Implementing GraphQL as a Query Language for Deductive Databases in SWI–Prolog Using DCGs, Quasi Quotations, and Dicts

Falco Nogatz Dietmar Seipel
University of Würzburg, Department of Computer Science,
Am Hubland, D – 97074 Würzburg, Germany

{falco.nogatz,dietmar.seipel}@uni-wuerzburg.de

The methods to access large relational databases in a distributed system are well established: the relational query language SQL often serves as a language for data access and manipulation, and in addition public interfaces are exposed using communication protocols like REST. Similarly to REST, GraphQL is the query protocol of an application layer developed by Facebook. It provides a unified interface between the client and the server for data fetching and manipulation. Using GraphQL's type system, it is possible to specify data handling of various sources and to combine, e.g., relational with NoSQL databases. In contrast to REST, GraphQL provides a single API endpoint and supports flexible queries over linked data.

GraphQL can also be used as an interface for deductive databases. In this paper, we give an introduction of GraphQL and a comparison to REST. Using language features recently added to SWI–Prolog 7, we have developed the Prolog library *GraphQL.pl*, which implements the GraphQL type system and query syntax as a domain–specific language with the help of definite clause grammars (DCG), quasi quotations, and dicts. Using our library, the type system created for a deductive database can be validated, while the query system provides a unified interface for data access and introspection.

1 Introduction

In times of big data, the increasing growth of data sets is a big challenge for both data storage and data access. The traditional back–end services became more diverse in the recent past. Today, relational databases are used side by side with document–oriented NoSQL databases, each with a different data query mechanism. To bridge over the various query languages, abstraction layers have been introduced.

In October 2015, Facebook open–sourced *GraphQL* [2], an application layer query language, which has been internally developed since 2012. Today, most of Facebook's applications make use of GraphQL as the data–fetching mechanism, resulting in hundreds of billions of GraphQL API calls a day [1].

The need for GraphQL arose while developing native mobile applications side by side with web applications. In order to share the code base where possible, a unified application programming interface (API) for data access should be provided. At the same time, the introduction of an abstraction layer hides the internals like the used database and schemas. In an agile development cycle, this allows to change front–end and back–end technologies simultaneously without impeding each other.

Because of various problems and deficiencies in the alternatives of such an abstraction layer and for accessing hierarchical data, a new query language has been defined. The most popular alternative architectural style REST [3], the representational state transfer, appeared unsuitable for applications that need a lot of flexibility when accessing the data while reducing the number of round–trip times

S. Schwarz and J. Voigtländer (Eds.): 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming (WLP'15/'16/WFLP'16).
EPTCS 234, 2017, pp. 42–56, doi:10.4204/EPTCS.234.4

in a distributed system. As a result, GraphQL provides only a single API endpoint for data access, which is backed by a structured, hierarchical type system. Because of this, the GraphQL specification defines besides the query language mechanisms for query validation and provides self–descriptiveness by introspection.

This makes GraphQL a good fit for a unified abstraction layer for static, predefined deductive database queries. The type system is storage–independent and can easily integrate various data sources, like relational and document–oriented databases as well as file–based data formats. With the deductive database system DDBASE [7], we already have available a system to connect various storage back–ends like CSV and SQL. It is part of the Dislog's Developer Toolkit DDK[1], which also contains tools for querying, updating and transforming XML data. With GraphQL, this results in a interface, which can be integrated in existing back–end environments and interact with any programming language used for the front–end development.

In this paper, we choose to use SWI–Prolog to develop a GraphQL server implementation, called *GraphQL.pl*. With the use of definite clause grammars (DCG), Prolog gives a natural approach to implement the type system and query document as an external domain–specific language so that the code examples used in the GraphQL specification can be used directly to define the GraphQL server. With quasi quotations, which were introduced in SWI–Prolog of version 6.3.17, this domain–specific language can be directly embedded into normal Prolog code, which lowers the barrier for GraphQL–experienced developers who are not yet familiar with Prolog. For the sake of a shorter and prettier syntax, which is more familiar to developers coming from other programming languages, our implementation makes great use of features supported only by recent SWI–Prolog releases. Therefore *GraphQL.pl* requires SWI–Prolog of at least version 7.

The paper is organised as follows: Section 2 introduces GraphQL and gives a comparison to the well–established architectural pattern REST. In Section 3, we discuss the definition of GraphQL as a domain–specific language in Prolog. Implementation details of *GraphQL.pl* are presented in Section 4, with a focus on the definition of GraphQL's type system using definite clause grammars, quasi quotations, and dicts. Section 5 presents a use case of our implementation to define a standalone GraphQL server. Finally, we conclude with a summary and discussion of future work in Section 6.

2 Background: The GraphQL Application Layer

Although used several years in production, the GraphQL specification is still under active development. The most recent Working Draft specification[2] is of April 2016. Most of the changes since its initial publication in October 2015 have been clarifications in the wording.

Along with the GraphQL specification, Facebook published a reference implementation of a GraphQL server using JavaScript[2]. This reference implementation only provides base libraries that would be the basis for a GraphQL server and is not yet feature–complete. In contrast to our implementation, the JavaScript module does not support the syntax used in the GraphQL specification, i.e. the types have to be specified in a proper format based on JavaScript, while *GraphQL.pl* implements them as a domain–specific language.

Because GraphQL got lots of exposure recently, there are GraphQL implementations in most popular programming languages, including PHP, Java, C/C++, and Haskell. There is a large number of tools to interact with GraphQL or connect it with existing back–end technologies. The *Awesome list of GraphQL*

[1] `http://www1.pub.informatik.uni-wuerzburg.de/databases/ddbase/`

[2] Facebook on GitHub: GraphQL.js, `https://github.com/graphql/graphql-js`

and Relay[3] collects a great number of existing GraphQL implementations, tools and related blog posts and conference videos.

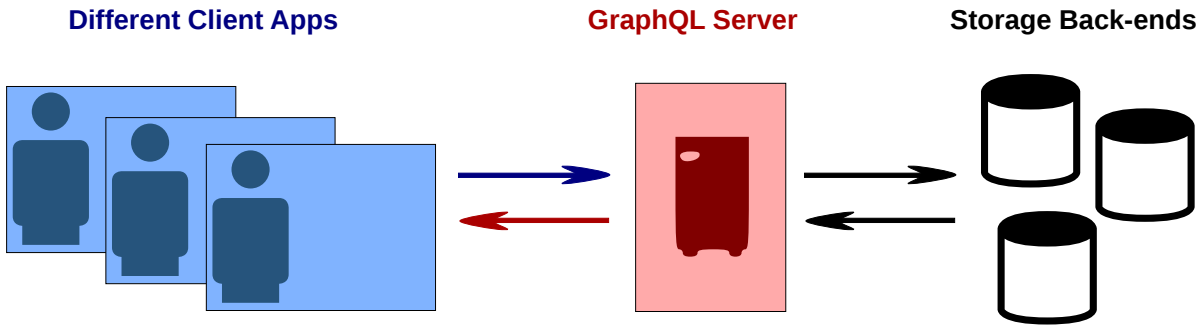

Figure 1: Illustration of the role of the GraphQL Server in a distributed system

The GraphQL server acts as a unified layer for data access and manipulation. In a distributed system, it is located at the same layer like REST, SOAP, and XML–RPC, that means it is used as an abstraction layer to hide the database internals. Similar to these architectural patterns, GraphQL does not support ad–hoc queries. That means that unlike SQL, the general structure of possible queries is defined by the GraphQL server provider in advance. It is not possible to join different entities in a single query, unless their relation has been specified in the GraphQL type definition.

As illustrated in Figure 1, client applications written in various programming languages can send requests to the GraphQL server, which validates and executes the query by collecting the information from the various data sources. This protocol is a classical request–response message exchange pattern, with the following two messages:

– a request, which must be provided in a well–defined query language and send as plain string to the single GraphQL endpoint, and

– a response of the GraphQL server specified as a JSON document.

In most applications, complex, structured data is requested from the API. To provide a single entity with all its relations (and possibly their relations, etc.) from a single API endpoint, a GraphQL query is structured hierarchically. The structure of the query represents the data that is expected to be returned. In general, each level of a GraphQL query corresponds to a particular type. They can be nested and even recursive. The result document is a set of entities with their relations specified in the type system. This clarifies the name `GraphQL`, as these entities and relations can be thought of as a graph.

As a motivating example, we will consider a GraphQL server instance to access and manipulate data of persons along with their favourite books. Figure 2 presents on the left–hand side an example query document to get some basic information about the person named `Alice`, i.e. her `name` and `age`; the attribute `age` should be renamed to `years` in the result set. The query includes her favourite books with their title and authors. The string–based format of the request message presented in the example strictly follows the GraphQL specification. The aim of our work is to implement this format as a domain–specific language in Prolog using DCGs.

On the right–hand side of Figure 2, an example result for the query is presented. By definition of the GraphQL specification, the returned data has to be encoded as a JSON document. The structure and

[3] `https://github.com/chentsulin/awesome-graphql`

```graphql
query getAlice {
   person(name: "Alice") {
      name
      years: age
      books(favourite: true) {
         # is implicitly a list
         title
         authors {
            name
         }
      }
   }
}
```

GRAPHQL

```json
{ "data": {
   "person": {
      "name": "Alice",
      "years": 31,
      "books": [
        {
          "title": "Moby-Dick",
          "authors": [{
            "name": "H. Melville"
          }]
        }
      ]
   } } }
```

JSON

Figure 2: Example GraphQL query and corresponding result in JSON

entities have to follow the request: only keys which were specified in the query are allowed to be part of the result document.

2.1 Type System

Being part of the data access and manipulation layer, GraphQL does not support ad–hoc queries like most standard database drivers provide. While with SQL it is possible to dynamically create queries and join multiple tables with almost no restriction, GraphQL queries have to satisfy the given type system defined by the GraphQL server administrator. The entire type system is called the *schema*.

GraphQL's type system is very expressive and supports features like inheritance, interfaces, lists, custom types, enumerated types. By default, every type is nullable, i.e. not every value specified in the type system or query has to be provided.

```graphql
type Person {
   name:    String!
   age:     Integer
   books(favourite: Boolean): [Book]
   friends: [Person]
}

type Book {
   title:   String!
   authors: [Person]
}
```

GRAPHQL

```graphql
type Query {
   person(name: String!): Person
   book(title: String!):  Book
   books(filter: String): [Book]
}
```

GRAPHQL

Figure 3: Type definitions for the motivating example query of Figure 2 and GraphQL's `Query` type

In Figure 3 we present a minimal definition of a type system to satisfy the example query of Figure 2. It defines the two types `Person` and `Book` as object types, i.e. as a number of field–value–pairs. The fields can have arguments. For example, in order to retrieve the books of a `Person`, it can be specified to return only favourite books by providing an appropriate `Boolean` flag.

The two types `Person` and `Book` reference each other, i.e. the underlying model for an author is a person. The type `Person` is recursive since the `friends` field returns a list of `Person`. `String`, `Integer`, and `Boolean` are scalars and among others primitive values defined in the GraphQL specification. The exclamation mark declares values which must not be `null`, i.e. every `Person` must specify a `name` value. Types enclosed by square brackets like `[Book]` are ordered collections of the given type.

Every GraphQL type system must specify a special root type called `Query`, which serves as the entry point for the query's validation and execution. As illustrated in Figure 3, the fields of this object type can also have parameters, here `name` and `title` which are of the type `String` and must be provided (as denoted by the exclamation mark).

A more detailed introduction to GraphQL's type system, which supports inheritance and composition through fragments, is out of scope of this work. Nevertheless, our presented *GraphQL.pl* system implements the more complex properties of the type system like interfaces, unions, enums, and execution directives.

We want to highlight that the GraphQL specification standardizes only the format of the query document, but not how to denote the type system. The query language is consistent for all GraphQL server implementations and related tools and every GraphQL server has to implement it, independent of the underlying programming language. As opposed to this, the mechanism to specify the type system is currently closely related to the used GraphQL system. For example, in Facebook's reference implementation, the types are defined by calling appropriate JavaScript functions. Instead we propose to use the same domain–specific language as used for the query document. This also eases the usage of the GraphQL specification, because the type examples there are denoted in a format similar to the query document, too.

2.2 Relation to the Communication Architecture REST

The type system of GraphQL is similar to the definition of resources in the REST architecture. In both architectural patterns there are no ad–hoc queries, instead the system's types are defined in advance, specifying possible parameters and the output format.

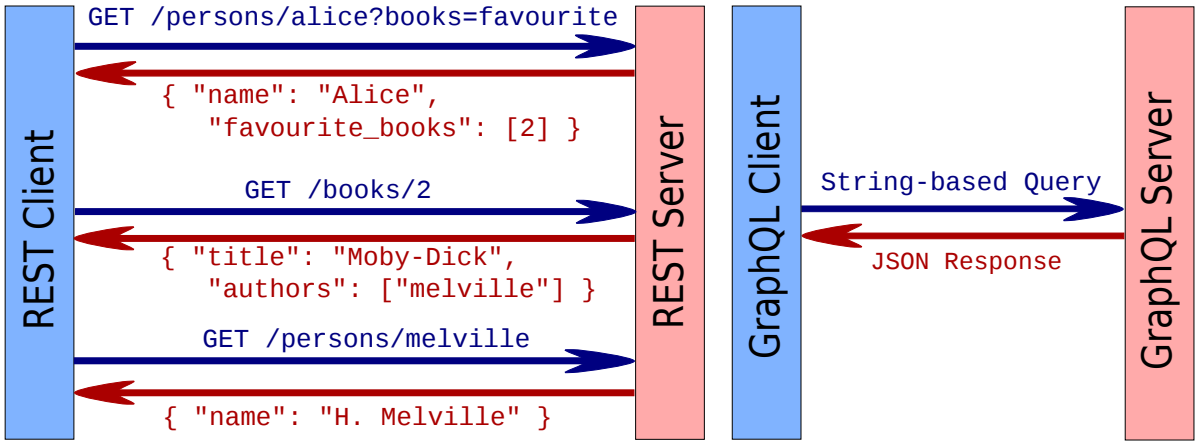

Figure 4: Illustration of the request–response message sequences of a complex query in REST and GraphQL

Both GraphQL and REST are independent of the underlying programming language and used database back–ends. The communication with the client is based on a request–response message exchange pattern. But in contrast to GraphQL, a REST interface in general provides multiple API endpoints, one for each resource. For example, to request complex data consisting of multiple entities and different types, a REST–based protocol has to address multiple resources, resulting in a sequence of round–trip times. The two accessing patterns are illustrated in Figure 4 using the example of requesting a person with its favourite books.

The REST–based data access approach has the need of three round–trip times to collect the required information, which must be transformed into a single object on the client–side. Besides this overhead, REST supports caching and partial documents. The often used transport protocol HTTP adds support for authentication and authorisation, too. These features as well as the freedom of choice with regard to the used data exchange formats are not possible with GraphQL as of yet. In Table 1 we give an overview of the properties of REST and GraphQL.

Property	**REST**	**GraphQL**
API Endpoints	multiple	single
Message format for queries	string–based	string
Message format for mutations	any	string
Message format for response	any, especially hypermedia	JSON
Type System	weakly typed, no machine–readable metadata	strongly typed, meta data for introspection
Built–in features	cacheable	query validation, introspection

Table 1: Comparison of REST and GraphQL

3 Definition of GraphQL as a DSL

The aim of our implementation is to support the GraphQL syntax presented in its specification to ease the use of the given examples with our Prolog implementation. The syntax for query documents is well–defined in the specification. However, the format to declare types[4] is not standardized at all and might therefore be different in GraphQL implementations of other programming languages. We choose to follow the format given for the code examples in the GraphQL specification. In this way, the code snippets mentioned in the documentation can easily be used to create a standalone GraphQL server.

To support the GraphQL syntax in *GraphQL.pl*, we define an appropriate domain–specific language (DSL), for what Prolog is very suitable. Simple languages can be modelled directly using Prolog syntax only, i.e. by defining terms and declaring appropriate operators. For instance, the JSON data format as presented on the right–hand side of Figure 2 is valid Prolog syntax, although a shorter notation is feasible using Prolog atoms.

In a similar way, a subset of the GraphQL language could be represented by Prolog terms with only minimal modifications, i.e. the language could be implemented as an internal DSL. As a result, it would

[4]The GraphQL types are declared when setting up a GraphQL server, generally by the database administrator.

be possible to use the GraphQL examples directly within Prolog, without having the need to write a parser first. The definition as an internal DSL would also profit from tools actually written for Prolog.

We want to discuss the definition of GraphQL as an internal DSL in Prolog based on the declaration of the type `Person` as presented in Figure 3. By defining the `(!)/1` suffix operator, the declaration of non–nullable types is syntactically valid. Similarly, we define `type/1` as a prefix operator. Since whitespaces and commas are insignificant for GraphQL and type names can start with a lowercase letter, the `Person` type definition of Figure 3 could be expressed by the term presented in Figure 5, which is valid syntax of standard Prolog.

PROLOG
```
:- op(800, xf, !).
:- op(650, fx, type).
?- Person = type person(
               name: string!,
               age: integer,
               friends: [person],
               books: [book] ).
```

Figure 5: Operator definitions to define GraphQL as an internal DSL

The notation of Figure 5 almost follows the GraphQL specification, only the curly braces had to be replaced to achieve a valid Prolog term[5]. Nevertheless, this approach of defining the GraphQL type specification as an internal DSL in Prolog is very restrictive:

– Types have to start with a lowercase letter to not create a variable. Although this is allowed according to the GraphQL specification, it is convention to start type names with an uppercase letter. Especially it is not possible to reference built–in types, since `String`, `Integer` etc. all start with an uppercase letter.

– Because we replaced curly with normal braces, query documents could contain compounds with two pairs of brackets: one for the arguments, another for nested items. In the query document of Figure 2, this results in `person(name: 'Alice')(name, years: age, ...)`, which is not valid in Prolog[6].

Because of these restrictions and since *GraphQL.pl* shall completely support the syntax given in the GraphQL specification, the definition of the language as an external DSL might be more reasonable. In the following we will develop a way of embedding GraphQL in SWI–Prolog 7 using quasi quotations.

4 Implementation Using Features of SWI–Prolog 7

We have implemented the *GraphQL.pl* system in SWI–Prolog [8], using features recently added to SWI–Prolog of version 7 [9]. Our implementation makes great use of quasi quotations, definite clause grammars, and dicts.

[5] Since the introduction of dicts in SWI–Prolog 7, `person{}` would be syntactically correct, i.e. it is not necessary to replace the curly braces. We present dicts in Section 4.3.

[6] Even using dicts in SWI–Prolog 7, i.e. to use the term `person(name: 'Alice'){name: _, years: age, ...}`, this would not be valid, since a dict's tag must be either a variable or atom.

4.1 Quasi Quotations

For more complex languages or if Prolog's syntax is too restrictive, other mechanisms for the definition of external DSLs have been discussed [10]. Since Prolog has poor support for quoting long text fragments, quasi quotations have been introduced to SWI–Prolog in 2013 [11]. They are inspired by Haskell and in similar form also part of other programming languages like JavaScript (called *tagged template strings*). Quasi quotations in SWI–Prolog have been used to implement external languages like HTML, SQL, and SPARQL.

The basic form of a quasi quotation is of the following form, where `Tag` is a callable Prolog predicate with an arity of 4 and `Content` is a string:

```
{|Tag||Content|}      e.g.      {|html||<p>Hello World!</p>|}
```

For better readability in this paper, we will enclose the `Tag` by whitespaces, which is still valid SWI–Prolog 7 syntax. The quasi quotation presented before invokes the following call by the Prolog parser. Because quasi quotations are resolved using Prolog's term expansion mechanism, the following call is invoked at compile time:

```
call(+TagName, +ContentHandle, +SyntaxArgs, +Vars, -Result)
```

The quasi quotation's `Tag` can be any term with the functor `TagName/TagArity`. The `Tag`'s arguments are passed as a list `SyntaxArgs`[7], which guarantees that the `TagName` predicate is always of arity 4. The `Vars` argument provides access to variables of the quasi quotation's context. `ContentHandle` is an opaque term that carries the content of the quasi quoted text and position information about the source code. It is usually passed to `with_quasi_quotation_input/3` or `phrase_from_quasi_quotation/2`. While the first predicate creates a stream, we are using `phrase_from_quasi_quotation/2`, which parses the enclosed string according to a given DCG. For instance, Figure 6 illustrates the definition of the {| type || ... |} quasi quotation, which processes the given `Content` by using the non–terminal `dcg_type` of the DCG.

PROLOG
```
:- use_module(library(quasi_quotations)).
:- quasi_quotation_syntax(type).

type(ContentHandle, SyntaxArgs, Vars, Result) :-
   phrase_from_quasi_quotation(dcg_type(Vars, Result), ContentHandle).

dcg_type(Vars, Result) --> ...  % DCG definitions
```

Figure 6: Definition of the `type` quasi quotation

In our implementation, we use quasi quotations for being able to embed the query document as well as the type definitions in the Prolog source code. In order to define types and the overall schema, *GraphQL.pl* provides several quasi quotation tags, for example `type/4` and `schema/4`. Figure 7 gives an example on how to use the quasi quotations to define types and a schema. All user–defined type names starting with an uppercase letter create corresponding Prolog variables, i.e. the quasi quotations in Figure 7 make the variables `Person`, `Book` and `Query` available. In this way, it is possible to reference the `Book` type in the type definition of `Person` and vice–versa. In contrast, the built-in types like `Integer` create atoms beginning with a lowercase letter, i.e. `integer` etc.

[7] This can also be expressed as: `Tag =.. [TagName|SyntaxArgs], length(SyntaxArgs, TagArity)`.

Besides these tags, we define quasi quotations for simple, text–based specifications of `enum` types and `interface` declarations. The query and mutation documents can be specified with a quasi quotation as well, although in general they are given as strings as part of the GraphQL request sent to the server.

```
:- use_module(library(graphql), [type/4, schema/4]).                    PROLOG
example_schema(S) :-
   Person = {| type ||
      name:                         String!
      age:                          Integer
      books(favourite: Boolean): [Book]
      friends:                      [Person]
   |},
   S = {| schema ||
      type Query {
         person(name: String!): Person
         book(title: String!):  Book
         books(filter: String): [Book]
      }
      type Book { ... }
   |}.
```

Figure 7: Definition of the type `Person` and schema `S` using quasi quotations

4.2 Definite Clause Grammars

The GraphQL specification provides a grammar to define the format of a query document. It is the basis for the DCGs [4] used in our implementation to parse the query document. For parsing the type definitions, the same mechanism is used.

In Figure 8 we present the simplified grammar to parse the `type` quasi quotation. It is used by `phrase_from_quasi_quotation/2` as shown in Figure 6. For the sake of simplicity, we omit the generation of the internal representation `Result`, and also the handling of whitespaces and the specification of minor DCG rules. Using this grammar, the type definition specified in Figure 7 can be processed.

Most of the grammar specified in the GraphQL specification for query documents has been translated into a DCG, resulting in more than 50 non–terminals just to parse queries. The entire DCG rule base to parse the type system as well is implemented in more than 600 lines of code with more than 80 non–terminals. Their names are based on the grammars presented in the GraphQL specification. However, to define sequences of elements, the DCG easily gets bloated by helping rules. Because the Extended Definite Clause Grammars [5] are also part of the DDK, we intend to adopt EDCG rules for writing more powerful grammars in a simpler way in the future.

4.3 Internal Representation Using Dicts

The DCG is used to parse the given quasi quotations of type definitions and query documents. Along with this, we create an abstract syntax tree (AST) as an internal representation. We use SWI–Prolog's dicts to create nested terms holding the given information.

With SWI–Prolog 7, dicts were introduced as a new data type for named key–value associations [9].

```
dcg_type(_Vars, _Result) -->        % ignore for simplicity          PROLOG
   list_of_type_definition.         % resolves to 'type_definition'
type_definition -->
   name,                            % the type's name in the object
   ( arguments | "" ),              % optional, e.g. favourite: Boolean
   ":",
   type.
type -->
   ( named_type                     % e.g. Integer
   | list_type                      % e.g. [Book]
   | non_null_type ).               % e.g. String!
named_type -->
   ( primitive_type                 % built-in type of GraphQL
   | prolog_var_name ).             % part of library(dcg/basics)
list_type --> "[", type, "]".
non_null_type -->
   ( named_type | list_type ),
   "!".
```

Figure 8: Extract of the DCG to parse the `type` quasi quotation

In previous versions this was often realised using a list of `Key=Value` pairs. For example JSON documents were encoded as what follows:

```
json([ name="Alice", years=31, ... ])
```

This notation does not ensure the uniqueness of the keys and does not provide a short notation to directly access a child node. The new syntax of dicts resembles the one of JSON and should be more familiar for developers of other programming languages. It is of the following form[8]:

```
Tag{ Key1: Value1, Key2: Value2, ... }
```

The `Tag` is either an atom or a variable. We also make use of the anonymous dict `_{...}`, where `Tag` is the anonymous variable. For example, the type information created by the `type` quasi quotation are encoded by a dict of the following form:

```
object{
   fields: _{
      FieldName1: field{
         type: Type1,
         resolver: Resolver1
      }, ... },
   resolve: Resolver }
```

[8] In the following, our code examples will include both quasi quotations and dicts, which might confuse the reader. Although their visual appearance is similar, they can easily be distinguished by their start token:

- quasi quotations are opened by `{|`,
- dicts with `Tag{`.

This nested `object{}` dict contains a collection of type definitions as the `fields` entry. It corresponds to the `list_of_type_definitions` DCG rule of Figure 8, i.e. it has a `FieldName` for every `type_definition`. The `resolve` entries are automatically created for every type and field. They are used to specify how to retrieve the data for this component and are unbound at first. In Section 5 we introduce their usage in detail.

Dicts can be unified following the standard symmetric Prolog unification rules, although the unification will fail if both dicts do not contain the same set of keys:

```
?- p{ a: 1, b: 2 } = P{ a: A, b: B }.
P = p, A = 1, B = 2.

?- p{ a: 1, b: 2 } = P{ a: A }.
false.
```

Besides this, there are two methods to access a single value of the `Key` in the dict. With `Dict.Key` the related value is retrieved. Since this *dot notation* will throw an error if the given `Key` does not appear in the `Dict`, we prefer using `Dict.get(Key)`, which instead silently fails for missing keys and can therefore be used as a pre–condition in a Prolog rule.

After parsing the quasi quotations using DCGs, the GraphQL schema and types are represented by dicts. The AST generation was omitted in the code example of Figure 8 for the sake of simplicity. In Figure 9 we present an extract of the generated dicts.

PROLOG

```
example_schema(S) :-
   Person = object{
      fields: _{
         name: field{ type: string, nonNull: true, resolve: _ },
         age: field{ type: integer, resolve: _ },
         books: list{
            arguments: _{ favourite: field{ type: boolean } },
            kind: field{ type: Book }, resolve: _ },
         friends: list{ kind: field{ type: Person }, resolve: _ } },
      resolve: _ },
   S = schema{
      query: object{
         fields: _{
            person: field{ type: Person, ... },
            book: field{ ... }, books: field{ ... } }, ... } }.
```

Figure 9: Extracts of the generated dicts for the type and schema definitions of Figure 7

In SWI–Prolog without occurs check, a call `Person = object{ ... Person ... }` does not raise an error. Instead, a corresponding cyclic substitution is created. Later, we will select the types of sub fields of `Person` using the dot notation, e.g., `T = Person.fields.friends.kind.type`. Since only GraphQL's types can be cyclic but not the queries, this will not lead to non-termination.

The AST for the query document is of an equivalent form. In Figure 9 we omitted additional internal properties used for introspection. For example it is possible to give each type and field a textual description which can be accessed with the help of the built–in `__type` query.

5 Query Execution Using Type and Field Resolvers

In this section, we present a working example for *GraphQL.pl*, which illustrates the query execution and validation. For data storage, we use Prolog facts `person/6` and `book/3` as presented in Figure 10. Their data correspond to the type definitions of Figure 3. To return either all or only the person's favourite books based on the `favourite` argument we use an additional `Favs` data entry. The example can easily be adapted to request the data from external databases with the help of SQL or from text–based formats like XML and CSV.

PROLOG

```
% person(Id, Name, Age, Books, Favs, Friends)
person(1, 'Alice', 31, [1, 2], [2], [2]).
person(2, 'Bob', 42, [2], [2], [1, 3]).
person(3, 'H. Melville', 72, [1], [], []).
person(4, 'D. Defoe', 71, [], [], []).

% book(Id, Title, Authors)
book(1, 'Robinson Crusoe', [4]).
book(2, 'Moby-Dick', [3]).
```

Figure 10: Example fact base

In order to execute a query document, both the AST of the query and of the type system are traversed simultaneously in a top–down approach. Beginning with the root type `Query` specified in the GraphQL schema, the *GraphQL.pl* system searches for the requested fields in the type definitions. To get the appropriate value, for every type of the schema a `resolve/5` predicate has to be defined, which we call the *resolver*. It is used to generate the resulting dict for a specific type, which is the basis for the JSON document returned to the client. A resolver is of the following form:

```
resolve(+Id, +Parent, +Args, +AST, -Result)
```

It is called every time the *GraphQL.pl* system has to retrieve the value of a field in the query document. This `Result`, which must be returned as a dict, is made up based on the given context information: `Id` is a unique identifier which might be used to retrieve a single data set; `Args` is a dict with all the arguments specified for the current field and might be empty, i.e. `_{}`; `AST` provides the static type information of the currently examined GraphQL type. The `Parent` argument passes the calculated result of the superior document level. In this way it is possible to, e.g., create a particular resolver for the `books` field in the `Person` type of Figure 3, which has access to the calculated answer dict of the person. This technique is used to refine or complete the answer of the parent's level.

The resolver's arguments might not be bound on every predicate call, for example, there is no `Parent` information for the query's root type. In most cases no unique identifier `Id` is specified, instead the `Result` has to be shaped based on the `Parent` and `Args` context information.

In Figure 11 we define the resolvers for our usage example[9], `resolve_p/5`, `resolve_b/5`, and `resolve_p_b/5`. They are required to calculate the answer dicts based on the given fact base:

[9]To use the *GraphQL.pl* system, the user has to implement a resolver for every GraphQL type, so the system is able to retrieve the requested data according to these resolver predicates. Since the resolvers are explicitly assigned to the appropriate types later, the predicate's name is not significant. We choose to call them using the type's first letter, i.e. `resolve_p` is the resolver for the type `Person`.

```
resolve_p(Id, _Parent, Args, _AST, Result) :-                    PROLOG
    ( Name = Args.get(name)
    ; nonvar(Id) ),
    person(Id, Name, Age, Fri, All, Favs),
    Result = _{ name:Name, age:Age, friends:Fri, books:(All, Favs) }.

resolve_b(Id, _Parent, Args, _AST, Result) :-
    ( book(Id, Authors, Args.get(title))
    ; nonvar(Id), book(Id, Authors, Title) ),
    Result = _{ title:Title, authors:Authors }.

resolve_p_b(_Id, Parent, Args, _AST, Result) :-
    (All, Favs) = Parent.books,
    ( Args.get(favourite) = true ->
      Result = Favs
    ; Result = All ).
```

Figure 11: Definition of the resolver predicates

- `resolve_p/5` is the used resolver for the type `Person`. It collects the person's information by its name, provided in the `Args` dict. If otherwise no `name` argument is specified, i.e. `Args.get(name)` fails, the appropriate data entry is retrieved by the given identifier `Id`.
- `resolve_b/5` retrieves the data for the type `Book` and similarly depends either on the book's title or its identifier.
- In addition to `resolve_p/5` we define the predicate `resolve_p_b/5`, which is used to only determine the returned value for the `books` field of a person.

The result of a requested `Person` type depends on the value of the boolean flag `favourite` of the `books(favourite: Boolean)` selector in the query: if it is `true`, only the person's favourite books should be listed; otherwise all. Therefore we determine the value of the `books` field in two steps, using the predicates mentioned before:

1. When determining the field–value associations for the `Person` object type by calling the `resolve_p/5` predicate, the value for the `books` field is set to the tuple `(All,Favs)`.
2. Because the fields are resolved in a top–down approach, *GraphQL.pl* now takes into account possibly defined resolvers of the next level's fields. We use the `resolve_p_b/5` resolver as a refinement of the `books` field. It returns either the `All` or `Favs` tuple element provided in `Parent.books`, depending on the `favourite: Boolean` flag given as `Args.favourite` property.

As a last step for the *GraphQL.pl* server declaration, the resolvers have to be connected to the appropriate types and fields. Every type and field generated using *GraphQL.pl*'s quasi quotations has a special `resolve` key which is unbound at first. It can be instantiated using the dot notation introduced in SWI–Prolog 7 for dicts. In our example, the type resolver for `Person` is declared by `Person.resolve = resolve_p`, as presented in Figure 12. The resolver for the person's `books` field is declared by `Person.fields.books.resolve = resolve_p_b`.

In a real–world application, the queries are raised by a remote client. The GraphQL server gets the query as string, runs the execution and sends back the response as a JSON document. To complete our

```
:- use_module(library(graphql), [query/3]).                    PROLOG
?- Schema = {| schema ||
      type Query { ... }
      type Book { ... } |},
   Person = {| type || ... |},
   Person.resolve = resolve_p,
   Person.fields.books.resolve = resolve_p_b,
   Book.resolve = resolve_b,
   Query = {| query ||
      person(name: "Alice") {
         name, years: age
         books(favourite: true) { title, authors { name } } } |},
   graphql:query(Schema, Query, Result).
```

Figure 12: Declaration of the schema and the resolvers with following query execution

given example, we directly call the query using the predicate `query/3` of *GraphQL.pl*, where `Schema` and `Query` are dicts generated from the corresponding quasi quotations:

```
graphql:query(+Schema, +Query, -Result)
```

`query/3` creates a result according to the type system and the resolver predicates and returns it as a dict, which could be directly sent to the client as a JSON document. In Figure 12 we give an example on specifying and executing a query given as a quasi quotation. `Result` is a dict representing the JSON output already stated in Figure 2. Because dicts are the preferred format to specify JSON since SWI–Prolog 7, they can be, e.g., printed by using the built–in `json_write_dict/3`. An example result is[10]:

```
Result = _{ data: _{
   person: _{ name: 'Alice', years: 31,
      books: [ _{ title: 'Moby-Dick',
         authors: [ _{ name: 'H. Melville' } ] } ] } } }
```

6 Conclusions and Future Work

In this paper, we have introduced the language GraphQL, an application layer used for data queries and manipulations developed by Facebook, and presented a comparison to REST, a well–established architectural pattern for distributed systems. We have implemented a GraphQL server in SWI–Prolog 7, called *GraphQL.pl*. We used features added to SWI–Prolog very recently in version 7, resulting in short and very readable source code. The *GraphQL.pl* system is an example for these features to underline their need and production–ready state in a real application.

For being able to specify the query and type system in the same way it is used in the GraphQL specification, the text–based format has been implemented as an external domain–specific language in Prolog. Along with the used syntactic elements – quasi quotations, definite clause grammars and dicts –, this allows for the specification of a GraphQL server even for developers not yet familiar with Prolog. In

[10] It is worth noting that although the type system given in Figure 3 defined `books` as a list of `Book`, i.e. `book: [Book]`, this is neither expressed in the string representation of the query nor its AST. But because this information can be deducted from the type system, the correct result document is calculated and the query is valid and unambiguous.

the GraphQL reference implementation, similar syntactic elements have been used in JavaScript: tagged template strings to embed the type system DSL, and JSON for the returned objects.

As the *GraphQL.pl* system has been developed in a test–driven approach, it also provides a test framework with a large number of use–cases of the implemented domain–specific language. The entire implementation is available online at `https://github.com/fnogatz/GraphQL.pl`.

In the future, we intend to connect *GraphQL.pl* with more back–end technologies, e.g. SQL, XML and REST APIs. Because our deductive database system DDBASE [7] already supports various, heterogeneous data sources, we intend to use GraphQL as an additional unified query language. Using the data's meta information collected by DDBASE, one could target reasoning about the data's format in order to improve GraphQL's introspection system or automatically generate the type system.

Because – in contrast to a REST approach – GraphQL is based only on a single request–response cycle, all the fields requested by the client are known at query execution, we intend to incorporate optimising strategies in the resolvers. In this way, it would be possible to use the abstraction layer introduced by GraphQL for query optimisations for existing query languages like SQL, too.

Besides for data access, GraphQL specifies also the syntax for mutating data sets. The `mutation` operation is not implemented in *GraphQL.pl* so far. As a proof–of–concept, we intend to connect GraphQL mutations with FN–Query [6], which allows the data access and manipulation of semi–structured data in XML documents.

References

[1] Lee Byron (2015): *GraphQL: A data query language – Engineering Blog – Facebook Code*. Available at `https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/`.

[2] Facebook (2016): *GraphQL Specification*. Available at `http://facebook.github.io/graphql/`.

[3] Roy Thomas Fielding (2000): *Architectural styles and the design of network–based software architectures*. Ph.D. thesis, University of California, Irvine.

[4] Fernando C.N. Pereira & David H.D. Warren (1980): *Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks*. *Artificial intelligence* 13(3), pp. 231–278, doi:10.1016/0004-3702(80)90003-X.

[5] Christian Schneiker, Dietmar Seipel, Werner Wegstein & Klaus Prätor (2009): *Declarative Parsing and Annotation of Electronic Dictionaries*. In: *6th International Workshop on Natural Language Processing and Cognitive Science (NLPCS 2009)*, pp. 122–132.

[6] Dietmar Seipel (2002): *Processing XML–documents in Prolog*. In: *17th Workshop on Logic Programming (WLP 2002)*. Available at `https://go.uniwue.de/fnqmanualpdf`.

[7] Dietmar Seipel (2015): *Knowledge Engineering for Hybrid Deductive Databases*. In: *29nd Workshop on (Constraint) Logic Programming (WLP 2015)*, pp. 66–78. Available at `http://ddbase.de`.

[8] Jan Wielemaker (2003): *An Overview of the SWI–Prolog Programming Environment*. In: *3rd Workshop on Logic–based methods in Programming Environments (WLPE 2003)*, pp. 1–16.

[9] Jan Wielemaker (2014): *SWI-Prolog version 7 extensions*. In: *Workshop on Implementation of Constraint and Logic Programming Systems and Logic–based Methods in Programming Environments 2014 (CICLOPS–WLPE 2014)*, pp. 109–123.

[10] Jan Wielemaker & Nicos Angelopoulos (2012): *Syntactic integration of external languages in Prolog*. In: *ICLP Workshop on Logic–based methods in Programming Environments (WLPE 2012)*, pp. 40–50.

[11] Jan Wielemaker & Michael Hendricks (2013): *Why It's Nice to be Quoted: Quasiquoting for Prolog*. In: *23rd Workshop on Logic–based methods in Programming Environments (WLPE 2013)*. Available at `https://arxiv.org/abs/1308.3941`.