A Functional Inference System for the Web

Laurent Thiry, Mariem Mahfoudh, Michel Hassenforder
ENSISA
12, rue des frères Lumière
68093, Mulhouse, France
{laurent.thiry, mariem.mahfoudh, michel.hassenforder}@uha.fr

ABSTRACT: *This paper explains how functional programming (FP) helps to integrate both theoretical and technological models by using generic datatypes and higher-order functions. As an application, the paper considers the domains of knowledge modeling and web technologies. More precisely, it explains how FP can be used to infer and to generate web pages from a graph based datamodel, or to embed a query language (helpful to search for a precise information). It then shows how the concepts lead to a generic platform dedicated to knowledge sharing and searching with an application to the CCAlps European project.*

Keywords: Functional Programming, Knowledge Management, Web Applications

Received: 11 November 2013, Revised 18 December 2013, Accepted 23 December 2013

1. Introduction

Software development is generally a complex task due to the variety of the models and technologies to be integrated. For instance, a web application needs a data model of the domain (with relational or graph models), a database to store it, web pages for a user friendly representation of the information (based on xHTML), a search engine to facilitate its use, etc.

In this context, the aim of the paper is to show how functional programming (FP) languages can help to formalize then compose all the preceding elements. To proceed, both the parts of an application are specified by the way of generic datatypes and higher-order functions to transform or to compose these latter. To better understand the benefits (and limitations) of the approach, this article also details the design of a generic web site built upon an inference system to facilitate the presentation, the add or query of a particular information.

FP focuses on simplicity and genericity [12]: functions are first class values, they can be passed as parameters of other functions, parameters can be omitted (with combinators), functions can be obtained from proofs (with the Curry- Howard isomorphism), etc. In addition to its mathematical foundation, FP can be used to manage databases, to build web servers, to design user interfaces [17], etc. The remaining question is then: how can FP be used to embed both theoretical/logical models (used for instance in knowledge modeling) and more standard technologies.

The article is divided into three parts. The first part gives an overview of knowledge modeling (and logic) and presents

functional programming with some typical applications. The second part proposes a functional model for a knowledge management platform, and its application to the CCAlps European project (alpine space program, project number 15-3-1-IT)[1]. Finally, the third part summarizes the elements presented and concludes by giving the considered perspectives.

2. Logic & Functional programming

2.1 Logic, languages & knowledge modeling

This section gives an informal presentation of the concepts commonly found in Logic and that will be used in the rest of the paper. Thus, "*knowledge*" is generally formalized by using a relational structure extended by a logical language for expressing properties to be satisfied.

A relational *structure* S is defined by: a set of symbols S, a set of relations R (also called a "*signature*"), an arity function *ar*: $R \rightarrow \mathbb{N}^+$, and an interpretation I with $\forall r \in R, I(r) \subseteq S^{ar(r)}$. If $ar(r) = 1$ then r is simply called a predicate (what defines a subset of S), and if $ar(r) = 2$ then r is a binary relation. Binary relations are interesting in the sense that information can be visually represented by a labeled graph (e.g. Figure 1). For instance, "*laurent* (l) *is* (i) *a person* (p) *that works* (w) *for ensisa* (e)" can be formalized by a structure S_1 defined by: $S = \{l, p, e, c\}$, $R = \{i, w\}$, $ar = \{i \mapsto 2\ , w \mapsto 2\}$, and $I = \{i \mapsto (l, p)\}, w \mapsto \{(l, e)\}\}$.

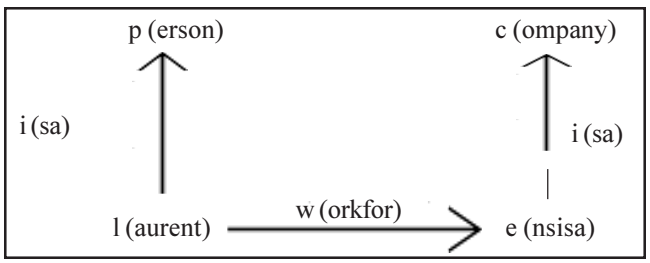

Figure 1. Example of structure S_1

Upon a structure, a *language* L, i.e. a set of terms T, can be inductively defined as follows. By considering a set of variables X $r \in R$ and $(s_i)_{i \in [1,n]} \in S \cup X$ then $r(s_1,..., s_n) \in T$. For all terms (t, t') and all variables x, $t \wedge t'$, $t \Rightarrow t'$ and $\forall x.\ t$ are also terms. Examples of terms are: $t_1 = i(l, p) \wedge w(l, e)$, and $t_2 = \forall x.\ \forall y.\ w(x, y) \Rightarrow c(y)$.

Next, a *satisfaction* relation written $S \models t$ where $t \in T$ can be defined as follows:

- $S \models r(s_1,...., s_n)$ if $(s_1,...., s_n) \in I(r)$,
- $S \models t \wedge t'$ if $S \models t$ and $S \models t'$,
- $S \models t \Rightarrow t'$ if $S \models t$ implies that $S \models t'$,
- $S \models \forall x, t$ if $S \models t$ when substituting x by v in t, forall $v \in S$.

With this definition, $S_1 \models t_1$ but t_2 is not satisfied because $(l, e) \in I(w)$ and $(e, c) \notin I(i)$, see dotted line in Figure 1.

In other words, to satisfy t_2 we infer that (e, c) must belong to $I(i)$. If $S \models t$ then S is called a *model* of t.

Many remarks can be made from the preceding definitions. Firstly, the definition of L is based on an inductive set T but languages are commonly presented by using production rules (also known as the grammar of the language [15]).

For instance, grammatical rules that summarize the possible constructs of L is [2]:

<V> := S | X
<T> := R (<V> [, <V>] ⋆) | <T> ∧ <T> | <T> => <T> | !X. <T>

Now, other grammars can be considered with for instance the one used by the Prolog system [5] or XML based representations

[1] www.ccalps.eu

(as found in the domain of ontologies [1, 6]). In Prolog, $\wedge$ is represented by a comma, and $t \Rightarrow t'$ by $t' :- t$. Thus, its is important to separate the concepts from their representations, and functional programming languages are interesting because they make this distinction: all the production rules from the grammar are expressed by using (constructor) functions that omit the "*concrete*" symbols [9]. As an illustration, the language L and the set/datatype T will be formalized by the following functions:

fact : String* –> T
and : $T \times T$ –> T
imply : $T \times T$ –> T
all : String $\times$ T –> T

t = all ("x", all ("y", imply (fact (["r", "x", "y"]), fact (["r", "y", "x"]))))

Thus, $\forall x. \forall y.\ r\ (x, y) \Rightarrow r\ (y, x)$ will be abstracted by t in the code above. It is not "*user friendly*" but: 1) FP proposes dedicated functions to pass from user friendly representation to this one [13], and 2) as a counter part, all the operations on T (e.g. the satisfaction function) can be easily defined with higher-order functions [16].

Additionally, a particular model/structure S can be fully defined by a subset of terms (t_i). Indeed, the satisfaction relation can be used to infer elements to be added in S, i.e. find S' such as $S' \Rightarrow t_1 \wedge ... \wedge t_n$. The set ($t_i$) can be split into "*facts*", e.g. $w\ (l, e)$ that have no variable, and "*rules*", e.g. $\forall x. \forall y. w\ (x, y) \Rightarrow i\ (x, p)$. In the case of binary relations, facts such as $r\ (a, b)$ can be interpreted as edges of a labeled graph $a \xrightarrow{r} b$. Thus, a set of facts is viewed as list/set of edges (e.g. Figure 1) and an interesting thing about FP is its capability in dealing with lists (then consequently with sets, relations and graphs). In particular, FP languages such as Haskell or ML generally propose higher-order functions (e.g. map or filter) to easily transform, or extract information from, a list.

As an illustration, the following code gives a possible implementation for the graph/model m of the Figure 1 in Haskell [17]. The function $f0$ selects from a model m the edges whose source is v, and f extracts from the result the label and the target of these edges. The preceding result is generally called the frame of v [14]. As a complement, FP is well suited to manipulate texts (i.e. lists of characters). For instance, the function view returns the code to display a frame into a browser (Figure 2).

```
m = [["i", "l","p"], ["w","l","e"], ["i","e","c"]]
f0 (m, v) = filter (\x -> (x !! 1) == v) m
f (m, v) = map (\x ->[x !! 0, x !! 2]) (f0 (m,v))
view (m, v) = concat ["<html> <body> <h1>", v,"</h1>"
                      , table (f (m,v)),"</body></html>"]
```

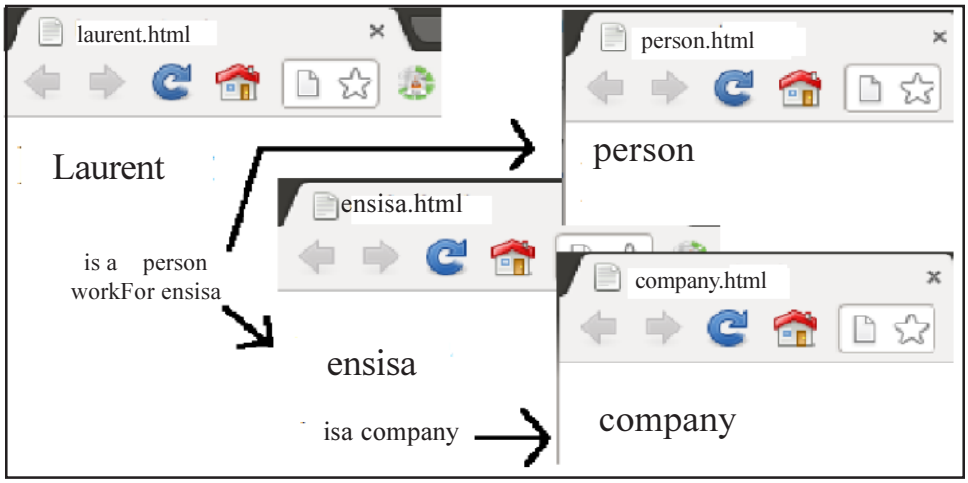

Figure 2. Example of frames and web views

[2]The symbol 8 being absent from the keyboards is replaced by!.

Finally as a third remark, the language L can be extended to more complex languages such as the ones found in description logics, semantic web and ontologies. With the explosion of web technologies, the domain of knowledge management has introduced new standards with in particular RDF and OWL languages. Thus, the Resource Description Framework RDF [6] proposes to use: (labeled directed multi-) graphs to specify web resources, and the eXtensible Markup Language XML [11] or N3 notation for the concrete syntax. In the N3 notation[3], each peace of information corresponds to a triple (subject, predicate, object). A model is queried by using other standards, in particular the SPARQL Protocol And RDF Query Language [3]. As an extension, the Web *Ontology* Language OWL [1] also proposes a set of knowledge modeling languages based on RDF and description logics. An *ontology* is a formal representation of the concepts used by a domain and the relationships between these concepts. Description logics [1, 2] are fragments of First Order Logic that represent knowledge into two parts: the TBox represents the concepts and relations, and the ABox defines the individuals and their assertions. An example of TBox ABox is presented in Figure 3. More precisely, each concept denotes a set, and all the concepts are related into the TBox by using set/relation operators and quantifiers. For instance, if R is a relation and C a concept then $\exists\ R.C$ denotes the set A such as there exists an X with $R(A, X)$ and $C(X)$. As an example, a constraint saying that "*the employees are the persons that work for a company*" will correspond to *employee* $\equiv \exists$ *work for:company.* From such a definition, the domain of ontologies proposes dedicated tools to check that an ABox satisfies the assertions of the TBox, or the consistency of the TBox, with for instance the reasoners Racer [10] or Pellet [20], and the object-oriented editor Protégé[4].

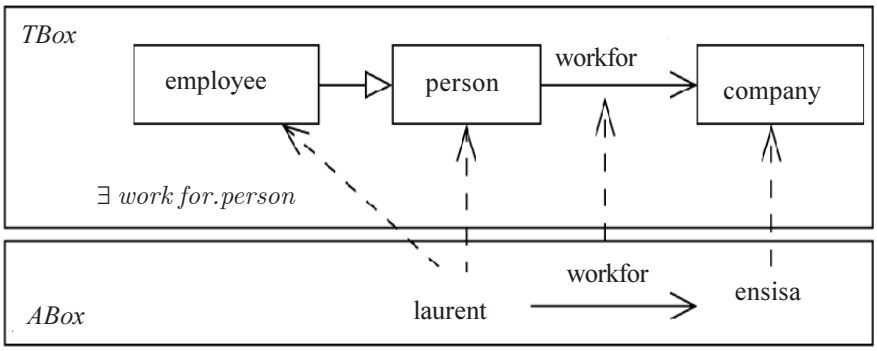

Figure 3. Separation TBox/ABox

The aim of this paper is now to show how the concepts and applications presented above can be "*shifted*" in the world of functional programming, and their advantages in terms of simplicity, extensionability, or usability

2.2 Functional programming, Haskell & the web

Functional Programming (FP) is as old as computer science, and has always been considered from a theoretical point of view. Indeed, FP has a formal foundation (the λ-calculus [4]) that is well suited to reason about programs [12], to improve their performances [8] or to prove properties. Now, the concept of a "*function*" and its interest can be presented by using set theory as follows.

A function f is simply subset of a cartesian product $f \subseteq X \times Y$ such as for all (x, y) and (x', y') in f, if $x = x'$ then $y = y'$ [7]. $F(X, Y)$, also written $X \rightarrow Y$, represents the set of all the functions from domain X to range Y. A higherorder function is a function where X or Y are sets of functions. For instance, the well known function composition (o) is an element of $F(F(Y, Z) \times F(X, Y), F(X, Z))$, i.e. it takes a pair of functions $(f: Y \rightarrow Z, g: X \rightarrow Y)$ and returns a function $(f\ o\ g): X \rightarrow Z$.

From this, functions can be used to specify datatypes. In particular, an inductive set X is a set generated by a family of functions($f_i: X_i \rightarrow X$) called the "*constructors*". For instance, the lists of Xs, written $L(X)$, can be defined by using two functions for the empty list $e: \emptyset \rightarrow L(X)$, and for adding an element to a list $a: X \times L(X) \rightarrow L(X)$. Thus, $[x_1, x_2, ..., x_n]$ is just a shortcut for $a(x_1; a(x_2, ...a(x_n; e())))$. This definition can be translated directly into most of the functional programming languages. For instance, the following code gives an implementation of lists in the functional programming language Haskell [17]: the inductive type/set L is

[3] www.w3.org/DesignIssues/Notation3.html

[4] http://protege.stanford.edu

specified by *data L* = followed by all the constructors separated with the symbol (|). The empty parameter for *E* is omitted, and *E* is simply called a "*constant*".

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

f (e, a) (E) = e
f (e, a) (A (x, y)) = a (x, f (e, a) (y))

The code also presents the higher-order function f that transforms recursively *E* into *e*, and all the *A (x, y)* into *a (x, f (y))*. Thus, the concatenation of two lists will be simply defined by *cat (l, l') = f (l', A) (l)*, the concatenation of a list of lists by *concat (l) = f (E, cat) (l)*, the application of a function *h* to all the elements of a list by *map (h, l) = f (E, a o h) (l)*, etc. As a remark, list data structure is found natively in Haskell: *L (x)* corresponds to [*x*], the empty list is simply represented by [], the add function is given by the infix operator (::), and the concatenation *cat (l, l')* corresponds to the binary operation *l++l'*.

As explained before, lists can serve to implement sets, relations and graphs (e.g. as lists of triples), and the choice of the Haskell language comes from its capabilities. In particular, the language integrates a lot of useful "*syntactic sugar*", pattern matching, operator overloading, etc. For instance, the application of a function *f* to a list *l* can be defined by using the higher-order function *map (f, l)* or list comprehension [*f (x)* | *x* <– l], and the selection of the elements that satisfies a predicate p with *filter(p, l)* or more easily [*x*| *x* <– l, *p (x)*]. This notation has many advantages in querying or managing databases [22], and will be used to simplify the description of the elements proposed. As an illustration, the following function returns the frame *f'* of *v* in a model *m*. It also presents an example of how a (simple) inference rule *r* can be implemented by using these constructs (see *r*): *m* is a graph model represented by a list of triples, r_1 uses list comprehension and pattern matching to get information that can be inferred, and *r* computes the *fix* point of r_1 by adding all the information that can be inferred.

f' (m,v) = [(x, z) | (x, y, z) <–m, y == v]
r (m) = fix (r1, m)
r1 (m)=[("isa", y, "company") | (" workFor", x, y)<–m]
fix (ri, m) = if (m == ri (m) ++ m) then m
else fix (ri (m)++ m)

From the previous principles, many libraries of pre-defined functions and datatypes are available to design "*real-world* " applications. In particular, a web application is mainly composed of three elements with: 1) a front-end user interface, 2) the server is the main application that awaits queries and replies with xHTML documents, and 3) the database to store information to be extracted and displayed in the documents, Figure 4.

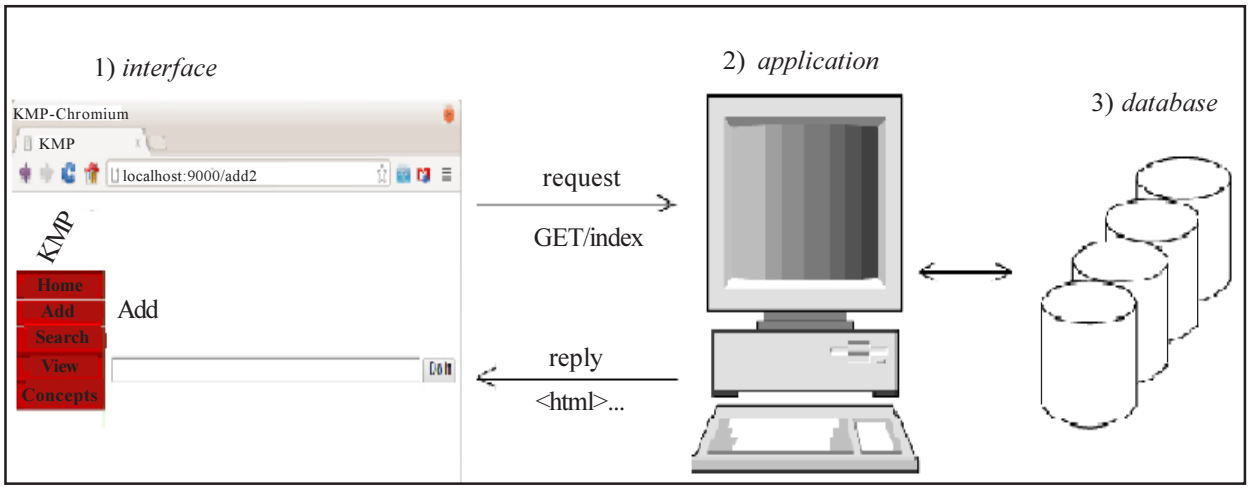

Figure 4. 3-layers architecture

Queries and replies are based on specific languages, and as mentioned before FP is well suited to integrate languages (i.e. texts and grammars). Thus, the queries are fundamentally texts interpreted by the server and composed by: an action to be performed

(e.g. "*GET* "), the main parameter (e.g. "/*index.html*") and optional parameters (e.g. "?*lang = en*"). The extraction of a value from a text is called a parse function [13]. Such a function that extracts a value x from a text (i.e. a list of characters [*Char*] or "*strings*") is modeled by the generic datatype *P x* in the following code. More precisely, this function returns a list of found values *x* together with the rest of the text to be analyzed. Examples of elementary parsers are "*sym*" to read a character and "*str* (*s*)" that awaits a string "*s*" at the beginning of a text. Higher-order functions[5] are then used to compose parsers with "*alt*"ernatives, "*seq*"uences and "*rep*"etitions. As an application, the following code define the "*query*" function used to extract the action A in the query "*GET /A*". Now, the same functions can also be used for interpreting the logical languages presented in the previous section or to define other languages as explained in the next section.

```
type P x = String –> [(x, String)]
sym :: P Char
str :: String –> P String
alt :: P x –> P x –> P x
seq :: P x –> (x –> P y) –> P y
rep :: P x –> P [x]
query = (str "GET /") `seq` (\a –> many sym)
```

For replies, web applications are essentially based on the Hyper Text Markup Language HTML, and more generally XML representations (with xHTML). An XML element is a tree structure composed of: 1) nodes having a tag value and a list of other elements considered as children, and 2) leaves that are simple texts. In FP, this definition can be formalized by a generic datatype (by following the same principles used for the definition of lists or parsers); see the Xml datatype in the following code and [24] for a more detailed presentation. This datatype is then used for web pages, templates, and any kind of information (e.g. RDF documents). In the following code, the "*html*" function defines a template to display a frame "*v*" with properties "*p*". The function "*show*" simply returns the concrete representation of an Xml value (to be sent to the browser). The inverse of "*show*" is "*read* " and is based on the parse functions presented above.

```
data Xml = Node String [Xml] | Leaf String
html v p = Node "html" [
              Node "body" [
                Node "h1" [Leaf v],
                p ]]
show :: Xml –> String
read :: P Xml
— show (html "X" "Y") == "<html><body><h1>X</h1>...</html>
```

Finally, an example of a simple web server can be defined by combining the *query* and *html* functions. In the following code, *m* is the model (e.g. Figure 1), *f* is the function that returns the frame *v* requested by a client (and obtained with the *query* function), and *html* is the template used to display the preceding information. Now, the preceding elements can be improved with more sophisticated templates (see left part of the Figure 4), to integrate navigation between frames (e.g. Figure 2), etc. The particular notation used in following code (i.e. the *do* notation) is based on the concept of "*monad* ", a functional model of actions and **IO** [23].

```
server m = do
s           <– listenOn (PortNumber 80) — Await connexion
(h,_ ,_ ) <– accept s
l           <– hGetLine h
let v = query l                          —  Interpret query
hPutStr h (show (html v f (m,v)))        — Send reply
```

The next section of this paper will now explain how FP can help the integration of both theoretical models (with inference systems) and technologies (with web applications).

[5]Note. These functions are also called "*combinators*" when their definitions omit parameters, e.g. $h = f \circ g$

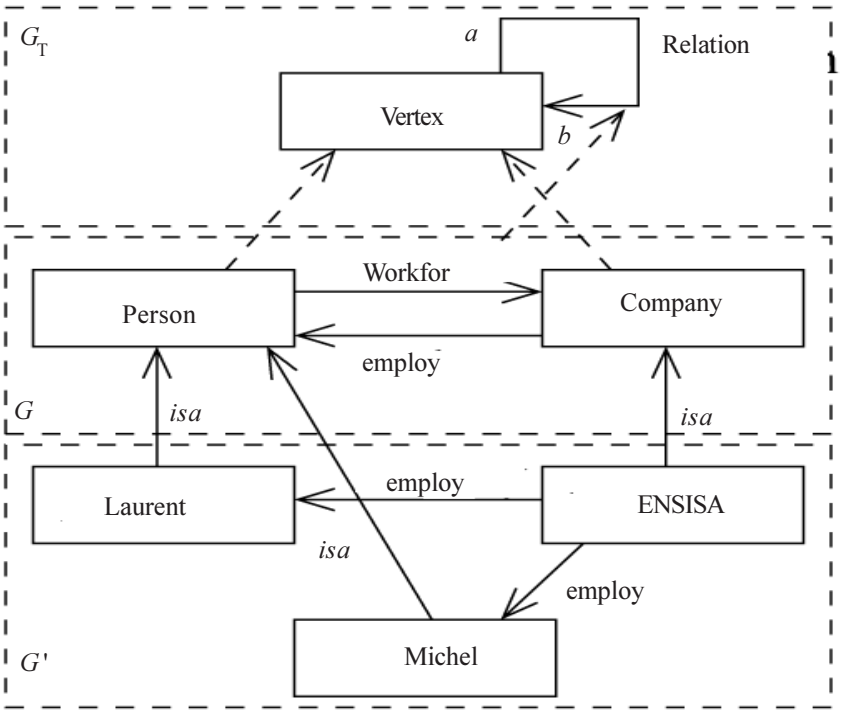

Figure 5. Example of graph data model

3. Functional inference for the web

3.1 Structure, inference and queries

As defined in Part 2.1, a knowledge can be formalized by a relational structure and a labeled graph. This graph can be defined as a set of edges and triples (r, a, b) where r is the label/relation, and (a, b) are the source/destination vertices. A set can be abstracted by a list while considering a function (called "*nub*" in Haskell) that removes all the duplicated elements. Thus, list comprehension and higher-order functions on lists can be used to define and manipulate sets and graphs. In particular, in the following code, if m is a relational structure then s (m) will return the set of symbols S, and r (m) the set of relations R with the interpretation I. The function f (m, v) simply returns the frame of v in m.

```
s m = nub (concat [[a, b] | [r, a, b] <– m])
r m = [(r', [[a, b] | [r', a, b] <– m, r' = = x]) | x <– rs]
  where rs = nub [r' | [r', a, b] <– m]
f m v = [[r, b] | [r, a, b] <– m, a = = v]
```

Next, the information in a graph can be organized into three layers (Figure 5) with: 1) the modeling concepts G (from the TBox for instance), 2) the individuals G' (or ABox), 3) a particular node called "*Vertex*" representing the root to all the modeling concepts G_T . Another example of this organization will be presented in Figure 8. The "*Vertex*" node is introduced as a root to all the modeling concepts G (and to the individuals G'), and it will be used in the next sections as a starting point for surfing (e.g. what are the concepts found in the web application ?).

The graph datamodel in Figure 5 formalizes the "*facts*" in a knowledge model. The "*rules*", the logical language, the satisfaction relation and inference can now be defined upon this structure. The language L and the set of Terms can be formalized by the datatype T and 4 constructors for the facts F, conjunction *And*, implication *Imp*, and the for all quantifier *All*. An example of term is then given by t below[6]. The satisfaction relation ($m \models t$) is then a higher-order function that transforms all the constructors as explained in the following code. In this code, the function *elem* tests if a value is contained by a list/set, (&&) is the boolean

[6]In Haskell, a binary function f (a, b) can be written in its curryfied form $f\ a\ b$, and in using an infix notation a 'f' b

conjunction, and the *all* function tests if all the elements of a list are true (all [*v*1, *v*2,..., *vn*] = = v1&& v2 &&...&& vn). The function "*sub*" substitutes a variable *x* by *v* in a term, and is also an higher order function. Finally, the function *sat′* is a variation of *sat* that adds to a graph the facts to be a model (in the mathematical sense). The function *fix* computes the fix-point model, i.e. the model containing all the inferred facts (see *delta*).

```
data T = F [String] | And T T | All String T | Imp T  T
t = All "X" (All "Y" (((F [ "isa", "X", "person"]) `And` (F ["has", "X", "Y"]))
`Imp`(F  [ "is", "X", "happy"] )))

-- satisfaction relation
sat (F r) m = r `elem` m
sat (And t t') m = (sat t m) && (sat t' m)
sat (Imp t t') m = if (sat t m) then (sat t' m) else True
sat (All x t ) m = and [sat (sub x v t) m | v <– s m]

-- substitutions
sub x v (F r) = F [f k | k <– r]
    where f k = if (k == x) then v else k
sub x v (And t t' ) = And (sub x v t) (sub x v t')
sub x v (Imp t t') = Imp (sub x v t) (sub x v t')
sub x v (All x' t) = if (x / = x') then All x' (sub x v t) else All x' t

— inference
sat' (F r)        m = if (r `elm` m) then [] else [r]
sa' (And t t') m = (sat' t m) ++ (sat' t' m)
sat' (Imp t t') m = if (sat t m) then (sat' t' m) else []
sat' (All x t)  m = concat [sat' (sub x v t) m | v <– s m]

fix m t = if (delta = = []) then m else (fix (m `union` delta) t)
    where delta = sat' t m
```

Thus, the code is a direct translation of the theoretical model introduced in 2.1. By taking back the other elements of this part, the presented grammar (and the user friendly representation of terms), has been integrated by considering the parse functions *P* of the Part 2.2. Thus, a user can define a model in an "*easy*" way (e.g. model.txt below). Among the proposed elements, the "*main*" function returns the graphical representation of all the inferred facts; the "*dot*" function simply converts the list representation of a graph to a format usable by the graph visualization software Graphiz[7]. An example result is presented in Figure 8.

```
— model.txt
workfor (laurent, ensisa) /\
!X.! Y. workfor (X,Y) => isa (Y, company) /\
!X.! Y. workfor (X,Y) => isa (X, person)

— main application
main = do f <– readFile "model.txt"
  let t = readT f :: T
  let m = fix [] t
  writeFile "out.dot" (dot m)

dot m = concat ["digraph g {"
, concat [a ++ "–>" ++b++" [label = "++r++" ];" | (r, a, b) <– m]
, "}"]
```

[7] www.graphviz.org

At this stage, the preceding elements are useful to define a model or to infer facts, but they are not adapted to query a particular information. To overcome this limitation, a simple solution consists in adapting the satisfaction relation as explained in Part 2.1 and the higher order function *sat″* below. Thus, the query *sat″* (read ("!*x*.*isa* (*x*. *person*)"), *m*) will return all the persons in a model m.

sat″ (*F r*) *m* = if (*r* `elem` *m*) then [*r*] else []
sat″ (And *t t′*) *m* = (sat″ *t m*) `inter` (sat″ *t′ m*)
sat″ (All *x t*) *m* = concat [sat″ (sub *x v t*) *m* | *v* <– *s m*]
sat″ (All "*X*" (*F* ["*isa*", "*X*", "*person*"])) *m* == [*F* ["*isa*", "*laurent*", "*person*"]]

As illustrated by the various examples, FP leads to compact implementations of mathematical definitions. Languages are abstracted by datatypes (see *T*) and semantic by higher order functions (see sat and its variations). Concrete syntaxes (and grammars) are added by using the generic functions (*P*) which facilitates the use of the previous elements (e.g. model.txt). Now, a concrete application can be obtained by integrating technological parts and web components.

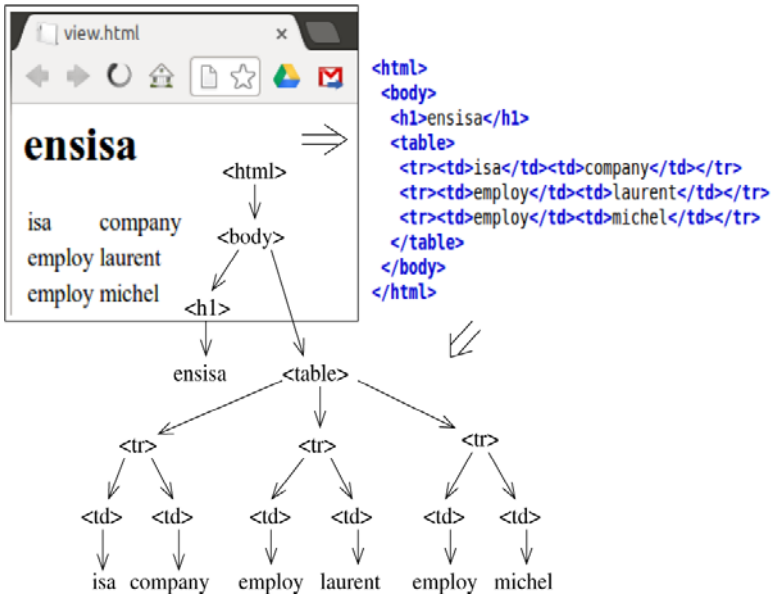

Figure 6. Viewing an information on the web

3.2 Web parts to navigate in, update or query the model

As mentioned before, an information is presented in the web by using the xHTML language. More precisely, a web page defines a tree structure as shown in Figure 6 which is composed of nodes that are tags (e.g. <*html*>) and that have "*children*" (represented by the edges in the figure). Children can be other nodes or leaf values (e.g. *ensisa*). Thus, to layout a particular information will consist mainly in defining a mapping from a part of a data model (e.g. the lower subgraph of the Figure 5) to a web template, and generating the corresponding textual representation [19].

Documents can be specified in FP by using the generic datatype *Xml* presented at the end of Part 2.2. Thus, a template is just a function that takes a data and returns an *Xml* value (e.g. the *html* function already presented) - which is similar to standard technologies such as the Java Server Pages (JSP) from Sun or the Active Server Pages (ASP) from Microsoft. As a remark, our current development also integrates a language for defining templates and each node on the graph data model can have its own template (: *html* being the default one) to add a description, a picture, a table, etc.

As explained, a document is queried by a client (and a browser) by the way of URLs such as *http://server/ensisa*; the query received by the server is then *GET /ensisa* and the last part of this expression (i.e. v) can be extracted with the query parser function. Finally in the examples developed, the server replies by sending to the client the response show (*html v* f (*m*, *v*)) – *f* being the frame of "*v*" (as illustrated in Figure 6). The URLs, or more precisely "*hyperlinks*", are then integrated to the template

model to make possible the navigation in the graph data model (as illustrated in the Figure 2). More precisely, a node such as laurent is replaced by <*a href* = "/*laurent*" *laurent*" > *laurent* </*a*>.

Now, other kinds of pages/templates are generally proposed by a web site to: Create a new information (e.g. insert a new company), Read (or select), Update or Delete an information; this classification is called the *CRUD model* [18]. By taking back the graph G in the Figure 5, a concept such as a "*company*" can be fully characterized by its outgoing edges (e.g. "*employ*" a "*Person*"). Thus, the Creation of a new company will require a HTML form composed at least of an identifier i and an "*employ*" textfield e, Figure 7-a. From this, the system has to add to the data model m: an edge saying that i is a *company* - i.e. (i, *isa*, *Company*), an edge (i, *employ*, e) and an edge (e, *isa*, *Person*). The Update of an information is similar to creation and consists mainly in introducing initial values in the preceding form. Finally, Deletion consists in removing a subgraph from m, i.e. the deletion of a x will be defined by $m0 = \{(v, r, v') \in m \wedge v \neq x \wedge v' \neq x\}$.

Now, all the web pages are generally linked together by a main menu. By using the organization of Figure 5, the main menu can be built automatically by finding all the elements linked to the *vertex* and, for each one, adding links to "*list*" all the values or to "*add*" a new one (Figure 7-b). As a consequence, the add of the graph {(*project*, *isa*, *vertex*), (*company*, *workon*, *project*)} will imply a new entry *project* to the main menu, and a new field *workon* in the form of Figure 7-a.

Finally, this (elementary but generic) web site can be extended with a search bar by adding a new form as illustrated in the upper part of Figure 7-b. The search of a p can consists simply in a redirection to *http://server/p* as mentioned before but a more sophisticated search engine could be proposed. In particular, the theoretical language T (and the satisfaction relation) can be used to search a more precise information. To get a user friendly representation of the queries, the concrete syntax of the language has been improved as follows:

<E> := <Q> | <R>
<Q> := true | <T> <*Q'*>
<Q'>:= and <T> <*Q'*> | epsilon
<R> := if <Q> then <T>
<T> := (<W> <W> <W>)

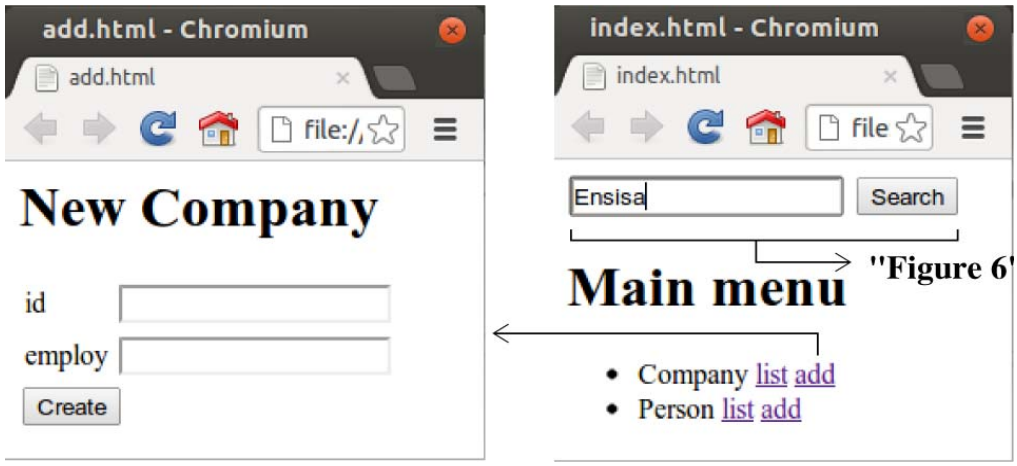

Figure 7. Sample form and main menu

An Expression E from a user is either (|) a Query Q or a Rule R. A query is either "*true*" (and is only here to get the same structure to all the queries), or a Triple of Words (: *isa* (*laurent*, *person*) is simply represented by (*laurent isa person*)) followed eventually by a "*and*" and other triples. A Rule is defined by a "*if*" followed by a Query, a "*and*" and finally a Triple. Some examples of rules are presented in the lower part of Figure 9. If the syntax is more readable, the semantics is the same as T: "*and*" corresponds to ($\wedge$), "*if then*" to ($\Rightarrow$)), and the $\forall X.e$ are obtained by extracting the set of variables (each variable being in uppercase). Thus, by the way of the "*search*" bar, a web site embeds an inference system and is not restricted to static information: new information, and the content of web pages, is deduced from already known information. Finally, facts can be added by using the particular rule: *if true then* (*subject verb object*).

All the presented elements lead to a generic web site that embeds a language for knowledge modeling to add or infer information. This web site, initially empty, is first configured by the concepts of the domain to be considered (e.g. graph G of Figure 5), by using the language introduced above. As explained, this configuration will automatically adapt the main menu of the site. A user can then use the latter, and the corresponding forms, to add a particular information (e.g. G' in the Figure 5) or to navigate within the existing information. To help the search of a specific and precise information, the generic web site is extended with a query language. This language can also be used in the configuration step to define rules (and a dynamic content that can be inferred from the knowledge of the domain).

3.3 Application: the CCAlps project

The generic web site has been considered within the CCAlps[8] project whose objective is to help collaboration between Creative Companies in Alpine space (project number 15-3-1-IT). To proceed, a model of the application domain has been proposed and is (partially) presented in the Figure 8: there are regions that contain companies that have skills and participate to projects, regions organize events for companies to meet each other, etc. Thus, finding the companies C in a region r having the same skill s becomes an easy task by using the query (C *registeredto* r) *and* (C *has* s). As another example, the query (r X Y) will return all information about the region r.

The resulting platform has been called Knowledge Management Platform (KMP) because its objective is to define and share knowledge (about companies and locations). The platform has been personalized with a style sheet to obtain the result of Figure . *Home* is a general introduction to the platform, *Add* is the web page presented in the figure and serves to add facts or rules, *Search* has a similar structure but is used to query an information, *View* displays a part of the data model (e.g. Figure 8) which is helpful to know the elements usable into queries or rules, and Concepts corresponds to the main menu of Figure 7.b.

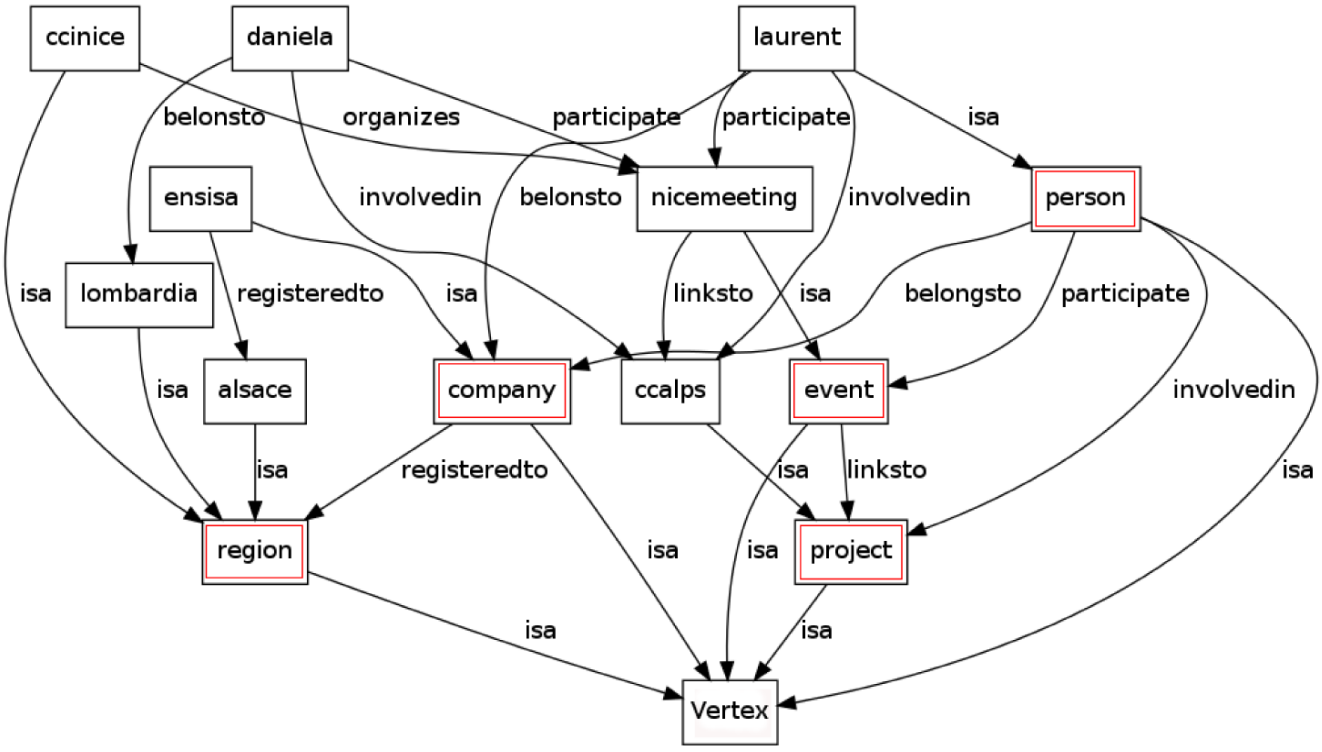

Figure 8. Partial model & metamodel behind CCAlps

The platform is intuitive and extensible: one can query 1) all the concepts C with (C *isa vertex*), 2) all the information about a concept c with (c X Y), 3) all the values V of a concept with (V *isa* c), 4) filter the values having a property p to q with (V *isa* c) and (V p q), etc. The addition of new values can pass by the general Concepts sub menu, or Add as shown in Figure 9. For instance, the add of a new v of type s is obtained simply with if true then (v *isa* c). Finally, the same principle is used to extend the platform

[8] www.ccalps.eu

with new concepts with for instance if true then (c isa vertex). As a particularity, the platform integrates some meta-rules, e.g. if r, s and t are relations then one can define (r *inverse* s), or (t *join1* r) and (t *join*2 s) for $t = r \circ s$, to facilitate the definition of models.

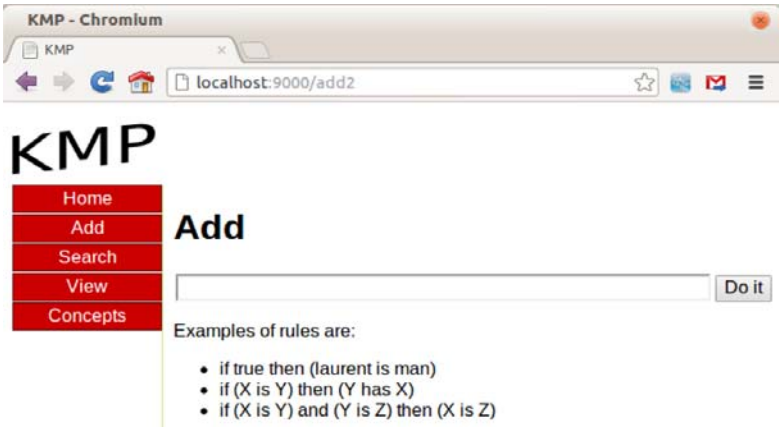

Figure 9. A Knowledge Management Platform (KMP)

4. Conclusion

This paper has presented a functional inference system for the web. More precisely, it has shown how the modern functional programming language Haskell can help to reconcile formal and technological aspects in computer science. As an application, the paper has considered the domains of knowledge modeling (with inference) and web applications. Both are represented by particular datatypes linked together by using generic and higher-order functions, e.g. to compute fix point, parse a particular language or extract/transform information. The result is then a platform dedicated to informa tion sharing and searching (KMP) with an application to the European project CCAlps. This platform proposes a lot of functionalities to add, query or display information in a particular domain. The code is compact (:all the main constituents are contained in the paper) and could be easily extended.

The functional interpretation of the web concepts is not new and the work presented is largely inspired from recent contributions such as [21] for instance. The interest of the work presented is then to explain how such platforms can be improved by more theoretical aspects as illustrated by the embedding of an inference engine, a user friendly representation of first order logic formulas for queries/rules, or the use of a graph-oriented database to store the information.

As a final remark, functional programming is well-suited to "*program calculation*". This means that the functions presented have properties that can be formally proven and are usable to improve performance for instance. The study of such properties is a modern research area (e.g. with cloud computing) and the elements presented can be integrated to this context; this is one of the perspectives considered to the work presented. Another perspective is to take into account other formal aspects of computer science to make them usable in a more "*standard*" application. In particular, the paper has shown how knowledge can be represented by graphs (e.g. Figure 8) and queries are then "*graph patterns*". Graphs and graph patterns are studied from a theoretical point of view by the domain of (attributed) graph grammars, and so it could be interesting to study the concepts that can be integrated to a web application (e.g. to rewrite or generate web pages).

References

[1] Antoniou, G., Van Harmelen, F. (2003). Web Ontology Language: OWL. In Handbook on Ontologies in Information Systems, p. 67–92. Springer.

[2] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., Patel-Schneider, P. F., editors. (2003). The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York, NY, USA.

[3] Barcelo, P., Hurtado, C. A., Libkin, L., Wood, P. T. (2010). Expressive languages for path queries over graph-structured data. In Jan Paredaens and Dirk Van Gucht, editors, PODS, p. 3–14. ACM.

[4] Barendregt, H. P. (1984). The Lambda Calculus – Its Syntax and Semantics, V. 103 of Studies in Logic and the Foundations of Mathematics. North-Holland.

[5] Bratko, I. (1990). PROLOG Programming for Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[6] Candan, K.S., Liu, H., Suvarna, R. (2001). Resource Description Framework: metadata and its applications. SIGKDD Explor. Newsl., 3 (1) 6–19.

[7] Coquand, T. (2008). Constructive mathematics and functional programming. In Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems, ESOP'08/ETAPS'08, p. 146–147, Berlin, Heidelberg. Springer-Verlag.

[8] Gibbons, J. (2002). Calculating functional programs. In Summer School and Workshop on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, p. 148–203. Springer-Verlag.

[9] Goguen, J. A., Malcolm, G. (1996). Algebraic Semantics of Imperative Programs. MIT Press, Cambridge, MA, USA.

[10] Haarslev, V., Möller, R. (2003). Racer: A core inference engine for the semantic web. *In*: Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003), p. 27–36, Sanibel Island, Florida, USA, October.

[11] Harold, E. R., Means, W. S. (2004). XML in a Nutshell. O'Reilly, Sebastopol, C.A., 3rd edition.

[12] Hughes, J. (1984). Why functional programming matters. *The Computer Journal*, 32, 98–107.

[13] Hutton, G. (1992). Higher-order Functions for Parsing. *Journal of Functional Programming*, 2 (3) 323–343, July.

[14] Kifer, M., Lausen, G. (1989). F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. *In*: Proceedings of the 1989 ACM SIGMOD international conference on Management of data, SIGMOD' 89, p. 134–146, New York, NY, USA. ACM.

[15] McCracken, D. D., Reilly, E. D. (2003). Backus-Naur form (BNF). In Encyclopedia of Computer Science, p. 129– 131. John Wiley and Sons Ltd., Chichester, UK.

[16] Meijer, E., Fokkinga, M., Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *In*: Proceedings of the 5th ACM conference on Functional programming languages and computer architecture, p. 124–144, New York, NY, USA. Springer-Verlag New York, Inc.

[17] O'Sullivan, B., Goerzen, J., Stewart, D. (2008). Real World Haskell. O'Reilly Media, Inc., 1st edition.

[18] Pereira, O. M., Aguiar, R. L., Santos, M. Y. (2010). CRUD-DOM: A Model for Bridging the Gap between the Object- Oriented and the Relational Paradigms. *In*: Proceedings of the 2010 Fifth International Conference on Software Engineering Advances, ICSEA'10, p. 114–122, Washington, DC, USA. IEEE Computer Society.

[19] Schwabe, D., Rossi, G., Esmeraldo, L., Lyardet, F. Web Design Frameworks: An Approach to Improve Reuse in Web Applications. In Web Engineering, Software Engineering and Web Application Development, p. 335–352, London, UK, UK,. Springer-Verlag.

[20] Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. Web Semantics Science Services and Agents on the World Wide Web, 5 (2) 51–53.

[21] Snoyman, M. (2012). Developing Web Applications with Haskell and Yesod. O'Reilly Media, Inc.

[22] Trinder, P., Wadler, P. (1988). List Comprehensions and the Relational Calculus. In Glasgow Workshop on Functional Programming, p. 187–202, Scotland.

[23] Wadler, P. (1992). Comprehending monads. *In*: Mathematical Structures in Computer Science, p. 61–78.

[24] Wallace, M., Runciman, C. (1999). Haskell and xml: generic combinators or type-based translation? SIGPLAN Not., 34 (9) 148–159.