20th International Conference on Knowledge Based and Intelligent Information and Engineering Systems

Type-Theoretic Means for Querying Heterogeneous Data Sources Available via Cloud APIs

Domracheva A.I.\textsuperscript{a}, Dr. Shapkin P.A.\textsuperscript{a,b,c*}

\textsuperscript{a}National Research Nuclear University MEPhI (Moscow Engineering Physics Institute), Department of Cybernetics and Information Security, Kashirskoe shosse, 31, Moscow and 115409, Russia
\textsuperscript{b}Innopolis University, 1 Universitetskaya str. Innopolis, 420500, Russia
\textsuperscript{c}Tylip, LLC

Abstract

We propose a unified approach for querying data sources which are accessible in form of heterogeneous APIs rather than some standards compliant databases. Different APIs use different query structures and support different capabilities. Type-theoretic approach is used for query structure representation. Because of the fact that not any API will support the full expressibility of the query language we propose techniques for “widening” the query to make it compatible with the given API possibilities. After the query is executed additional filters are applied to the acquired data. In order to achieve this goal a special set of query expansion rules was developed. It consists of the rules that generate special conversion functions which transform more specific queries into less specific ones.

© 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of KES International

Keywords: Type theory, API integration, propositional calculus.

1. Introduction

A lot of different services provide access to a large amount of the diverse information for their users. These services include SaaS applications, open data banks etc. The provided data is accessible via heterogeneous APIs which are mostly not standardized. The lack of standards is more obvious among SaaS applications: CRM, ERP and other systems which are nevertheless very popular and important for a wide range of businesses and organizations. This situation leads to a high demand for cloud integration providers. This paper proposes an approach to heterogeneous API data access unification by developing an intermediate unified queries layer\textsuperscript{1,2}.

\textsuperscript{*} Corresponding author. Tel.: +7 (903) 731-42-72.
\textit{E-mail address}: pashapkin@mephi.ru

1877-0509 © 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of KES International
doi:10.1016/j.procs.2016.08.089
The search query represents formalized information requirements supplied by the user. However query syntax and search possibilities vary from source to source. For example, some SaaS CRM systems provide search possibilities only for a narrow range of object types: e.g. only for Customers and Deals. Furthermore, the set of data fields available for querying as well as query expressiveness may be restricted: e.g. usage of logical operations may be or may not be supported etc. Nevertheless every API may be considered as a set of functions. This set may be understood as some syntax for querying data.

The problem of unification of the information retrieval queries can be solved within type theory. The search function can be considered as strictly typed function which parameter is the query. The type of query is defined by its structure or types of the filters included in the query. Filter is a structure that keeps information about field type and value by which the data must be selected. The query is a logical expression which is true when all the imposed conditions are met. Thus, the query can be reviewed as a set of filters connected by logical conjunction. According to the Curry-Howard’s isomorphism conjunction corresponds to the product type — a type that allows to build up complicated structures in various ways.

The information-search queries are used in heterogeneous systems, each system supports a specific set of filters. The user’s query can include filters, which are not supported by the system, therefore, it is necessary to develop the rules that “expand” the original query. The query extension means its transformation to the less specific query. The query is analysed in order to identify the filters that can not supported by the target system.

The article is organized as follows. In the second section we consider a method of formalizing API functions, which are intended to retrieve the data. In section three the basics of the Type Theory are presented. Query structure formalization in given in the fourth section. Last two sections of the paper consider the system of query transformation rules. The conclusion summarizes the main results.

2. API Formalization

SaaS applications are required for businesses and organizations. They provide data access via heterogeneous APIs. APIs are mostly specific for each service. All the applications offer the data search, but many of them do not provide extensive search possibilities.

An API can be represented as a set of strongly typed functions providing access to the data. The search conditions are represented as the parameters of these functions. The result of executing such a function executing is a set of objects which correspond the imposed conditions. Thus, the type of function is

\[ \text{DataConditions} \rightarrow \text{List[Concept]}, \]

where DataConditions is a list of applicable filters and Concept represents the required entity type. For simplicity we will understand filters as elementary logical conditions involving a simple logical operation such as ‘=’ or ‘>’ or ‘<’ and etc. Filters may be represented as follows:

\[ \text{field } \langle \text{operator} \rangle \text{ parameter} \]

where the field is an entity attribute, the operator is a simple logical operation and the parameter is a value of field.

E.g. consider a system which has an Employee entity. The system provides a search function for this type.

Suppose that the Employee type has the following fields:

\[ \text{surname, name, position, department, manager}. \]

The system provides functions which only allow to retrieve objects filtering by surname, department and position. Obviously, in some cases this set of filters may not meet the user’s demands. For example, the user wants to get a list of employees who work at the certain department on the certain position under the certain manager.

Definitely, system can not execute this query. It can execute only a “wider” request, i.e. the query with less filters then the user initially set. In example system can filter a data just by department and position attributes. Thus, user gets an excessive data.
It is necessary to develop an intermediate layer to resolve these problems. This layer can analyse user’s request and determinate filters which are supported by API, and which are should be applied on an outside system. Also, this layer allows to transform user’s request to the query structure supported by API.

3. Introduction to Type Theory

Let us introduce some basic definitions of the type theory which we will be using further as the text goes. In formal, the Type Theory (TT) studies processes of type inference and type checking in programs. For this purpose, it is necessary to have a formal representation of programs — \( \lambda \)-calculus, where programs are interpreted as the composition of computable functions. We will be giving only major definitions omitting details\(^5\)\(^6\).

The basic construct in \( \lambda \)-calculus — \( \lambda \)-term — is defined as follows.

**Definition 3.1. (\( \lambda \)-terms)**

- **Variables** are denoted by arbitrary strings of letters and numbers.
- **Constants** are also denoted by strings. We will distinguish them based on a context.
- **Abstraction** of \( \lambda \)-term \( M \) by a variable \( x \) — \( \lambda x.M \) is an unary function of parameter \( x \).
- **Application** is an application of a function (term) \( M \) to an argument \( N \) and is denoted as \( (MN) \). Braces have left associativity and can be omitted if possible.

Basic rule of computing (“reduction”) a value of expression is (\( \beta \)):

\[
(\lambda x.M)N = M[x \leftarrow N],
\]

where \( M[x \leftarrow N] \) denotes the result of substituting all occurrences of \( x \) for \( N \) in \( M \). This rule is also equipped with a set of rules that enable reduction of not only full term but of it’s parts as well.

Types are defined as follows:

**Definition 3.2. (Types)**

- If \( V \) is a type variable or constant then \( V \) is a type.
- If \( V \) and \( U \) are types then \( V \rightarrow U \) is a type.

And finally, the typing rules. Let \( \Gamma \) be some context, then \( \Gamma \vdash m : V \) means that term \( m \) has type \( V \) in a context \( \Gamma \). For instance for simple terms like variables this is stated explicitly. For the consideration of architecture at the top level, without details of the implementation, it is enough to observe *simply typed \( \lambda \)-calculus* which has the following system of typing rules:

\[
\frac{\Gamma \vdash t : V \rightarrow T \quad \Gamma \vdash u : V}{\Gamma \vdash tu : T} \quad \text{(Application)} \quad \frac{\Gamma, m : V \vdash n : T}{\Gamma \vdash \lambda m.n : (V \rightarrow T)} \quad \text{(Abstraction)}
\]

Also, consider the main types that will be needed further. First of all, consider the type which called “generics” in the programming terms. This type allow to operated with heterogeneous data sets. Although, on the one hand, this notation has a complex structure and, in fact, depends on the type of parameter, we understand it as a name of an atomic simple type. For example, the type “optional value of type \( T \)” denoted as \( \text{Option}[T] \) which allows to represent a value that may be missing. So, the term of type \( \text{Option}[T] \) can take value \( t \) of type \( T \), or value \( \text{none} \). The type of value \( \text{none} \) called \( \text{Unit} \) type. \( \text{Unit} \) is a type that allows only one value — \( \text{none} \). In fact, \( \text{none} \) is the only possible result of \( \text{Unit} \) type expression.\(^5\).
Besides generic type, we need to consider another type — product type. In programming languages and type theory, a product of types is another, compounded, type in a structure. The “operands” of the product are types and the structure of a product type is determined by the fixed order of the operands in the product. An instance of a product type retains the fixed order, but otherwise may contain all possible instances of its primitive data types. A product of types is a direct product of two or more types. It is called “pair” \( T ::= T_1 \times T_2 \) if product type consist of two components, otherwise it is called “tuple” \( T ::= \langle T_{1}^{e_{1}}, ..., T_{n}^{e_{n}} \rangle \).

Another structure corresponding to the product type is a “record”. The record differs from the tuple by each field is provided with a label selected from a predefined set \(^3\).

Consider the pair in details. The term pair is defined as \( t ::= (t_1, t_2) \) \( t_1, t_2 \) are the first and second projection respectively. Type of this term is: \( T ::= T_1 \times T_2 \).

Typing rules\(^3\):

\[
\begin{align*}
\Gamma \vdash t_1 : T_1 & \quad \Gamma \vdash t_2 : T_2 \\
\Gamma \vdash \langle t_1, t_2 \rangle : T_1 \times T_2 \\
\Gamma \vdash p : T_1 \times T_2 & \quad \Gamma \vdash p.1 : T_1 \\
\Gamma \vdash p.2 : T_2 
\end{align*}
\]

Another type we should to represent is a \( List[T] \). For all tuples \( T \), type \( List[T] \) describes a list of finite length consisting of elements of type \( T \). An empty list (list without elements) is written in the form \( nil[T] \). The list generated by the addition of a new element of type \( T \) \( t_1 \) at the begin of \( t_2 \) written in the form \( cons[T] t_1 t_2 \). Introduce the concept of “head” and “tail” of list. Under “head” of list undersood the first element of this list and it is written in the form \( head[T] t \). All elements of list without “head” called ‘tail’ and written in the form \( tail[T] t \) \(^3\).

Evaluation rules:

\[
\begin{align*}
& t_1 \rightarrow t_1^* \\
& cons[T] t_1 t_2 \rightarrow cons[T] t_1^* t_2 \\
& t_2 \rightarrow t_2^* \\
& cons[T] t_1 t_2 \rightarrow cons[T] t_1^* t_2 \\
\end{align*}
\]

\[
\begin{align*}
isnil[S]\langle \text{nil}[T] \rangle & \rightarrow \text{true} \\
isnil[S]\langle \text{cons}[T] v_1 v_2 \rangle & \rightarrow \text{false} \\
& t_1 \rightarrow t_1^* \\
& t_1^* \text{isnil}[T] t_1 \rightarrow \text{isnil}[T] t_1^* \\
& head[S] (\text{cons}[T] v_1 v_2) \rightarrow v_1 \\
& head[T] t_1 \rightarrow head[T] t_1^* \\
tail[S] (\text{cons}[T] v_1 v_2) \rightarrow v_2 \\
& tail[T] t_1 \rightarrow tail[T] t_1^* \\
\end{align*}
\]

Typing rules:

\[
\begin{align*}
\Gamma \vdash \text{nil}[T_1] : List T_1 & \quad \Gamma \vdash \text{isnil}[T_{11}] t_1 : \text{Bool} \\
\Gamma \vdash t_1 : T_1 & \quad \Gamma \vdash t_2 : List T_1 \\
\Gamma \vdash \text{cons}[T_1] t_1 t_2 : List T_1 \\
\Gamma \vdash t_1 : List T_{11} \\
\Gamma \vdash \text{head}[T_{11}] t_1 : T_{11} \\
\Gamma \vdash \text{tail}[T_{11}] t_1 : List T_{11} \\
\end{align*}
\]
4. Type-Theoretic Formalization of the Query Structure

The query is a list of applicable filters. Filter is a logical condition imposed on the data. In an elementary case, filter is a single predicate. A single predicate $P(x)$ is all the functions of one variable, that takes all values of argument $x$ from set $M$ and then function takes one of two values: true or false. As a parameter, the function takes an entity from set $M$ (set of all entities). If the value of entity attribute respond to stored condition then function takes $true$, otherwise $false$.

The concept of predicate abstracts the concept of the proposition. The proposition is a sentence which can be true or false. The predicate differs from the proposition by possibility to substitute the arguments in it. So, we formalized filter as a predicate and, in turn, formalization of a query is a logical proposition.

The proposition is a sentence which can be true or false. It can be simple (elementary) or complex (composite). The query is a complex proposition consisting of elementary statements (filters).

The section of logic, which considering a relationship between complex and simple proposition called propositional calculus.

Alphabet of proposition calculus:

- $X_1 \ldots X_n$ — propositional variables. Propositional variable (also called a sentential variable or sentential letter) is a variable which can be true or false. Propositional variables are the basic building-blocks of propositional formulas, used in propositional logic and higher logics.

- Logical constants: $\rightarrow$, $\neg$, $\land$, $\lor$, $\leftrightarrow$

- The concept of a propositional formula. A propositional formula is a type of syntactic formula which is well formed and has a truth value. If the values of all variables in a propositional formula are given, it determines a unique truth value. A propositional formula may also be called a propositional expression, a sentence, or a sentential formula. A propositional formula is constructed from simple propositions, using connectives — logical constants.

Rules which generate propositional formula:

- $X_1 \ldots X_n$ — is a propositional expression (Propositional variable is sentential formula)

- If $X_1$ and $X_2$ — propositional formula, then $(X_1 \land X_2)$, $(X_1 \lor X_2)$, $(X_1 \rightarrow X_2)$ and $\neg X_1$ are also formulas.

- There are no other formulas.

Thus, the query is a complex propositional formula, which is true if all filters return true. In the simplest case the query structure consist of proposition variables — filters related with the logical operation — conjunction. In general, the query may have more complex structure. It is the simplest case of considering query structure for us.

The connection of logic to type theory is represented by the Curry-Howard isomorphism. In the constructive logic proof of the statement $p$ is a demonstrating concrete evidence in favor of $p$. Curry and Howard noted that it is similar to computation. At the level of formulas and types, the correspondence says that implication behaves the same as a function type, conjunction as a product type and etc. According to the Curry-Howard’s isomorphism the product type matches to the logical conjunction.

The most suitable structure for the query is a tuple. The tuple is an ordered set of fixed length.

In practice, it is not always convenient to use such structure as a tuple. The heterogeneous list is a similar structure with the tuple, but list is more usable. Thus, heterogeneous list is an appropriate structure for the query.

The list has the following structure:

\[
List ::= ElemA :: ElemB :: \ldots :: \text{Nil}
\]
where \( ElemA, ElemB, \ldots \) — elements, \( :: \) — the constructor of the list, \( Nil \) — en empty list.

To create the query structure similar to the \( List \), it is necessary to introduce the concept of “empty” query. “Empty” query is a query, which do not containing any filters. Called such type of the query \( AllObjects \).

(Request without any conditions return all data).

It is necessary to consider another point. The filters in a query should be optional because the user can create such request, that not all supported filters will be involved in it. We will reflect this by using the \( Option[:]\) type.

So, the query has the following structure:

\[
\text{Query} :: \text{Option}[\text{FilterA}] \land \text{Option}[\text{FilterB}] \land \ldots \land \text{AllObjects}
\]

where \( \text{FilterA}, \text{FilterB}, \ldots \) — applying filters, \( \land \) — a constructor of the query, and \( \text{AllObjects} \) — an empty query.

5. The formalization of Query Expansion Rules

5.1. Rules

As already mentioned, the query can be formalized as a logical proposition. It is consist of single predicates — filters, which bound with logical operation — conjunction. Propositional calculus is a section of logic, which considering a relationship between complex and simple proposition. Therefore system of “wider” rules will be based on axioms and theorems of proposition calculus.

There is a number of different axiom’s systems in propositional calculus. One of them is:

\[
\begin{align*}
(A_1): & \quad F \rightarrow (G \rightarrow F) \\
(A_2): & \quad (F \rightarrow (G \rightarrow H)) \rightarrow (F \rightarrow G) \rightarrow (F \rightarrow H) \\
(A_3): & \quad F \land G \rightarrow F \\
(A_4): & \quad F \land G \rightarrow G \\
(A_5): & \quad (F \rightarrow G) \rightarrow ((F \rightarrow H) \rightarrow (F \rightarrow (G \land H))) \\
(A_6): & \quad F \rightarrow F \lor G \\
(A_7): & \quad G \rightarrow F \lor G \\
(A_8): & \quad (F \rightarrow H) \rightarrow ((G \rightarrow H) \rightarrow ((F \lor G \rightarrow H)) \\
(A_9): & \quad (F \rightarrow G) \rightarrow (\neg G \rightarrow \neg F) \\
(A_{10}): & \quad \neg \neg F \rightarrow F
\end{align*}
\]

Inference rules:

\[
\frac{F \rightarrow G}{F} \quad \frac{F \rightarrow G}{G} \quad \text{(MP)}
\]

The proof (or inference) of formula \( F \) from set of formulas is a finite sequence of formulas \( B_1, B_2, \ldots, B_k \) such that each formula of it is an axiom or a formula of \( \Gamma \), or can in turn be inferred from the axioms. If there is a conclusion of a plurality of \( F \) from set, it is said that formula \( F \) is derivable and it is written \( \Gamma \vdash F \). The elements from \( \Gamma \) are called the hypotheses. If there is a inference of formula \( F \) in the empty set of hypotheses, then it is said that \( F \) is deducible from the axioms (or that \( F \) is provable), and the sequence \( B_1, B_2, \ldots, B_k \) is called a proof of this formula. \( F \) is called a theorem”.

Let’s consider three axioms: \( (A_3), (A_4), (A_5) \). These axioms can solve the problem of the “extension” query.

\[
\begin{align*}
(A_3): & \quad F \land G \rightarrow F \\
(A_4): & \quad F \land G \rightarrow G
\end{align*}
\]

This axioms can be regarded as rules to extract the necessary elements (filters). By the axiom \((A_3)\) we can get a “head” of the filters list. Analogically, using the axioms \((A_4)\) can be obtained the “tail” of the filters list. After all it is necessary to collect the filters in the correct order. It can be done by using the following theorem:
\((A_5)\): \((F \rightarrow G) \rightarrow ((F \rightarrow H) \rightarrow (F \rightarrow (G \land H)))\)

Formalization of the received system of rules.

Rule \textit{head}:

\[
\Gamma \vdash \text{head} : H \land T \rightarrow H \quad \text{(HEAD)}
\]

says about that it is always possible to receive list’s “head”.

Rule \textit{tail}:

\[
\Gamma \vdash \text{tail} : H \land T \rightarrow T \quad \text{(TAIL)}
\]

says about that it is always possible to receive list’s “tail”.

Rule \textit{cons}:

\[
\Gamma \vdash f : A \rightarrow H \quad \Gamma \vdash g : A \rightarrow T \quad <f, g> : A \rightarrow H \land T \quad \text{(CONS)}
\]

says that if from some term \(A\) it is possible to receive the list’s head \(H\) and tail \(T\), it is possible to receive from this term the necessary list.

Except that, the filters can be optional. It is necessary a new rule, to make optional filters.

\[
A \rightarrow \text{None} \quad \text{(OPT)}
\]

this rule, say, that it is always possible to get \(\text{None}\) value.

However, only this rules are not enough. In this rules system conversion occurs only once by only rule. Thereby it is impossible to build a conversion process by means of this system. Consequently it is necessary to extend this system.

Firstly, change the rule “tail” by the way that it would build a proof process. Rule \(\text{(widTail)}\):

\[
\Gamma \vdash f : T \rightarrow A \quad \Gamma \vdash f \circ \text{tail} : H \land T \rightarrow A \quad \text{(widTAIL)}
\]

this rule is understood as follows: if some expression \(A\) is derivable from the “tail” \(T\), then it is derivable from \(H \land T\).

Secondary, we need some rule for composition of functions. This rule allow to build a conversion process\(^8\)

\[
\Gamma \models f : C \rightarrow B \quad \Gamma \models g : A \rightarrow C \quad \Gamma \models f \circ g : A \rightarrow B \quad \text{(\(\circ\))}
\]

this rule allows to calculate type of a specified term, based on the types of its components. There is a problem that initially, we do not know the term structure and the type variable \(C\) is not predefined. The solution of this problem consists in that the components selected directly from the context, where all types are known. And the rule takes the following form:

\[
\Gamma \models f : C \rightarrow B \quad g : A \rightarrow C \in \Gamma \quad \Gamma \models f \circ g : A \rightarrow B \quad \text{(\(\circ L\))}
\]

This system of the rules allowed only to widen a query. So, if the data will be filtering only by wider request, user will get unnecessary data.
5.2. Examples

E.g. consider a system with entity Employee. Entity Employee has the following set of fields: Name, Surname, Age, Department, Position, Manager, EmployeementDay. The system provides search function pullEmployee. This function allows to filter and receive data using the following filters Surname, Department, Position.

\[
pullEmployee(Option[Surname], Option[Department], Option[Position])
\]

hereby pullEmployee has follow type:

\[
Option[Surname] \land Option[Department] \land Option[Position] \land AllObjects \rightarrow List[Employee]
\]

Surname, Department, Position filters corresponding fields.

Assume that the user wants filter the data in the following fields Surname, Department, Position, Manager. and he build the search function of the type:

\[
Option[Surname] \land Option[Position] \land Option[Manager] \land Option[Department] \land AllObjects \rightarrow List[Employee]
\]

Obviously that the system can not perform the user-defined search function. To convert a user-defined function to the structure, which is supported by the system will use the widen rules.

Construct the tree proof of inference user-defined function. For clarity, we drop the wrapper Option[T], just going to write the first letter of the filter.

6. The Final Query Expansion Rules

The previous system of rules has several weaknesses. Firstly, this system can only “widen” a query. As a result, the user get an excessive data. Secondary, this system generate “bad” solutions. For example, the rule opt can apply in any place, and it is possible to get None instead of filter. Obviously, that the selection on this filter will not carried.

To resolve first problem, it is necessary to filtering data after request to the API. By means of the current system of rules this is impossible. In this system, the filters, which unsupported by API are deleting and the information stored in them not memorize anywhere. The modification of the existing system can resolve this problem.

Consider the current system of rules. Rule of composition intended for built of the proof process. Thus, this rule isn’t necessary to change because of it does not directly affect on the query transformation. The rule “head” can be regarded as a rule by which the necessary filters can be identified, it can be assumed that it does not remove the unsupported filters. The rule “cons” collects the filters in the correct order. It isn’t delete filters, thus, it isn’t necessary to modify this rule. Finally, the rule “widTail” exactly responsible for the deleting of the filters. To modify this rule, introduce two functions: predicate and filter:

- **predicate** is a function which will contain the “removed” filter (predicate).
- **filter** is a function which takes as parameters the data which will filtering and the filtering condition.
Consider the rule \textit{widTail}.

Rule (\textit{widTail}):

\[
\frac{
\Gamma \vdash f : T \rightarrow A
}{
\Gamma \vdash f \circ \text{tail} : H \land T \rightarrow A \quad (\text{WIDTAIL})
}
\]

Now, this rule delete unsupported filters not remembering them. This rule schematically works as follows:

\[
h :: t \rightarrow f(t) \rightarrow \text{List}[A],
\]

i.e. take the tail of the list and apply it to the function f, so it is obtained the filtering and excessive data.

Now consider another scheme rule “\textit{widTail}” allow us to filtering by “deleting” filters.

\[
\frac{
\Gamma \vdash f : T \rightarrow A
}{
\Gamma \vdash \text{filter} \circ (f \times \text{predicate}) \circ <\text{tail}, \text{head}> : H \land T \rightarrow A
}
\]

The same way rule \textit{tail} should be transformed.

In this approach, all supported by API filters will be applied twice. Firstly, by means of API, and secondary, then will be applied “removed” filters.

To resolve second problem ("bad" solutions) we propose to introduce a ranking of solutions, and choose and always choose the solutions that contain the maximum number of supported (by API) filters.

7. Conclusion

The article presented the universal means of accomplishment search queries to heterogeneous systems. The query is modelled as a particular logical statement which consists of imposed conditions. The link between a logic system and the type theory is the Curry-Howard isomorphism. We presented the approach to carry out the required query transformations based on propositional logic. Query rewriting rules where obtained from the axioms of the propositional calculus.

8. Acknowledgement

The research was supported by a Russian Foundation for Basic Research grant (Project № 16-31-50011) and the Competitiveness Program of NRNU “MEPhI”.

References