209, Apr. 2014.

[2] T. Erl, W. Khattak, and P. Buhler, *Big Data Fundamentals: Concepts, Drivers & Techniques*. Upper Saddle River, NJ, USA: Prentice Hall Press, 2016.

[3] J. A. Goguen, “A categorical manifesto”, in *Mathematical Structures in Computer Science*, 1991, pp. 49–67.

[4] M. Barr and C. Wells, Eds., *Category Theory for Computing Science, 2nd Ed.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1995.

[5] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O’Reilly Media Inc., 2013.

[6] P. T. Wood, “Query languages for graph databases”, *SIGMOD Rec.*, vol. 41, no. 1, pp. 50–60, 2012.

[7] L. Thiry, M. Mahfoudh, and M. Hassenforder, “A functional inference system for the web”, *International Journal of Web Applications*, vol. 6, no. 1, pp. 1–13, 2014.

[8] T. Segaran, C. Evans, J. Taylor, S. Toby, E. Colin, and T. Jamie, *Programming the Semantic Web*. O’Reilly Media, Inc., 2009.

[9] C. Okasaki, *Purely Functional Data Structures*. New York, NY, USA: Cambridge University Press, 1998.

[10] R. Bird and O. de Moor, *Algebra of Programming*, ser. Prentice-Hall international series in computer science. Prentice Hall, 1997.

[11] N. Shukla, *Haskell data analysis cookbook*. Birmingham: Packt Publ., 2014.