A Mechanized Formalization of GraphQL

Tomás Díaz
IMFD Chile
Santiago, Chile

Federico Olmedo
University of Chile & IMFD Chile
Computer Science Department (DCC)
Santiago, Chile

Éric Tanter
University of Chile & IMFD Chile
Computer Science Department (DCC)
Santiago, Chile

1 Introduction

GraphQL is a technology-agnostic framework that provides a common language to define interfaces to services' data and to query them. It has been mainly proposed as a new alternative to RESTful Web Services. After being designed and used internally in Facebook for three years, in 2015 they released a specification [3] and a reference implementation[1]. Since then, GraphQL has seen a huge increase in their extent, community and popularity, becoming an independent foundation[2] early on 2019 and being incorporated to the services of major firms such as Coursera, Github and Airbnb. Some of its strong appeals are that many REST requests can be replaced by a single GraphQL query and that queries follow a "what you ask is what you get" spirit. This means that, in contrast with REST-based services, one can be very precise with the data requested and the response will look very similar to the query.

As recently mentioned, GraphQL has a specification that describes its main components, the *Spec* for the remainder of the paper. This document includes definitions for the query language and validation processes, among other things. The specification actively undergoes revisions, with an open working group that meets monthly to discuss related issues and improvements. These include extending the language to support new features or fix possible ambiguities present in the document. This is because the document is written in natural language, i.e. plain English, and does not include a rigorous formalization of its inner mechanics and limitations.

Hartig and Pérez proposed the first (and so far only) formalization of GraphQL [5] and used it to prove complexity boundaries for GraphQL queries. We refer to it as *HP* for the remainder of the paper. These results are based on two major premises. The first one is that "*for every query φ that conforms to a schema $\mathcal{S}$, there exists a* non-redundant *query φ' in* ground-typed normal form *such that $\varphi \equiv \varphi$'*". The second one is that for queries that are *non-redundant* and in *ground-typed normal form*, it is possible to define a simplified version of the semantics which is equivalent to the original. For the former, they propose a set of equivalence rules to transform queries but they do not actually prove that their application yield a query in this particular form or that they preserve the query semantics. The latter is also exploited, without providing any correctness proof. Since both are fundamental for their complexity results, we believe they must be rigorously addressed.

On another note, we believe that GraphQL is still a very young and active technology which could greatly benefit by having its specification mechanically verified from its early stages. It has a very active and growing community, with many different implementations in different programming languages and technologies, and more importantly, with many open questions and issues. It currently has a reference implementation, written in Javascript, that could be improved by introducing a formally and mechanically verified one.

Given the previous factors, we implement *GraphCoQL*[3], which formalizes GraphQL in Coq. We believe that it can serve as a starting point towards fully formalizing GraphQL and extracting it to be its official reference implementation. Transformations over queries, such as *HP*'s normalization, can then be completely specified and proven correct, as well as possible extensions and optimizations made to the language and its algorithms. To address the trustworthiness of our implementation, GraphCoQL tries to match the *Spec*'s definitions whenever possible. This provides a component of trustworthiness given by an eyeball correspondence, following the examples of [1] and[2].

With respect to the semantics of GraphQL, we follow a mixed approach between the *Spec* and *HP*. The semantics are defined in a graph setting, as is in *HP*, but the algorithm can be traced more closely to the *Spec*'s. One of the biggest difference between both approaches (besides the graph model) is that the *Spec* performs a processing of queries during the evaluation, while *HP* performs a post-processing of the responses generated. We took the mixed approach, which brings out some benefits as well as some limitations, which we discuss further in a following section.

[1] https://github.com/graphql/graphql-js

[2] https://foundation.graphql.org/

[3] The "CoQ" part is pronounced as "Coq", not pronouncing the "Q" separately as in "GraphQL".

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Finally, regarding the development itself, we made heavy use of SSReflect and their mindset of using boolean reflection as much as possible. Also, the use of the *Equations*[4] library to define non-structural recursive functions is essential for our definitions. Other libraries, such as *Function* and *Program* did not provide sufficient tools to handle rewriting and inductive reasoning about our definitions, which *Equations* incredibly facilitates. *GraphCoQL* is not currently extracted to other languages but we believe that it should not be a difficult task, given the design decisions considered.

Contributions

The main contributions of this work are:

1. The first mechanized formalization of GraphQL, including the definition for schema's DSL, query definition, schema and query validation, and its semantics over a graph data model.
2. Detection and correction of unsound definitions in *HP*.
3. The implementation of a normalization function with proofs of its correctness and preservation of semantics. This is a result used by *HP* to prove complexity boundaries about GraphQL queries.
4. Proof of equivalence between the semantics and a simplified version. This is also an important result for posterior analysis made in *HP*.

Structure of this paper

We first begin by gently and briefly introducing GraphQL in Section 2, which we do by means of an example. Then, in Section 3, we describe the basic building blocks of our Coq formalization. This includes the definition of a GraphQL schema, the graph data model, queries and their semantics. Section 4 describes the normalization process and proofs of its correctness and preservation of semantics. We finalize that section with the definition of the simplified semantics, as described in *HP*, and a proof of equivalence between the semantics defined in Section 3 and the simplified one. In Section 5, we describe some of the work we did to validate our implementation and finally Section 6 and 7 we discuss related and future work. TD ►*include note on code as anonymous supplementary material*◄

2 A brief introduction to GraphQL

TD ►*Meant to rewrite it but time's up :(*◄

GraphQL is a framework that provides a common language to define the interface to a service's data and to query it. It provides a language to describe how the data is structured and how it can be queried. This is called the schema or type system of the service. The schema consists of types and their fields. Queries may only be performed over these types and their fields. The resolution of each field is defined by the implementors, since GraphQL is not tied to any particular technology.

```
interface Animal {
    name: String
    friends: [Animal]
}
type Dog implements Animal {
    name: String
    friends: [Animal]
    favoriteToy: Toy
}
type Pig implements Animal {
    name: String
    friends: [Animal]
    oink: Float
}
type Toy {
    chewiness: Float
}
enum Goodness { BESTBOI GOODBOI OKBOI BADBOI }
union SearchResult = Dog | Pig | Toy
type Query {
    goodboi(goodness: Goodness): Animal
    search(text: String): SearchResult
}
schema {
    query: Query
}
```

Figure 1. Example of GraphQL Schema.

In the rest of this section, we will introduce GraphQL by means of an example. We will recurrently come back to this example throughout the rest of the paper. TD ►*Maybe not if we don't have space lol*◄

GraphQL Schema

Let's picture ourselves having a database with information about dogs and pigs; the *GoodBois* database. We want to define an API so our frontend developers may get the information and display it in our website. Our first step is then to describe how the data is structured and how it may be queried. This is done by means of the schema, which represents the type system of our GraphQL service.

Figure 1 depicts our type system. We define an interface for animals and two types implementing it; `Dog` and `Pig`. We know that animals have other animal friends, so we define the field `friends` whose return type is a list of other animals. We can also define enumeration types, which contain scalar values such as `GOODBOI`, and union types containing other object types. Finally, we have to define a `Query` type, which represents the entry point to our service's data. Any query that our frontend developers may do must begin by accessing this type's fields.

This is all it takes to describe our data and how our developers can query it. It describes exactly the data they can access and which are the entry points to it. However, each

[4] http://mattam82.github.io/Coq-Equations/

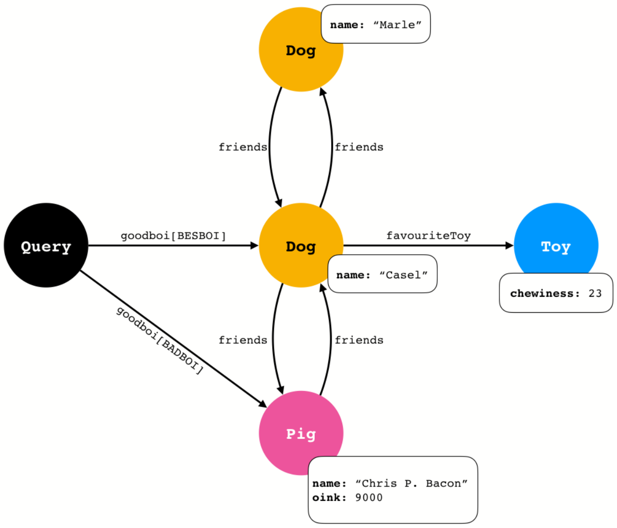

Figure 2. Example of GraphQL graph ET ►*favourite → favorite*◄

field has to somehow connected to actual data. When a developer requests the field `chewiness` we have to actually get that information from somewhere.

Graph Data model

Since GraphQL does not impose a particular technology or data model, it is not simple to reason about queries and their semantics. It is the job of the service's implementor to define how each field of a given type is resolved.

In our scenario, our data will be stored in a graph. Figure 2 illustrates our service's graph database. There is a root node from which every query must begin. This root node represents the `Query` type described in the schema. We also see that each node has a type, such as `Dog` or `Toy`, and properties such as their names. Each edge is also labeled with a name as defined in the schema. For instance, the edge connecting the dog named "Casel" is labeled `favoriteToy`, as declared in the type `Dog` in the schema.

Finally, now that we have defined our type system and data, our developers can proceed to query it.

GraphQL Query and Response

As previously mentioned, the queries we perform over our system must be over the types and fields defined in the schema. Every query must start by requesting information from the `Query` type. That means that, in our setting, queries must all start with the `goodness` or `search` fields.

In figure 3, we can see a query where we asking for all the friends of the `BESTBOI` in our system. For each friend we ask for their `name`. The query can be further specified, using fragments, and say that for the `Dog` friends we want to know their toy's `chewiness` and for the `Pig` friends, their `oink` level. We rename this last selection to `loudness`. As we can see from this example, queries in GraphQL have a tree structure similar to JSON.

If we evaluate this query in the graph depicted in 2, we would get the response shown in figure 3. This response was obtained by navigating the graph and collecting the information contained in each of the relevant nodes. It is easy to see that the response has a structure very similar to the query's.

If we wanted to ask another the same query but now without the friends' names, we would only have to remove the `name` field and *voilà*, that's it. We use the same endpoint as before and the GraphQL service handles the resolution of our fields.

With this we conclude our brief introduction to GraphQL and we can now move onto the formalization. We will come back to this example throughout the rest of the paper, illustrating how it can be replicated in our system.

The key points to take from our example are ...

3 GraphCoQL

In this section we describe our formalization of GraphQL in Coq. We start by defining a schema and its properties, then the graph data model and finally we review queries and their semantics. The definitions are as close as possible with respect to the *Spec*. This eyeball correspondence between the english-written definitions and the code gives a first level of trust that our formalization is correct, following the examples of X, Y and Z. Whenever there is a mismatch we point it out and explain the reasoning behind each decision.

TD ►*mention that we want to correlate to the spec and eyeball correspondence when possible.*◄

TD ►*The definitions consist of around 3700 loc and 1400 of lemmas*◄ .

3.1 GraphQL Schema

The GraphQL schema is pretty straightforward to define from the grammar of the *Spec*. It consists of a collection of type definitions and a root query operation type. There is, however, a slight ambiguity when the *Spec* refers to the schema, as it is described as being "*defined in terms of the types and directives it supports as well as the root operation types for each kind of operation*"[5]. It then proceeds to define a structure called `schema` containing only the root operation types (query, mutation and subscription) and *separately* it defines the type definitions, as well as the directives. The previously quoted definition actually matches the *Type System* structure[6]. Our formalization follows the latter but rename it to schema to also match the quoted description.

```
Record graphQLSchema := GraphQLSchema {
    query_type : Name;
    type_definitions : seq TypeDefinition
}.
```

[5] https://graphql.github.io/graphql-spec/June2018/#sec-Schema

[6] https://graphql.github.io/graphql-spec/June2018/#TypeSystemDefinition

```
query {
    goodboi(goodnessLevel: BESTBOI) {
        name
        friends {
            name
            ... on Dog {
                favoriteToy {
                    chewiness
                }
            }
            ... on Pig {
                loudness:oink
            }
        }
    }
}
```

```
{
    "goodboi": {
        "name":"Casel",
        "friends":[
            {
                "name": "Marle",
                "favoriteToy": {
                    "chewiness": 23
                }
            },
            {
                "name": "Chris P. Bacon",
                "loudness": 9000
            }
        ]
    }
}
```

Figure 3. Example of GraphQL query (left) and its response (right).

Similarly, for type definitions we follow the grammar as specified in the *Spec*. Figure 4 shows the grammar and the corresponding implementation in Coq. As can be seen from the figure, our implementation looses information about non-emptiness of fields, union and enum members. We push this validation to a posterior predicate, as well as the discussion about the reasons behind this decision, to the following paragraphs.

Although the definitions are straightforward, both the *Spec*'s grammar and the Coq implementation allow building invalid schemas. For instance, it is possible to build an Object that implements scalar types or use a nonexistent type as the query type. To this end, the *Spec* includes validation rules scattered throughout the document[7]. In *GraphCoQL*, we summarize these rules into predicates and refer to it as the *well-formedness* property of a GraphQL schema. *HP* refers to this property as the *consistency* of the schema, to which we will refer briefly in a following paragraph.

Definition 3.1. A GraphQL schema is *well-formed* if it satisfies the following conditions:

- Its root query type is defined and is an Object type.
- There are no duplicated type names.
- Every type definition is *well-formed*.

The implementation in Coq is described by the following boolean predicate. As indicated in the introduction of this paper, we try to use boolean reflection as much as possible, following the SSReflect mindset.

```
Definition is_a_wf_schema (s : graphQLSchema) : bool :=
      is_object_type s s.(query_type) &&
      uniq s.(schema_names) &&
      all is_wf_type_def s.(type_definitions).
```

Due to space constraints, we omit the definition of well-formedness for type definitions. The complete definitions can be found in the file `SchemaWellFormedness.v`. We will, though, resume the discussion about non-emptiness of fields, union and enum members, which are included in the predicate. The main reason behind this decision is that, even though the *Spec* embeds this information in the grammar, it still includes it in their validation rules later on. We believe that it is simpler to use common lists instead of defining new structures or using dependent types, from an implementation point of view, while still preserving the correspondence to the algorithmic description given by the *Spec*. TD ►*Not sure if correctly worded... but it was just simpler to use lists. A non-empty list structure required coercions to lists and then redefining some lemmas and things. Or using dependent types (sigma type) adds complexity when proving and defining things (at least that was the case for me)*◄

Regarding *HP*'s consistency property, they embed many properties in their structures, such as uniqueness of types given by using sets. They include an additional check on objects implementing interfaces, where they validate that fields are properly implemented. The definition given is not complete due to missing validation on arguments, but a corrected version is included in [4].

With the well-formedness property, we proceed to define a structure that encapsulates this notion, by passing both a schema and a proof of its validity.

```
Record wfGraphQLSchema := WFGraphQLSchema {
    schema : graphQLSchema;
    _ : schema.(is_a_wf_schema);
    is_a_valid_value : type -> Vals -> bool;
}.
```

It is immediate that this structure requires an additional `is_a_valid_value` predicate, which receives an element of `type` and a value of type `Vals`. This predicate is necessary to establish when a value used in a query or in the graph actually matches the scalar type expected by the schema. For instance, if an argument requires a `Float` value, then the actual value passed to the query must be something that represents a

[7] Most can be found in the **Type Validation** subsection of each type described in https://graphql.github.io/graphql-spec/draft/#sec-Type-System.

double-precision fractional value[8]. This predicate validates that this is satisfied.

Finally, having defined the GraphQL schemas, we can move onto defining the data model used when evaluating queries.

3.2 GraphQL Data model

GraphQL is not tied to any particular database technology and implementation. When resolving fields in a query, GraphQL assumes the existence of *resolvers*. These are internal functions defined by the user implementing a GraphQL service. They are not tied to any particular data model and the only requirement is that they must adhere to the schema. Whether they access a database, return static values or even modify existing data, is up to the user[9]. This makes reasoning about the semantics hard.

We choose to follow *HP*'s approach and define the underlying data model as a graph over which queries are evaluated. With this model, the unspecified resolvers can be instantiated to concrete definitions which allow reasoning over them. The semantics are then described as being implemented over a graph setting. Although this provides benefits when reasoning about the semantics, it also comes with some potentially severe limitations over the completness of the possible results generated. TD *►They may not actually be limitations with the model, but there are open questions on how to model some things.◄* We cover these limitations more thoroughly in a following paragraph and in Section 3.4. It is worth mentioning that the limitations of this model are not described nor discussed in *HP*.

Informally, a GraphQL graph is a directed property graph, with labeled edges and typed nodes. The graph describes entities with their types and properties, as well as the relationship between them. This means that every node has properties (key-value pairs) and a type. Also, every label in an edge describes the relation between two nodes. Finally, every property or label may also contain a list of arguments (key-value pairs).

We consider the type *Vals*, representing the values associated to properties or used for arguments. A value in *Vals* may be a single scalar value or a list of values.

Definition 3.2. A GraphQL graph over *Vals*is defined by the following elements:

- A root node.
- A collection of edges of the form $(u, \texttt{f}[\alpha], v)$, where u, v are nodes and $\texttt{f}[\alpha]$ is a label with arguments (key-value pairs).

This is defined with the following structures in Coq.

```
Record fld := Field {
                label : string;
                args : seq (string * Vals)
              }.

Record node := Node {
                 ntype : Name;
                 nprops : seq (fld * Vals)
               }.

Record graphQLGraph := GraphQLGraph {
                        root : node;
                        E : seq (node * fld * node)
                      }.
```

TD *►probably rewrite this paragraph...◄* Our definition is in essence the same as in *HP* but differs greatly in implementation. *HP* defines a GraphQL graph in a more "centralized" manner. For instance, nodes and field names are defined by sets. Node types are defined by a single function which receives a node identifier and gives its type. Properties are also defined by a single function which receives a node identifier and a field name with arguments. Contrarily, our approach attempts to recreate the structures individually. For instance, a node contains all the information pertaining to itself; its type and its properties. We believe this is a more natural approach to defining the graph from an engineering point of view.

The definition of graph is completely independent of any GraphQL schema, so we need a way to relate the data to the type system. We implement the notion of *conformance* of a graph as partially described by *HP*. This notion is, in essence, a well-formedness property for graphs with respect to a given schema. At the moment of development, there was no complete definition of conformance given by *HP*. However, in a recent work by Hartig and Hidder's [4], they give a complete definition and extend it. Their approach uses a similar "decentralized" idea to define graphs. Their definitions capture more features than we currently implement, such as directives and non-null types. TD *►And variables if I'm correct - should check◄*

Definition 3.3. A GraphQL graph *conforms* to a schema $\mathcal{S}$ if it satisfies the following conditions:

- The root node's type is equal to the query type.
- Every edge *conforms* to $\mathcal{S}$.
- Every node *conforms* to $\mathcal{S}$.

This is captured in the following predicate in Coq.

```
Definition is_a_conforming_graph
        (s : wfGraphQLSchema)
        (graph : graphQLGraph) : bool :=

        root_type_conforms s g.(root) &&
        edges_conform s g &&
        nodes_conform s g.(nodes).
```

Similarly to GraphQL schemas, we define a structure that encapsulates the notion of a *conformed* graph. It contains a graph and a proof of its *conformance* to a particular schema.

[8] The *Spec* declares a set of minimal scalar values and how they should be represented, such as floating-point values adhering to IEEE 754. We do not include this base restrictions but leave it open to implementation.

[9] The *Spec* states that these "*must always be side effect-free and idempotent*" but the definition of a resolver does not actually impose these restrictions.

⟨*TypeDefinition*⟩ ::= **scalar** ⟨*name*⟩
  | **type** ⟨*name*⟩ **implements** ⟨*name*⟩* **{** ⟨*Field*⟩+ **}**
  | **interface** ⟨*name*⟩ **{** ⟨*Field*⟩+ **}**
  | **union** ⟨*name*⟩ = ⟨*name*⟩ | ⟨*name*⟩*
  | **enum** ⟨*name*⟩ **{** ⟨*name*⟩+ **}**

⟨*Field*⟩ ::= ⟨*name*⟩ **(** ⟨*Arg*⟩* **)** **:** ⟨*type*⟩

⟨*Arg*⟩ ::= ⟨*name*⟩ **:** ⟨*type*⟩

⟨*type*⟩ ::= name
  | **[** ⟨*type*⟩ **]**

(a) Grammar of GraphQL types

```
Inductive TypeDefinition : Type :=
| ScalarTypeDefinition (name : Name)

| ObjectTypeDefinition (name : Name)
                       (interfaces : seq Name)
                       (fields : seq FieldDefinition)

| InterfaceTypeDefinition (name : Name)
                          (fields : seq FieldDefinition)

| UnionTypeDefinition (name : Name)
                      (members : seq Name)

| EnumTypeDefinition (name : Name)
                     (members : seq EnumValue).

Inductive type : Type :=
| NamedType : Name -> type
| ListType : type -> type.
```

(b) Implementation in Coq TD ►*Should include fields and arguments?*◄

Figure 4. Definition of GraphQL types.

```
Record conformedGraph (s : wfGraphQLSchema) :=
            ConformedGraph {
                graph : graphQLGraph;
                _ : is_a_conforming_graph s graph
            }.
```

Due to space limitations, we omit a detailed review of *conformance* of nodes and edges. The complete definitions can be found in the file GraphConformance.v.

Finally, we partially retake the discussion on the limitations of this model. These have consequences on the semantics of GraphQL queries, so we delay some of it to the corresponding section. The main issue is that there is no proper accounting with respect to list types containing other list types (with any nesting depth). The different features that compose a GraphQL schema can be translated to a graph somehow. For instance, a field is either a property or the label of an edge, while its return type can be associated to a target node in an edge. However, when it comes to list types it is not clear what they represent in a graph. Let us illustrate this with an example.

A service may declare the field friends:[Human] in a given type, representing the list of friends. In a graph this can be pictured as having a node with multiple outgoing edges labeled friends, reaching other nodes of type Human. It is possible to then extend the service by including a new field friendsByName:[[Human]], in which one can request a list of friends but grouped by their names. At the moment neither our implementation, *HP* nor [4] properly handle this situation. The open question is what does this represent in the graph? These should be outgoing edges similarly to the previous case but, what should the target nodes be? Should these be intermediate blank nodes? Is every edge labeled or only the last one that reaches a node with type Human? What happens if we increase the nesting? Since the information is ultimately collected from “concrete” nodes, should the graph be kept the same but introduce *formatter* functions to match the schema?

These questions and more are not addressed nor discussed in *HP* and it is actually more restrictive than expected, by not allowing nested lists for scalar values (in nodes’s properties). Meanwhile, our approach and the one used in [4] allow any list type at the property level but simply ignore any possible nesting when the list type refers to neighboring nodes (composite types), as in the example above. In the case of [4], they do not address nor discuss these questions. This choice of modeling has some consequences when defining the semantics of GraphQL queries, because the possible results generated are restricted to a smaller subset. It is not clear what the proper way is to handle this issue but more is explored in Section 3.4. We also address the *Spec*’s semantics and how this is managed.

With both the schema and the underlying data model we can proceed to define GraphQL queries and their semantics.

3.3 GraphQL Query

As we mentioned in section 2, GraphQL queries are selections over types and fields defined in the schema. A GraphQL query can be seen as a tree structure where leaf nodes are selections of fields with a scalar return type. An inner node can be a selection on fields with an object or abstract return type. Inline fragments that condition when its subqueries are evaluated can also be seen as inner nodes. For instance, the query in Figure 3 can be depicted as the tree in Figure 5.

Similar to the schema definition, we try to follow the *Spec*’s grammar as closely as possible. The grammar and implementation can be seen in Figure 6. There is a lost of information regarding non-emptiness of subqueries, as seen by rule 2 and

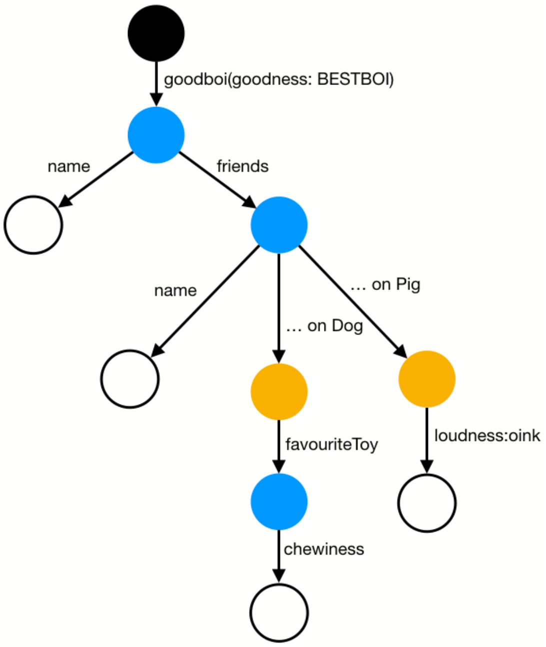

Figure 5. GraphQL query as a tree.

the constructor `NestedField`. The reasoning behind this decision is very similar to the one used when implementing type definitions, which is described in Section 3.1.

Both the *Spec* and our formalization differ from *HP* when defining queries. The main difference is that *HP* include an additional rule for lists of queries. Their grammar includes a production rule for lists of queries which is at the same level of the other rules. The main issue we found with this approach is that it allows building arbitrary trees instead of just a list of queries. These trees can be flattened to recover the list structure but this represents additional effort when defining functions and reasoning over queries. We believe this is assumed by *HP* but not explicitly mentioned otherwise.

As in the case of well-formedness of schemas or conformance of graphs, queries must go through a validation process. We define the *conformance* of queries based on validation rules scattered throughout the *Validation* section of the *Spec*[10].

Before defining the validation process, it is very important to address the notion of *type in context* where selections are used. This notion is necessary to validate queries and when transforming queries, as described in Section 4. The type in context is the type over which someone might be requesting information on its fields. For instance, in the following example the field selection `goodboi` is used in the context of the `Query` type. However, the type in the case of the field `name` is not entirely clear. In one case, the type in context is `Dog`, while in the other the field is used in the context of the `Pig` type.

```
query {
  goodboi {
    ... on Dog {
      name
    }
    ... on Pig {
      name
    }
  }
}
```

The importance of this type in context is that fields or inline fragments might be valid in certain cases but not in others. Similarly, a field may have a particular return type in one case and a different one in another type, like in the following example. Both types have an `age` field, but in one case it returns an integer value while in the other a floating point value. If that field is encountered in a query, it is necessary to know to which type it is being requested.

```
type Human {
  age: Int
}

type Martian {
  age: Float
}
```

Definition 3.4. A GraphQL query φ *conforms* to a schema $\mathcal{S}$ if it satisfies the following conditions:

- Selections in φ are consistent.
- Field merging between fields is possible.
- Fields with same response name have compatible response shapes.

The definition in *GraphCoQL* is given by the following code. Due to space constraints, we do not include the complete definitions but they can be found in the file `QueryConformance.v`.

```
Definition queries_conform (type_in_scope : Name)
                           (queries : seq Query) : bool :=
       all (is_consistent type_in_scope) queries &&
       is_field_merging_possible type_in_scope queries &&
       have_compatible_response_shapes
           [seq (type_in_scope, q) | q <- queries].
```

As described earlier, this rules are mostly a condensation of a set of validation rules defined in the *Spec*. The first one refers to whether a selection holds by itself. It includes checks such as: if query is over a field, then that field must be defined in the type in context and its arguments are defined in the given field. Similarly, if a selection is an inline fragment, then the type condition has to be valid with respect to the type in context.

The second and third predicates are defined as a single validation rule in the *Spec*[11]. We split them into two separate predicates because there is a chance for optimization.

[10] https://graphql.github.io/graphql-spec/June2018/#sec-Validation

[11] https://graphql.github.io/graphql-spec/June2018/#sec-Field-Selection-Merging

$$
\begin{array}{lll}
\langle \mathit{Query} \rangle & ::= & \langle \mathit{name} \rangle \ \textbf{(}\ \langle \mathit{Arg} \rangle^{*}\ \textbf{)} \\
 & | & \langle \mathit{alias} \rangle\ \textbf{:}\ \langle \mathit{name} \rangle \ \textbf{(}\ \langle \mathit{Arg} \rangle^{*}\ \textbf{)} \\
 & | & \langle \mathit{name} \rangle \ \textbf{(}\ \langle \mathit{Arg} \rangle^{*}\ \textbf{)}\ \textbf{\{}\ \langle \mathit{Query} \rangle + \textbf{\}} \\
 & | & \langle \mathit{alias} \rangle\ \textbf{:}\ \langle \mathit{name} \rangle \ \textbf{(}\ \langle \mathit{Arg} \rangle^{*}\ \textbf{)}\ \textbf{\{}\ \langle \mathit{Query} \rangle + \textbf{\}} \\
 & | & \textbf{... on}\ \langle \mathit{name} \rangle\ \textbf{\{}\ \langle \mathit{Query} \rangle + \textbf{\}}
\end{array}
$$

$$
\langle \mathit{Arg} \rangle \ ::= \ \langle \mathit{name} \rangle\ \textbf{:}\ \langle \mathit{value} \rangle
$$

(a) Grammar of GraphQL queries

```
Inductive Query : Type :=
| SingleField (name : Name)
              (arguments : seq (Name * Vals))

| AliasedField (alias : Name)
               (name : Name)
               (arguments : seq (Name * Vals))

| NestedField (name : Name)
              (arguments : seq (Name * Vals))
              (subqueries : seq Query)

| NestedAliasedField (alias : Name)
                     (name : Name)
                     (arguments : seq (Name * Vals))
                     (subqueries : seq Query)

| InlineFragment (type_condition : Name)
                 (subqueries : seq Query).
```

(b) Implementation in Coq

Figure 6. Definition of GraphQL queries.

We noticed that the original definition includes redundant recursive calls which may result in increased computational time. At the time of writing this paper, a new algorithm was proposed by a team at XING[12] that also addresses this very same issue and is described in [6]. They follow an approach using sets and provide a much more elaborate analysis of execution times than us. Comparing both approaches and analyzing execution times could be an interesting venue to explore.

During development, we also noticed that the *Spec*'s rule is too conservative and may consider valid queries as invalid. In a nutshell, the *Spec* allows defining fragments that are never evaluated. The issue is that the validation rule can then consider that subqueries in these fragments are invalid, even though they are never evaluated, rendering the whole query invalid[13]. The definition of the second predicate attempts to remove this conservativeness but we have not proved it. For the third predicate, we still have some conservative checks. Section **??** delves a little deeper into this issue.

Finally, with these definitions we can build queries in a GraphQL service. Examples may be found in the files `SpecExamples.v` and `HPExample.v`. From now on, we will assume that queries conform to a given schema. We can then move onto their semantics.

3.4 Semantics

In this section we describe the semantics of GraphQL queries. We begin by briefly examining the responses generated by executing queries. Then we give an informal description of the semantics, followed by the formal definition. We finish by discussing some implementation choices and comparison with the *Spec* and *HP*.

The *Spec* describes responses as a map. Our implementation differs slightly, modeling them as a tree structure, similar to JSON. We choose this structure to preserve similarity to queries and because it is simpler to preserve order of the responses. The *Spec* does not impose an ordering of responses, although encourages it[14]. We believe that preserving the order is one of the selling points for GraphQL (queries and their responses are very similar and easy to read). Our approach has two main disadvantages: uniqueness of response names and cost of access. Since we use lists instead of maps, we can encounter duplicated names and accessing a value has a linear cost given by the lists size, instead of the constant access obtainable with a map. We still argue that the simplicity to obtain order is worth it. We do include a proof that the results obtained with the semantics have unique names. Finally, we use option types to represent null values in the leaves of the response tree.

```
Inductive ResponseNode (A : Type) : Type :=
| Leaf : A -> ResponseNode
| Object : seq (Name * ResponseNode) -> ResponseNode
| Array : seq ResponseNode -> ResponseNode.

Definition GraphQLResponse (Vals: eqType) :=
    seq (Name * (@ResponseNode (option Vals))).
```

Moving onto the semantics of GraphQL queries. As we described in Section 3.2, the underlying data model is a graph, therefore the semantics are instantiated to this setting. In a following paragraph we briefly explore an alternative that is closer to the *Spec*, in the sense that it can be detached from a particular data model. In our setting a query then represents

[12] https://www.xing.com/

[13] An example query can be seen in the following link: https://tinyurl.com/y3hz5vgv.

[14] https://graphql.github.io/graphql-spec/June2018/#sec-Serialized-Map-Ordering

a navigation over a graph. At top level, a query starts from the root node and then moves around its edges and nodes, collecting data along the way. In this sense:

- A field selection represents either accessing a node's property or traversing an edge to a neighboring node. On the neighboring nodes we recursively evaluate subqueries.
- An inline fragment conditions whether using a node to access its properties or to traverse to other nodes.

Figure 7 shows the formal definition of the semantics. It displays the cases where a field selection is accessing a node's property, when it is navigating to other nodes and when it is evaluating an inline fragment. Aliased fields are omitted for brevity but the complete definition can be explored in the file `QuerySemantics.v`.

The main difference with respect to *HP* and the main similarity to the *Spec* is that we perform a collection of fields at the query level, whereas *HP* performs a post-processing of responses. The main reasons are similarity to the *Spec* and difficulty in reasoning with *HP*'s approach, which we explore in more depth in Section **??**.

We finish this section by addressing two major aspects about our formalization; completeness and errors.

The first one was briefly mentioned in Section 3.2, when discussing the limitations and open questions regarding the graph model. These translate in the fact that we currently do not produce list results with nested lists of objects. For instance, the field `friendsByName:[[Human]]` is treated as if it were defined as `friendsByName:[Human]` and the results match the latter format. Otherwise, there is no restriction in the case of nested lists for scalar values. In *HP*, there is no possibility to produce nested lists for either scalar or object values[15] and there is no mention of this restriction.

Regarding error handling, we currently do not implement it. Errors may have two main sources; validation errors and execution errors. TD ►*Not sure how to write this*◄

TD ►*I am also missing functors and how the spec "should" be defined.*◄

This concludes the base formalization of GraphQL schemas, graph data model, and queries and their semantics. Using this basic structures we can start defining query transformations and prove some properties about them.

4 Query Transformation: Normalization

As a first case study for query transformation, we decided to tackle the normalization process used in HP. This is a fundamental process on which they base their results on complexity for GraphQL queries. One of their base statements is that every query can be normalized and the resulting query is semantically equivalent. They provide equivalence rules to transform the queries but do not provide the full proof of correctness for them.

In this section we review the property of being in *normal form*, as well as the normalization procedure we implemented. We then prove that our normalization procedure is correct and that it preserves the semantics of the original queries, as postulated by HP. In the end, we briefly review some differences and observations with respect to HP's definitions.

It is worth mentioning that the bigger part of our development was dedicated to defining and establishing the correctness of this normalization procedure. In terms of code it required around 350 lines of code for the definitions and around 1200 of lemmas and tactics.

4.1 Normal form

The notion of *normal form* is defined by the conjunction of two other properties; being *grounded*[16] and being *non-redundant*.

Groundness

Informally, the *groundness* property refers to whether queries are completely specified down to the objects and scalar types. The main idea is that if we are querying an Object type then we should only ask for its fields, while if we are querying an Abstract type (Interface or Union), then our queries should be specified down to their object subtypes. In the former case, it does not make sense to use fragments to further specify our query (we cannot be more specific when querying an object), while in the latter we want to use fragments to clearly state what we want from each concrete subtype.

Definition 4.1. A GraphQL query φ is *grounded* if it satisfies the following conditions, where `ty` is the type in scope.
If `ty` is an Object type, then φ contains only fields.
If `ty` is an Abstract type (Interface or Union), then φ contains only inline fragments. The type condition on these fragments must be Object types.
Subqueries of φ are *grounded* wrt. to the field's return type or the fragments type condition.

This definition differs slightly from the one given by HP, because we use information on the type in context where queries might be defined. We prove that our definition still implies being in *ground-typed normal form*. We made this choice because we found that the notion given by HP was too general for our implementation. This came up during the proofs of correctness for our normalization procedure. We will not go into much detail due to space constraints.

```
Variable (s : wfGraphQLSchema).

Lemma are_grounded_in_ground_typed_nf (type_in_scope : Name)
                                      (queries : seq Query) :
        are_grounded s ty queries ->
        are_in_ground_typed_nf s queries.
```

[15] The grammar itself does not permit it.

[16] HP refers to it as *ground-typed normal form*. We believe this name is a bit misleading.

$$[\![\cdot]\!]^u_G = [\cdot] \tag{1}$$

$$[\![\mathtt{f[}\alpha\mathtt{]} :: \overline{\varphi}]\!]^u_G = \begin{cases} \mathtt{f:v} :: [\![\mathit{filter}_{\mathtt{f}}(\overline{\varphi})]\!]^u_G & u.\mathit{property}(\mathtt{f[}\alpha\mathtt{]}) = \mathtt{v} \\ \mathtt{f:null} :: [\![\mathit{filter}_{\mathtt{f}}(\overline{\varphi})]\!]^u_G & \sim \end{cases} \tag{2}$$

$$[\![\mathtt{f[}\alpha\mathtt{]\{}\overline{\beta}\mathtt{\}} :: \overline{\varphi}]\!]^u_G = \begin{cases} \mathtt{f:[}\mathit{map}(\lambda\ v_i \Rightarrow [\![\overline{\beta} +\!\!+ \mathit{merge}(\mathit{collect}_{\mathtt{f}}(\overline{\varphi}))]\!]^{v_i}_G)\ \mathit{neighbors}(u)\mathtt{]} :: [\![\mathit{filter}_f(\overline{\varphi})]\!]^u_G & \mathit{type}(f) \in L_t \text{and} \{v_1, \ldots, v_k\} = \{ \\ (f : \{[\![\overline{\beta}]\!]^v_G\}) :: [\![\mathit{filter}_f(\overline{\varphi})]\!]^u_G & \mathit{type}(f) \notin L_t \text{and} (u, f[\alpha], v) \in E \\ (f : \mathit{null}) :: [\![\mathit{filter}_f(\overline{\varphi})]\!]^u_G & \mathit{type}(f) \notin L_t \text{and there is no } v \text{ s.t.} \end{cases} \tag{3}$$

$$[\![\ldots\ \mathtt{on}\ \mathtt{t\{}\overline{\beta}\mathtt{\}} :: \overline{\varphi}]\!]^u_G = \begin{cases} [\![\overline{\beta} +\!\!+ \overline{\varphi}]\!]^u_G & \mathit{does\_fragment\_type\_apply}_{\mathtt{t}}(u.\mathit{type}) = \mathtt{true} \\ [\![\overline{\varphi}]\!]^u_G & \sim \end{cases} \tag{4}$$

Figure 7. Semantics for GraphQL queries. TD ►*This looks bad but I don’t know how to format it :/*◄

Non-redundancy

Informally, the notion of non-redundancy refers to whether there might queries that may produce repeated results.

Definition 4.2. A GraphQL query φ is *non-redundant* if it satisfies the following conditions.

- There is at most one field selection with a given response name. This includes visiting inline fragments.
- There is at most one inline fragment with a given type condition. This does not include visiting other inline fragments.
- Subqueries are *non-redundant*.

This definition is slightly different from the one given by HP but we leave this discussion to section 4.5. TD ►*Not much more to add...*◄

4.2 Normalization procedure

The normalization procedure is very similar to how the semantics are defined. In a sense, it is essentially a static evaluation of the queries, using only information about the type in context where the queries might be defined.

The process consists of two main parts, which deal with the two aforementioned properties. It first assumes that the type in context is an Object type[17]. We describe them separately but occur simultaneously.

- Merging: Whenever a field is encountered, the procedure tries to find all fields with the same response name and merge their subqueries. It then proceeds to remove them from the list to ensure *non-redundancy*. Comparing it to the the semantics, this is equivalent to the case when we evaluate a field and collect similar ones.
- Grounding: Since it is assumed that the type in context is an Object type, it will try to transform the query such that there are only fields left. This means it will try to get rid of inline fragments and lift their subqueries as much as possible. Much like if we were standing on a node in the graph, we only evaluate fragments and subqueries that make sense for that node’s type (which is an Object type). In the case of fields, it will first check on its return type. If it is an abstract type, then it will create a cover of all possible concrete subtypes of the abstract type, by wrapping the subqueries with inline fragments. Otherwise, it will proceed recursively. Once again, this is like finding the neighbors of a node. Since we don’t know their types, we anticipate all possible cases.

With this definition, we proceed to define a second one, which makes no assumption on the type in context. This procedure only checks what kind of type it receives and either pipes the job to the previous one, or covers the queries with the possible concrete subtypes (and then pipes the work to the previous definition).

```
Definition normalize_queries (type_in_scope : Name)
                             (queries : seq Query) :
                                         seq Query :=
  if is_object_type s type_in_scope then
     normalize type_in_scope queries
  else
     [seq on t { normalize t queries } |
         t <- get_possible_types s type_in_scope].
```

With this definition we can the move onto proving their correctness and that the semantics are preserved for the source query.

4.3 Proofs of correctness and preservation

The previous definitions do not ensure that our resulting queries are in normal form, so we must prove them correct.

[17] If we lift this to the top level we will find the Query type, which is an Object type.

We can then prove that the source queries are semantically equivalent to their normalized versions. This satisfies the statement by HP

First, we prove that the procedure delivers *grounded* queries. By transitivity we get that they are in *ground-typed normal form*.

```
Lemma normalize_are_grounded ty φ :
    is_object_type s ty ->
    are_grounded s ty (normalize s ty φ).

Lemma normalize_queries_are_grounded ty φ :
    are_grounded s ty (normalize_queries s ty φ).
```

Immediately afterwards we can prove that the resulting queries are indeed *non-redundant*.

```
Lemma normalize_are_non_redundant ty φ :
    is_object_type s ty ->
    are_non_redundant (normalize s ty φ).

Lemma normalize_queries_are_non_redundant ty φ :
    are_non_redundant (normalize_queries s ty φ).
```

Finally, we prove that the semantics are preserved for the resulting queries. First, we prove the case where we are normalizing the queries by the type of a node u and evaluating them on that same node. Pushing this to top level, we find ourselves evaluating queries on the root node which has type equal to the query type (given by *conformance* of the graph). We then extend this notion to normalization with any type `ty` but with the restriction that the node's type must be a subtype of `ty`. Once again, this is valid at top level over the root node. For nodes in between we know their types are subtypes of the field by which we reached them (given by *conformance* of the graph and its edges).

```
Lemma normalize_exec φ u :
    u \in g.(nodes) ->
    s, g ⊢ ⟦ normalize s u.(ntype) φ ⟧ in u with coerce =
    s, g ⊢ ⟦ φ ⟧ in u with coerce.

Theorem normalize_queries_exec ty φ u :
    u \in g.(nodes) ->
    u.(ntype) \in get_possible_types s ty ->
    s, g ⊢ ⟦ normalize_queries s ty φ ⟧ in u with coerce =
    s, g ⊢ ⟦ φ ⟧ in u with coerce.
```

Having proved this statements we can now define a simplified version of the semantics.

4.4 Simplified semantics

As proposed by HP, one of the main properties of queries in normal form is that they produce a unique response, without the need of any collecting and merging of fields. This allows defining a second evaluation function $\ll \varphi \gg_G$, similar to the one defined in 3.4 but without any filtering and collecting of fields.

We implemented this function and then proved that for queries in normal form, both $[\![\varphi]\!]_G$ and $\ll \varphi \gg_G$ produce the same response.

```
Theorem exec_equivalence u φ :
    are_in_ground_typed_nf s φ ->
    are_non_redundant φ ->
    s, g ⊢ ⟦ φ ⟧ in u with coerce =
    s, g ⊢ ≪ φ ≫ in u with coerce.
```

This concludes the normalization process and satisfy the requirements set by HP for their complexity results.

4.5 Discussion

There are some final notes we must address regarding some of the definitions. This includes some discoveries we made regarding HP and how we resolved them. In particular, we review the *non-redundancy* property and the equivalence rules they define.

For the former, we noticed that their definition is unsound TD ►*?*◄ , in the sense that there are queries that are considered *non-redundant* but they actually would produce redundant results. A simple example is the following valid query.

```
Theorem exec_equivalence u φ :
    are_in_ground_typed_nf s φ ->
    are_non_redundant φ ->
    s, g ⊢ ⟦ φ ⟧ in u with coerce =
    s, g ⊢ ≪ φ ≫ in u with coerce.
```

This is considered as *non-redundant* when, in fact, it would produce two repeated values. It is a very minor slip, which occurs because they only compare unaliased fields with unaliased fields and, respectively, aliased fields with aliased fields. They do not compare unaliased with aliased fields, which causes the problematic cases.

Regarding the equivalence rules, there are three elements we have to highlight. The first one is that rule number (2), which deals with merging of fields with subqueries, is correct but does not preserve ordering of the queries. While this is not a hard requirement, it is an important aspect in GraphQL evaluation. This is also important when comparing that the results are equivalent; Does order matter? Is it just its content?

The second aspect is about the elements they use TD ►*?*◄ in their rules. In some cases they use list of queries while in some other they define it over single queries, or sometimes mix them. While this is no big issue, it was a bit confusing when trying to implement their rules in Coq. TD ►*Not sure how to describe this, but the thing is their rules are a bit weird. They describe rules for individual selections, but there is no... "global" rewriting. I imagine this is "simpler" to understand with their semantics, because they do not modify the queries as they evaluate them (pushing everything to the responses), but it is still weird to define it as a procedure in Coq (or even as inductive relation).*◄

Finally, there is an implicit notion of type in context when they describe their rules TD ►*and maybe a missing rule?*◄ . This is crucial, because otherwise there are queries that cannot be normalized. For example, the following query cannot be normalized with the rules as they are.

```
Theorem exec_equivalence u φ :
    are_in_ground_typed_nf s φ ->
    are_non_redundant φ ->
    s, g ⊢ ⟦ φ ⟧ in u with coerce =
    s, g ⊢ ≪ φ ≫ in u with coerce.
```

However, if we include the type in context, which corresponds to Query in this case, we can do more. We can wrap all queries in an inline fragment with type condition Query. We can then use a mix of rules to obtain the normalized query.

TD ►*Not sure where to mention the whole process of doing this (since it took the most of our time). Things such as:*

- *Trying to implement HP's rules of equivalence.*
- *Trying to work on a subset of queries with no invalid fragments.*
- *Change/Discovery of their semantics and responses.*
- *Definition of normalization in two separate functions; one for grounding and one for removing redundancy.*
- *etc.*

◄

5 Implementation and Validation

LOC, files, man-month, major effort

Examples - Jorge's, Spec,

Most of the development time was spent in the definition and proofs of normalization. We initially worked on the semantics as specified by [5]

6 Related Work

Talk about recent work by Olaf about using the Schema DSL to define type systems for property graphs.

Work by Véronique.

Work by Dumbrava/Emilio.

There's some work by Christian Doczkal and Damien Pous about Graph Theory in Coq: Minors, Treewidth, and Isomorphisms (presented at coq workshop 19) that might be worth mentioning?

7 Future Work

Extraction.

Testing and comparing with ref implementation.

Automation of proofs

Extend to include more things (handle errors, handle variables, etc.)

Collab with GraphQL foundation/community

8 Conclusions

References

[1] Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., and Smith, G. A trusted mechanised javascript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014* (2014), pp. 87–100.

[2] Bodin, M., Diaz, T., and Tanter, É. A trustworthy mechanized formalization of R. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018, Boston, MA, USA, November 6, 2018* (2018), pp. 13–24.

[3] Foundation, G. Graphql specification. https://graphql.github.io/graphql-spec/.

[4] Hartig, O., and Hidders, J. Defining schemas for property graphs by using the graphql schema definition language. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (2019), ACM, p. 6.

[5] Hartig, O., and Pérez, J. Semantics and complexity of graphql. In *Proceedings of the 2018 World Wide Web Conference* (2018), International World Wide Web Conferences Steering Committee, pp. 1155–1164.

[6] XING. Graphql: Overlapping fields can be merged fast. https://tinyurl.com/y3wqmnrw, 2019. [Online; accessed 20-Sept-2019].