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 \( oe : d \times q \rightarrow N \), 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 \( AE \). 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 \( O(n) \), where \( n \) is the number of elements, while the search in a (balanced) binary tree as a \( O(\log(n)) \) 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 \( \text{map}, \text{filter}, \text{etc.} \) - 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
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 \( N = \{ z, s(z), \ldots, s^n(z), \ldots \} \) corresponding here to the Peano representation of natural numbers \( N \). 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 \( p(s^2(z), s^4(z)) = s^3(z) \) by applying \( e_2 \) twice then \( e_1 \). Now, the expressions defining \( N \) can formalized by the way of a grammar:

\[
\text{data N = Z | S(N)}
\]

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.

\[
\text{data L x = E | A (x, L x)}
\]

\[
\begin{align*}
\text{pl(E ,l')} &= l' \\
\text{pl(A(x,l),l') &= A(x,r)} \\
\text{where } r &= \text{pl(l,l')}
\end{align*}
\]

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.

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 \( opl : L(x) \times L(x) \rightarrow L(x) \times N \). All the functions are prefixed by \( "o" \) (to show that they embedded the complexity \( O \)) 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 \( \sum_i n_i \) when functions are composed.

\[
\begin{align*}
\text{opl(E ,l') &= (l',0)} \\
\text{opl(A(x,l),l') &= (A(x,r),n+1)} \\
\text{where } (r,n) &= \text{opl(l,l')} \\
\text{o cat(E) &= (E,0)} \\
\text{o cat(A(x,l)) &= (x',n+n')} \\
\text{where } (r,n) &= \text{o cat(l)} \\
\text{(r',n') &= o pl(x,r)} \\
\text{o map(f,E) &= (E,0)} \\
\text{o map(f,A(x,l)) &= A(r',n+n')} \\
\text{where } (r,n) &= \text{o map(f,l)} \\
\text{(r',n') &= f(x)} \\
\text{o filter(p,l) &= \text{cat(map(f,l))}} \\
\text{where } f(x) &= \text{if } (p(x)) \text{ then } A(x,E) \text{ else } E \\
\text{where } (r,n) &= \text{p(x)} \\
\text{(r,n) &= \text{o map(f,l) }}
\end{align*}
\]

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 \( O(n) \)"). Or again, \( o filter(odd,[1..10]) = ([1,3,5,7,9],25) \).

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 \( o filter(odd,[1..10]) = ([1,3,5,7,9],10) \) 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.
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 x q -> r where r represent the set of results.

1) Syntax & Semantic: Datatypes d can be simple values d0, records d x d, or lists L(d). Some examples are d1 = L(x x t) and d2 = L(t x L(x)), and the code below gives sample values for these types. The first value v1 can be interpreted as the table represented in Fig. 1, and v2 as another representation of the same information similar to a map.

By considering the example illustrated in Fig. 1, sample queries q can consist in finding the values having a particular tag q0, values having two tags q & q′, or values not having a specific tag ∼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:

\begin{align*}
\text{e1}(v1,q0) &= \text{map}(\{(x,y)\rightarrow x, \text{filter}(\{(x,y)\rightarrow y=q0,d1}\} \\
\text{e1}(v1,q1\land q2) &= \text{inter} (\text{e1}(v1,q1),\text{e1}(v1,q2))
\end{align*}

... 

In the code, an expression such as \( λx.e \) corresponds to a anonymous function \( f(x) = e \), and the function \text{inter} returns the common elements of two lists (similar to set’s intersection).

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

\begin{align*}
\text{oelem}(x,\{\}) &= (\text{False},1) \\
\text{oelem}(x,y\mid z) &= \text{let } (r,n) = \text{oelem}(x,z) \text{ in } \\
\text{if } (x=y) \text{ then } (\text{True},1) \text{ else } (r,n+1) \\
\text{ointer}(x,y) &= \text{ofilter}(\neg x \rightarrow \text{oelem}(z,y),x)
\end{align*}

As an illustration, \oe1(v1,T1 \land T2) = ([x2], 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

\( v2 = [(T1,[x1,x2,x3]),(T2,[x2,x4])] \)

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

\begin{align*}
\text{e2}(v2,q0) &= \text{head}(\text{map}(\{(x,y)\rightarrow y, \text{filter}(\{(x,y)\rightarrow y=q0,v2}\}) \\
\text{e2}(v2,q1\land q2) &= \text{inter} (\text{e2}(v2,q1),\text{e2}(v2,q2))
\end{align*}

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

\begin{align*}
\text{oe2}(v2,q0) &= \text{let } (r,n) = \text{ofilter}(\{(x,y)\rightarrow x=q0,d1\},v2) \\
\text{oe2}(v2,q1\land q2) &= \text{inter} (\text{oe2}(v2,q1),\text{oe2}(v2,q2))
\end{align*}

As an illustration, \oe2(v2,T1 \land T2) = ([x2], 13) with a difference of 12 computations (twice faster) from the original implementation. The reasons can be: 1) the use of the \text{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 \text{t21}: d2 \rightarrow d3 can be defined by:

\begin{align*}
\text{t21}(v2) &= \text{concat}(\text{map}(\{(x,y)\rightarrow y, \text{filter}(\{(x,y)\rightarrow x=q0,d1}\},v2)) \\
\text{ot21}(v2) &= \text{let } \text{off}(x,y) = \text{omap}(\{(z\rightarrow ((z,x),1),y\},v2) \\
\text{ot21}(v2) &= \text{let } \text{ocat}(r) = \text{omap}(\{(z\rightarrow (r,z),1),y\},v2)
\end{align*}

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

\begin{align*}
\text{t21}'(\{\}) &= \{\} \\
\text{t21}'(\{x,\{\}\}):xs &= \text{t21}'(\{x\}):xs \\
\text{t21}'((x,y):xs) &= ((y,x):\text{t21}'((x,y):xs) \\
\text{ot21}'(\{\}) &= \{\{\},1\} \\
\text{ot21}'(\{x,\{\}\}):xs &= \text{let } (r,n) = \text{ot21}'(\{x\}) \text{ in } (r,n+1) \\
\text{ot21}'((x,y):xs) &= \text{let } (r,n) = \text{ot21}'((x,y):xs) \text{ in } ((y,x):r,n+1)
\end{align*}

A comparison of the performances gives: \text{ot21}(v2) = v1(16) and \text{ot21}(v2) = v1(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, \(q_1 = \text{Apollo}_11\), to get the pages pointing to “Apollo_11”, and a complex one to get the pages pointing both to two given pages, \(q_2 = \text{Moon} \land \text{Florida}\).

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:

\[
\begin{align*}
&\text{time}(./s-query --service="localhost:3030/wiki/query" \\
&\text{"SELECT ?x WHERE { ?x <o:linkto> <o:Apollo_11> . }"})
\end{align*}
\]

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 \(\exists \text{hasTag} \{T_i\}\) are possible but have not been detailed for clarity reasons), it is not a full DL language.

APPENDIX

A. Command Line Utility

\[
\begin{align*}
\text{main} & = \text{do} \\
\{q\} & \leftarrow \text{getArgs} \\
\text{f} & \leftarrow \text{readFile} \ "v1.txt" \\
\text{let vl} & = \text{read} \ f :: [(\text{String},\text{String})] \\
\text{let q_} & = \text{read} \ q :: Q \\
\text{let rl} & = \text{oe1}(\text{vl},q_) \\
\text{print} \ rl \\
\text{createV1} & = \text{do}
\end{align*}
\]

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
        sock <- d1
        forever $ do
            (handle, host, port) <- accept sock
            if t then return ()
            else do
                q2 <- hGetLine handle
                let q2 = read q2 :: Q
                let r1 = oe1(d1, q2)
                hPutStrLn handle (show r1)
                hClose handle
    service2 :: PortNumber -> IO ()
    service2 port = withSocketsDo $ do
        f <- readFile "Wikipedia.tsv"
        let r = splitOn "\n" f
        let r2 = map (splitOn "\t") r
        let r3 = map (\[x,y\] -> (x, y)) r
        let q2 = read q2' :: Q
        q2' <- hGetLine handle
        let q2 = read q2' :: Q
        let r1 = oe1(d1, q2)
        hPutStrLn handle (show r1)
        hClose handle

REFERENCES


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.