World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:13, No:2, 2019

Functional and Efficient Query Interpreters: Principle, Application and Performances' Comparison

Laurent Thiry, Michel Hassenforder

***Abstract*—This paper presents a general approach to implement efficient queries' interpreters in a functional programming language. Indeed, most of the standard tools actually available use an imperative and/or object-oriented language for the implementation (e.g. Java for Jena-Fuseki) but other paradigms are possible with, maybe, better performances. To proceed, the paper first explains how to model data structures and queries in a functional point of view. Then, it proposes a general methodology to get performances (i.e. number of computation steps to answer a query) then it explains how to integrate some optimization techniques (short-cut fusion and, more important, data transformations). It then compares the functional server proposed to a standard tool (Fuseki) demonstrating that the first one can be twice to ten times faster to answer queries.

***Keywords*—Data transformation, functional programming, information server, optimization.

I. Introduction

THE availability of a growing number of structured information requires dedicated tools to query, in an efficient manner, a specific information. If standards exist to describe information (e.g. knowledge models/graphs, ontologies, etc.) or queries (SPARQL and DL languages in particular), and if some tools are proposed to deal with these elements (e.g. triple stores such as Jena-Fuseki), there is no real formalism to describe both the behavior and performances of these tools. To solve this limitation, the article explains how to use functional paradigm to formalize both the syntax and the semantic of the languages used to specify information's structure and queries. From this, it extends this formalization to integrate performances' calculation and uses it to make optimizations. Finally, it presents an implementation of the concepts introduced to get a functional server and shows, by the way of a concrete example (Wikipedia pages' links), how this one can be 10 times faster than existing tools (e.g. Fuseki). More precisely, the main elements presented consist in: 1) a datatype language d usable to describe various data organizations (e.g. tables, maps, graphs, etc.) and a query language q, 2) a semantic function $e : d \times q \rightarrow r$ to evaluate queries, 3) a methodology to integrate performances in the model $oe : d \times q \rightarrow \mathbb{N}$, 4) a set of possible transformations $t : d \rightarrow d'$ that are optimizations - formally: $oe'(t(d), q) \leq oe(d, q)$. These elements are implemented into the functional programming language Haskell and can be compiled to get a concrete tool (executable) to start an information server or send it queries. Most of the code is presented in the article to: 1) show how functional paradigm leads to simple code (to be compared with existing tools such as Fuseki that has 25Mo of code, for instance), 2) be precise as much as possible - functional code being very close to mathematical model, and 3) help the reader to see the rigor used to get the results presented (e.g. performances calculation).

This document is divided in five sections. Section II gives an overview of the elements related to the work presented and helps to understand its interest. Section III introduces the fundamental elements required to understand the elements proposed. More precisely, it introduces some important concepts from functional programming, and how it helps to specify simply datatypes and data transformations (required for optimizations). Then, it proposes an approach to measure performances of a functional program (to count computation steps). Section IV then proposes an interpreter for a subset of the attributive language $\mathcal{AL}$. It also studies its performances and detailed possible improvements - what illustrates how data transformations can lead to optimized programs. Section V uses a concrete example (Wikipedia links) to study the performances of the server proposes and compare them with a standard triple store (Fuseki). Section VI summarizes the main elements presented and gives an overview of the perspectives considered.

II. Related Works

It is a well-known fact that the data structures used by a program has a direct impact on its performances, i.e. memory usage or time consumption, as explained in [1]. For instance, the search of an element in a list as a complexity $\mathcal{O}(n)$, where n is the number of elements, while the search in a (balanced) binary tree as a $\mathcal{O}(log(n))$ complexity. Next, if imperative and object-oriented programming languages are the most used languages in an industrial context, functional languages keep having many interest as explained in [2] (e.g. shorter code by using generic and higher-order functions such as map, $filter$, etc. - these ones are used again in this paper). As a complement, a study of data structures and functional programs' performances can be found in [3], and [4] shows how a modern functional programming language such as Haskell (the language considered in this paper) can also leads to industrial applications.

In the particular context of Semantic Web, whose a general presentation can be found in [5], tools proposed are mainly

M. Hassenforder is with the IRIMAS Lab., 12, rue des frères Lumière, 68093 Mulhouse, France.
L. Thiry is with the IRIMAS Lab., 12, rue des frères Lumière, 68093 Mulhouse, France (e-mail: laurent.thiry@uha.fr).

presented by the way of their usage or architecture (e.g. classes) but not really by the way of their behavior and performances. The most complete presentations can be found in [6]-[8] or [9]. Our previous works have tried to solve this lacks with in particular [10] that presents, in a detailed manner, a knowledge management platform integrating a triple store, a query interpreter, an inference system and a web browser to navigate within or manage a knowledge model. A more recent work is given by [11] that studies the performances of the preceding system and studies possible optimizations based on data transformations. This paper then synthesizes the results obtained with an extension to a subset of Description Logic [12].

III. Background

This part gives an illustrated presentation of fundamental concepts used in functional programming (and the paper) and detailed the process proposed to get the performances of the programs.

A. Functional Descriptions

1) Illustrated Presentation: Functional paradigm uses functions such as $s : N \rightarrow N$, and constants - e.g. $z : N$, to describe ”things”. In particular, the preceding examples define a set $N = \{z, s(z), ..., s^n(z), ...\}$ corresponding here to the Peano representation of natural numbers $\mathbb{N}$. Operators on this set are then defined by using functions and rewriting rules such as: $e_1 : p(z, m) = z$ and $e_2 : p(s(n), m) = s(p(n, m))$. These rules correspond to the addition, and one can check that $p(s^2(z), s^1(z)) = s^3(z)$ by applying e_2 twice then e_1. Now, the expressions defining N can formalized by the way of a grammar:

```
<N> := z | s(<N>)
```

Or by using a datatype definition in any (functional) programming language. The code below then gives an implementation of (N, p) in the Haskell programming language.

```
data N = Z | S(N)

p(Z,m)    = m
p(S(n),m) = S(p(n,m))
```

2) Data Collections: As another example, lists of N can be specified by a constant for the empty list $e : L_N$, and a function to add a value to a list $a : N \times L_N \rightarrow L_N$. Thus, an expression such as $a(n_1, a(n_2, e))$ will be interpreted as a list $[n_1, n_2]$. The catenation operator is then defined in a similar manner of the plus operator on numbers with: $p_1 : pl(e, l') = l'$ and $p_2 : pl(a(x, l), l') = a(x, pl(l, l'))$. One can then check for instance that $pl([1, 2], [3]) = [1, 2, 3]$ by applying p_2 twice then p_1.

Lists can be generalized by using a parameterized datatype $L(x)$ where x is a type variable replacing N in the previous definition (and thus $L_N = L(N)$). Some generic functions, used in the rest of the document, can then be defined to: concatenate a list of lists (cat), apply a function to all the elements of a list ($map(f)$), select the elements satisfying a predicate ($filter(p)$), etc. The implementation of these functions in Haskell are given below.

```
data L x = E | A (x,L x)

pl(E     ,l') = l'
pl(A(x,l),l') = A(x,r)
 where r = pl(l,l')

cat(E)      = E
cat(A(x,l)) = pl(x,r)
 where r = cat(l)

map(f,E)      = E
map(f,A(x,l)) = A(f(x),r)
 where r = map(f,l)

filter(p,l) = r'
 where f(x) = if (p(x)) then A(x,E) else E
       r  = map(f,l)
       r' = cat(r)
```

B. Performance Measurement & Optimizations

1) Principle: The performance of a functional program depends on the number of computation steps to get a result. To get this value, we propose to add an extra result to the functions as illustrated below, e.g. $pl : L(x) \times L(x) \rightarrow L(x)$ is transformed into $opl : L(x) \times L(x) \rightarrow L(x) \times \mathbb{N}$. All the functions are prefixed by ”o” (to show that they embedded the complexity $\mathcal{O}$) and are constructed in a systematic manner on: taking the original functions, replacing results r_i by (r_i, n_i), and summing the various performances $\Sigma_i n_i$ when functions are composed.

```
opl(E     ,l') = (l',0)
opl(A(x,l),l') = (A(x,r),n+1)
 where (r,n) = opl(l,l')

ocat(E)      = (E,0)
ocat(A(x,l)) = (r',n+n')
 where (r,n)   = ocat(l)
       (r',n') = opl(x,r)

omap(f,E)      = (E,0)
omap(f,A(x,l)) = (A(r',r),n+n')
 where (r,n)   = omap(f,l)
       (r',n') = f(x)

ofilter(p,l) = cat(map(f,l))
 where f(x) = if (r) then (A(x,E),n+1) else (E,n+1)
         where (r,n) = p(x)
       (r,n) = omap(f,l)
```

Thus, performance measurement is now possible. For instance, $opl([1, 2], [3]) = ([1, 2, 3], 2)$ - more generally $opl(l, l') = (pl(l, l'), n)$ where n is the length of l (what is also written ”pl is $\mathcal{O}(n)$”). Or again, $ofilter(odd, [1..10]) = ([1, 3, 5, 7, 9], 25)$.

2) Sample Optimization: Optimizations consist in transforming code to reduce the number of computation steps. For instance, the code below proposes another realization of the $filter$ function, and $ofilter'(odd, [1..10]) = ([1, 3, 5, 7, 9], 10)$ what requires half of the computations ! The change realized is based on a well-known ”short-cut fusion” [13] and consists simply in eliminating intermediate functions (here, cat and map) by using their definition. Other examples of this principle will be detailed in the rest of the document.

```
ofilter'(p,E)      = (E,0)
ofilter'(p,A(x,l)) = if (r) then (A(x,r'),n+n')
                     else (r',n+n')
 where (r,n)   = p(x)
       (r',n') = ofilter'(p,l)
```

IV. Information Server

This part proposes: 1) an interpreter for a query language using a particular data structure (table), 2) a quick data transformation that leads to 3) an optimized interpreter using another data structure (map).

A. Presentation

An information server is mainly defined by a set of data d, queries on this one q, and an evaluator e for these queries, i.e. $e : d \times q \to r$ where r represent the set of results.

1) Syntax & Semantic: Datatypes d can be simple values d_0, records $d \times d$, or lists $L(d)$. Some examples are $d_1 = L(x \times t)$ and $d_2 = L(t \times L(x))$, and the code below gives sample values for these types. The first value v_1 can be interpreted as the table represented in Fig. 1, and $v2$ as another representation of the same information similar to a map $t \to L(x)$ - this point is detailed later in the document.

```
v1 = [("x1","T1"),("x2","T1"),("x3","T1")
     ,("x2","T2"),("x4","T2")]
v2 = [("T1",["x1","x2","x3"]),("T2",["x2","x4"])]
```

Value	Tag
x_1	T_1
...	...
x_4	T_2

Fig. 1 Sample data (v_1)

By considering the example illustrated in Fig. 1, sample queries q can consist in finding the values having a particular tag q_0, values having two tags $q \wedge q'$, or values not having a specific tag $\neg q$ - what corresponds to a subset of class expressions found in Description Logic (DL).

A semantic function for this language is then simply defined as follow:

```
e1(v1,q0)     =map(\(x,y)->x,filter(\(x,y)->y==q0,d1))
e1(v1,q1/\q2)=inter(e1(v1,q1),e1(v1,q2))
...
```

In the code, an expression such as $\lambda x \to e$ corresponds to a anonymous function $f(x) = e$, and the function $inter$ returns the common elements of two lists (similar to set's intersections).

2) Performances: The principle introduced in Section III-B1 is now applied to get the performance of the implementation (see code below). The function $elem$ that tests the presence of an element into a list is introduced to define the function $inter$section.

```
oelem(x,[])  = (False,1)
oelem(x,y:z) = let (r,n)=oelem(x,z) in
 if (x==y) then (True,1) else (r,n+1)

ointer(x,y) = ofilter(\z->oelem(z,y),x)
```

```
oe1(v1,q0)=let (r,n)   = ofilter(\(x,y)->(y==q0,1),v1)
              (r',n')= omap(\(x,y)->(x,1),r)
 in (r',n+n')
oe1(v1,q1/\q2) = let (r,n)    = oe1(v1,q1)
                     (r',n')  = oe1(v1,q2)
                     (r'',n'')= ointer(r,r')
 in (r'',n+n'+n'')
```

As an illustration, $oe_1(v_1, T_1 \wedge T_2) = ([x_2], 27)$.

B. Optimization

1) New Data Structure & Semantic: A well-known optimization technique is "caching" that consists in memorizing the result of a query. With the elements presented, this technique can be viewed as a datatype's transformation to:
$v_2 = [(T_1, [x_1, x_2, x_3]), (T_2, [x_2, x_4])]$.

This change implies a new semantic function that can be defined as follow:

```
e2(v2,q0)     =
 head(map(\(x,y)->y,filter(\(x,y)->x==q0,v2)))
e2(v2,q1/\q2)=inter(e2(v2,q1),e2(v2,q2))
...
```

2) New Performances: The performance is obtained with the same principles already used, and the code:

```
oe2(v2,q0)=let (r,n)  =ofilter(\(x,y)->(x==q0,1),v2)
              (r',n')=omap(\(x,y)->(y,1),r)
 in (head(r'),n+n'+1)
oe2(v2,q1/\q2) = let (r,n)    = oe2(v2,q1)
                     (r',n')  = oe2(v2,q2)
                     (r'',n'')= ointer(r,r')
 in (r'',n+n'+n'')
```

As an illustration, $oe_2(v_2, T_1 \wedge T_2) = ([x_2], 13)$ with a difference of 12 computations (twice faster) from the original implementation. The reasons can be: 1) the use of the $head$ function returning the first element of a list that eliminates extra computations (on the tail of the list), and 2) the length of the list analyzed (i.e. number of tags rather than (value,tag) pairs).

3) (Optimized) Data Transformation: The transformation $t_{21} : d_2 \to d_1$ can be defined by:

```
t21(v2) = concat(map(\(x,y)->map(\z->(z,x),y),v2))

ot21(v2) = let off(x,y) = omap(\z->((z,x),1),y)
              (r,n)    = omap(off,v2)
              (r',n')  = ocat(r)
 in (r',n+n')
```

An interesting point here is the possibility to use the "short-cut fusion" principle mentioned before to define a equivalent function:

```
t21'([])           = []
t21'((x,[]):xs)    = t21'(xs)
t21'((x,(y:ys)):xs) = (y,x):t21'((x,ys):xs)

ot21'([])           = ([],1)
ot21'((x,[]):xs)     = let (r,n)=ot21'(xs) in (r,n+1)
ot21'((x,(y:ys)):xs) = let (r,n)=ot21'((x,ys):xs)
 in ((y,x):r,n+1)
```

A comparison of the performances gives: $ot_{21}(v_2) = (v_1, 16)$ and $ot'_{21}(v_2) = (v_1, 8)$ what shows that the new version of the transformation is twice faster than the first one.

V. Real-World Application & Comparison

As a sample application, we have used a dataset representing the Wikipedia pages' links of the web site, and available at: `snap.stanford.edu/data/wikispeedia.html`. The data is store in a tsv file (with size 3Mo) and consists in 119882 links between 4600 pages.

A. IO & Servers

In the application, the data is first parsed, transformed into a list of records (d_1), then serialized in a file V1.txt - see the $createV1$ function given in Appendix VI-A. Next, a command line utility as been defined (see the $main$ function in the appendix) to load the file, read a query, and print the result. The source is then compiled by using the Glasgow Haskell Compiler (ghc) in a command called "query". Two sample queries have been considered with: a simple one, $q_1 = Apollo\_11$, to get the pages pointing to "Apollo_11", and a complex one to get the pages pointing both to two given pages, $q_2 = Moon \wedge Florida$.

The tool is then used to get the answer of these queries - this, with the Linux "time" command to get the duration of the evaluation function, as follow:

```
> time(./query "Q0 \"Apollo_11\"")
([...,"Wernher_von_Braun"],119901)
real 0m1,755s

> time(./query "And (Q0 \"Moon\") (Q0 \"Florida\")")
(["Apollo_11",...,"George_W._Bush"],260214)
real 0m1,800s
```

Next, the command line utility has been transformed into a service by using the code given in Appendix VI-B. Thus, the data is now loaded in memory what eliminates the file reading from the preceding examples. The code below then presents the usage of the new utility: the first line starts the server on a port, and the second one send query to the server.

```
./query server1 9000 &

time(./query query localhost 9000 "Q0 \"Apollo_11\"")
real 0m0,060s

time(./query query localhost 9000
               "And (Q0 \"Moon\") (Q0 \"Florida\")")
real 0m0,111s
```

The code has then be adapted to use the optimized interpreter (oe_2) and has given the following performances:

```
# q1
real 0m0,928s (in file storage)
real 0m0,007s (in-memory/server)

# q2
real 0m0,903s (in file storage)
real 0m0,015s (in-memory/server)
```

B. Triples & Fuseki

Finally, the dataset has been: 1) transformed into a list of triples (N-Triples notation), and 2) loaded into the Fuseki server. The command line utilities proposed with the server distribution have then been used as follow to get performances:

```
time (./s-query --service="localhost:3030/wiki/query"
  "SELECT ?x WHERE { ?x <o:linkto> <o:Apollo_11> . }")
real 0m0,100s

time (./s-query --service="localhost:3030/wiki/query"
  "SELECT ?x WHERE { ?x <o:linkto> <o:Moon> .
                     ?x <o:linkto> <o:Florida> . }")
real 0m0,107s
```

C. Synthesis

Fig. 2 summarizes the various performances obtained for the standard triples's store Fuseki, and the server proposed in its initial version (e_1) and its optimized one (e_2) - this for a simple query (q_1) and a more complex one (q_2).

Server	q_1	q_2
e_1	0,060	0,111
e_2	0,007	0,015
$Fuseki$	0,100	0,107

Fig. 2 Global performances (s)

Thus, the original/functional version of the server and Fuseki have similar performances for complex queries, while the optimized version is 10x faster.

VI. Conclusion

This paper has illustrated how functional concepts can be used to organize or transform information, and query a specific element in an efficient manner. Queries are expressed with a subset of DL language and two semantic functions are proposed depending on how data are structured. Then, performances measurement and comparison show that a simple interpreter for the language can be dramatically improved (twice to ten time faster) by using data transformations. Most of the concepts are implemented in the functional programming language Haskell, and the total code of the resulting server is approximatively 5.3Ko - what can be compared to more standard tools such as Fuseki, for instance, that has 25Mo. Sure, this later offers more functionalities but can be more difficult to manage and optimize. As another comparison, the paper has shown that this tool is globally ten times slower than the server proposed (Fig. 2).

The main perspectives considered now will consist in 1) looking for other possible optimization (e.g. splitted data and concurrent computations), and 2) extending the query language with other constructs. Indeed, if the language is actually more general than the one presented (e.g. expressions such as $\exists hasTag\,\{T_i\}$ are possible but have not been detailed for clarity reasons), it is not a full DL language.

Appendix

A. Command Line Utility

```
main = do
 [q] <- getArgs
 f   <- readFile "v1.txt"
 let v1 = read f :: [(String,String)]
 let q_ = read q :: Q
 let r1 = oe1(v1,q_{})
 print r1

createV1 = do
```

```
 f <- readFile "Wikipedia.tsv"
 let r = splitOn "\n" f
 let r2= map (splitOn "\t") r
 let r3= map (\[x,y]->(x,y)) r2
 writeFile "d1.txt" (show r3)
```

B. Information Server

```
main = do
 xs <- getArgs
 case xs of
  ["server1",p]     -> service1 (read p)
  ["server2",p]     -> service2 (read p)
  ["query" ,h,p,q] -> transmit h (read p) q

service1 :: PortNumber -> IO ()
service1 port = withSocketsDo $ do
 f <- readFile "d1.txt"
 let d1 = read f :: [(String,String)]
 sock <- listenOn $ PortNumber port
 servicebody1 sock d1

servicebody1 sock d1 =
 forever $ do
  (handle, host, port) <- accept sock
  t <- hIsEOF handle
  if t then return ()
   else
    do
     q2' <- hGetLine handle
     let q2 = read q2' :: Q
     let r1 = oe1(d1,q2)
     hPutStrLn handle (show r1)
     hFlush handle
     hClose handle
```

References

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[2] J. Hughes, "Research topics in functional programming," D. A. Turner, Ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990, ch. Why Functional Programming Matters, pp. 17–42.

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

[4] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008.

[5] P. Hitzler, M. Krtzsch, and S. Rudolph, *Foundations of Semantic Web Technologies*, 1st ed. Chapman & Hall/CRC, 2009.

[6] O. Cur and G. Blin, *RDF Database Systems: Triples Storage and SPARQL Query Processing*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[7] M. Martin, J. Unbehauen, and S. Auer, "Improving the Performance of Semantic Web Applications with SPARQL Query Caching," in *Proceedings of 7th Extended Semantic Web Conference (ESWC 2010), 30 May – 3 June 2010, Heraklion, Crete, Greece*, ser. Lecture Notes in Computer Science, L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Eds., vol. 6089. Berlin / Heidelberg: Springer, 2010, pp. 304–318.

[8] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. Van de Walle, "Querying datasets on the Web with high availability," in *Proceedings of the 13th International Semantic Web Conference*, ser. Lecture Notes in Computer Science, P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandei, P. Groth, N. Noy, K. Janowicz, and C. Goble, Eds., vol. 8796. Springer, 2014, pp. 180–196.

[9] A. Hogan, A. Harth, J. Umrich, S. Kinsella, A. Polleres, and S. Decker, "Searching and browsing linked data with swse: the semantic web search engine," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 9, no. 4, 2011.

[10] L. Thiry, M. Mahfoudh, and M. Hassenforder, "A functional inference system for the web," *IJWA*, vol. 6, no. 1, pp. 1–13, 2014.

[11] L. Thiry, H. Zhao, and M. Hassenforder, "Categories for (big) data models and optimization," *Journal of Big Data*, vol. 5, no. 1, p. 21, Jul 2018.

[12] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications*, 2nd ed. New York, NY, USA: Cambridge University Press, 2010.

[13] A. Takano and E. Meijer, "Shortcut deforestation in calculational form," in *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '95. New York, NY, USA: ACM, 1995, pp. 306–313.

Laurent Thiry is Professor of Computer Science at University of Mulhouse (France). His main research interests are Software and Model-Driven Engineering, Formal Methods and Functional Programming for complex software, He has published several research articles in peer-reviewed international journals and conferences, and has served several conferences as a program chair, on these topics. The elements proposed result mainly from its participation to various international, european or national projects. Dr Laurent Thiry is the corresponding author and can be contacted at: laurent.thiry@uha.fr.

Michel Hassenforder is Full Professor of Computer Science at University of Mulhouse (France). His research interests are Software Engineering, Information Systems and Programming Languages. He has published several research articles in peer-reviewed international journals and conferences, and has participated to many international, european or national projects, on these topics. Michel Hassenforder can be contacted at: michel.hassenforder@uha.fr.