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 used internally in Facebook for three years, in 2015 they released a specification and a reference implementation. Since its release, GraphQL has seen a huge increase in popularity, with major firms such as Coursera, Github and Airbnb incorporating it to their services. Early on 2019, it became an independent foundation, separating itself from Facebook. 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 one can be very precise with the data requested and the response will look very similar to the query.

GraphQL has a specification that describes its main components. We will refer to it as the *Spec*throughout 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 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 and its semantics [1]. They then use it to prove some complexity boundaries for GraphQL queries. These results are based off two major statements. 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 equal to the original. For the former, they propose equivalence rules to transform queries but do not actually provide proof that the normalization is correct and that it preserves the semantics. The latter is also missing its proof. Since both are fundamental for their complexity results, we believe it is essential to tackle them.

On another note, we believe that GraphQL is still a very young and active technology which could benefit greatly by having its specification mechanically verified from its early stages. Its scope is not so vast that it cannot be formalized, and it is still growing and with open questions. It currently has a reference implementation[1], written in Javascript, which could be improved by introducing a formally verified one. We will refer to it as *GraphQLJs* throughout the document.

We therefore decided to implement *GraphCoQL*[2], which formalizes GraphQL and its semantics in Coq. Our intention is 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 to the language.

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 X, Y, Z. We also test our implementation with examples from the *Spec*but a more thorough comparison should be made against *GraphQLJs* and a bigger test suite.

With respect to the semantics itself, 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.

Finally, in regards to extraction and the code itself, GraphCoQL is not currently extracted to any language. However, we made heavy use of SSReflect and their mindset of using

[1] https://github.com/graphql/graphql-js
[2] The "CoQ" part is pronounced as "Coq", not pronouncing the "Q" separately as in "GraphQL".

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

boolean reflection as much as possible. We were first motivated to use it to define the data model and try to narrow our scope to finite types, as was used by (Veronique, Ev, Emilio, Dumbrava). In the end, we did not use any of it but the computational aspect of SSReflect was kept, as it facilitated developing the proofs. This same element is what makes us believe that extraction should not be hard. A final note on the implementation is the use of the *Equations*[3] library to define non-structural recursive functions. 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. We therefore found it crucial in our development.

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.

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.

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

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 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.

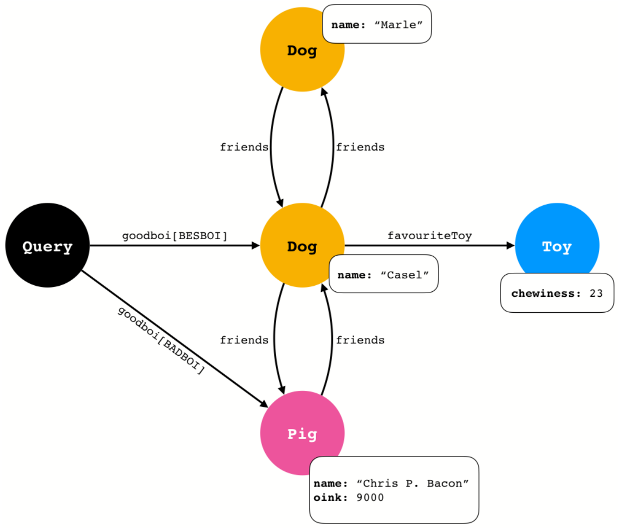

Figure 2. Example of GraphQL graph ET ►*favourite → favorite*◄

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 Syntax and Semantics

In this section we will describe our formalization of GraphQL in Coq. We will follow a similar structure as the previous section. We start by defining a schema and its properties, then the graph data model and finally we review queries and their semantics.

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 Spec. It consists of a collection of type definitions and a root query operation type. The Spec uses a structure called `schema` where it defines the root operation types (query, mutation and subscription) and *separately* the type definitions. We opted to define them all in a single structure.

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

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 in the figure, we tried to match the Spec's definition as much as possible. This eyeball correspondence gives us a degree of confidence about the implementation.

Although both definitions are straightforward, they allow building invalid schemas. For instance, it is possible to build an Object that implements scalar types, an Interface with no fields or use an nonexistent type as the query type. We therefore require a notion of *well-formedness* of a schema.

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*.

We can then proceed to define a structure that encapsulates this notion, by passing both a schema and a proof of its *well-formedness*. TD ►*The property of well-formedness is implemented as a boolean predicate.*◄

```
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).

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

As you may have noticed, this structure also requires an `is_a_valid_value` predicate, which receives a `type` and a value of `Vals`. This predicate is necessary to establish when a value used in a query actually matches the scalar type expected

```
{
    "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).

by the schema. For instance, if an argument requires an `Int` value, then the actual value passed on to the query must be something that looks like an integer. This predicate validates that this is satisfied.

Due to space constraints, we omit the definition of *well-formedness* for type definitions. This property includes things such as: interfaces and objects must declare at least one field, objects correctly implement their declared interfaces, union types are not empty and contain only object types, amongst others. These definitions are collected from the Spec TD ►*Scattered throughout the Spec**◄ .

Having defined our GraphQL schemas, we can move onto defining the graph data model used when evaluating queries.

3.2 GraphQL Data model

We chose to describe the underlying data model of a GraphQL service as a graph, in the same line as HP. Informally, a GraphQL graph is a directed property graph, with labeled edges and typed nodes. This means that every node has properties (key-value pairs) and a type. Also, every property of a node or label in an edge 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 is a structure over $Vals$, with the following elements:

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

This is defined with the following structures in Coq.

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

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

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

This definition of graph is completely independent of any GraphQL schema, so we need a way to relate our data to our type system. To this end, we define the notion of *conformance* of a graph. This notion is, in essence, a well-formedness property for graphs with respect to a given schema.

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 $\mathcal{S}$.

Once again, we can define a structure that encapsulates the notion of a *conformed* graph, by passing it a graph and a proof of its *conformance* to a particular schema. TD ►*The property of conformance is implemented as a boolean predicate.*◄

```
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).

Structure 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. These properties include

⟨*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.

things such as: every node must have an object type and their properties must be defined in their associated type, or an edge's label must be declared as a field in the source node's type and the target node must have a type compatible to the field's return type, amongst other things.

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. Leaf nodes are selections of fields with a scalar return type. Inner nodes can be either selections on fields with an object or abstract return type, or inline fragments which condition when its subqueries are evaluated. 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 close as possible. The grammar can be described as follows TD ►*It actually is a tiny bit different but unnecessarily... this captures it*◄ :

⟨*Query*⟩ ::= ⟨*name*⟩ **(** ⟨*Arg*⟩* **)**
  | ⟨*alias*⟩ **:** ⟨*name*⟩ **(** ⟨*Arg*⟩* **)**
  | ⟨*name*⟩ **(** ⟨*Arg*⟩* **) {** ⟨*Query*⟩+ **}**
  | ⟨*alias*⟩ **:** ⟨*name*⟩ **(** ⟨*Arg*⟩* **) {** ⟨*Query*⟩+ **}**
  | **... on** ⟨*name*⟩ **{** ⟨*Query*⟩+ **}**

⟨*Arg*⟩ ::= ⟨*name*⟩ **:** ⟨*value*⟩

And the implementation in GraphCoQL is the following. TD ►*Not sure how to display this.*◄

```
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)
```

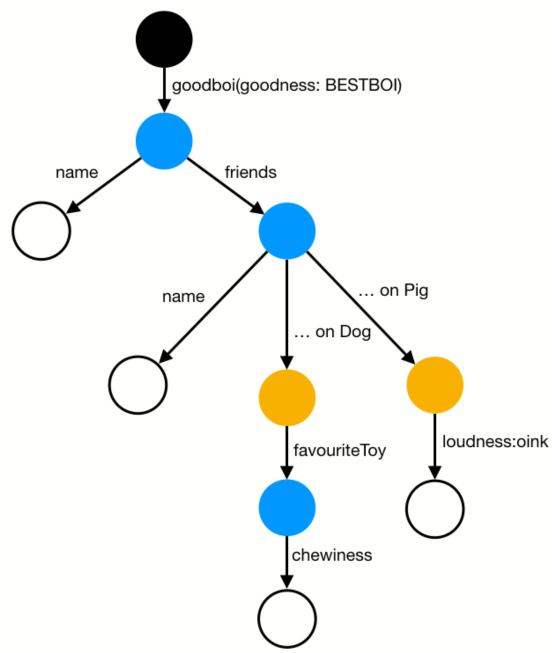

Figure 5. GraphQL query as a tree.

```
                   (arguments : seq (Name * Vals))
                   (subqueries : seq Query)

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

TD ►*PH includes a rule for list of queries at the same level as the other rules. The issue with this is that it allows building arbitrary tree instead of just a list of queries. They can be flattened but it's something they just assume I believe.*◄

Before evaluating queries, they must go through a validation process. This is similar to the *well-formedness* of schemas or *conformance* of graphs. Whenever we validate queries, there is always a notion of a type in context. This is the type to which we might be requesting information on its fields[4]. Requesting a field might be valid in a certain type but not in others. Similarly, a field may have a particular return type in one case and a different one in another type. Due to space constraints, we do not include the complete definitions.

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

- Selections in φ are consistent by themselves. This notion includes things such as: if we query a field, then that field is defined in the type in context and its arguments are defined in the given field, or if it is an inline fragment, then the type condition has to be valid wrt. the type in context.
- Field merging between fields is possible. During the evaluation process, fields with the same response name are collected and merged to ensure that they are all executed at the same time. This validation rule checks that it makes sense to merge those fields. The following example illustrates two queries that have the same response name but should not be merged. The first one is accessing the field `name` while the second is accessing the field `age` but renaming it to `name`. Both are selections on different fields of the same type but with the same response name.

```
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).
```

- Fields with same response name have compatible response shapes. This checks whether two fields with the same response name will produce response values that are consistent to each other. These values should be unambiguous for a user. For instance, the following example TD ►*These examples look a bit off I think.*◄ shows two queries that produce similar responses but with ambiguous values. In the first one, we ask for dog's names, which are strings, and in the second for pig's ages, which are integers. We also rename the `age` value to `name`. The responses we get will have some cases where `name` is associated to a string and other where it is associated to integers.

```
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).
```

```
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].
```

Finally, with these definitions we can build queries in a GraphQL service. From now on, we will assume that queries are well-formed with respect to a given schema. We can then move onto their semantics.

3.4 Semantics

We are now ready to review how queries are evaluated. We will begin by briefly reviewing the responses generated by executing our queries. Then we will give an informal description of our semantics, followed by the formal definition. We will finish by discussing some implementation choices and comparison with the Spec and HP.

We chose to model responses as a tree structure, similar to JSON. The Spec only states that responses must be a map. We chose this structure because it is similar to the one used by queries and because it is simpler to preserve order of the

[4]At top level, this would be the query type.

responses. The ordering of responses is not a hard requirement but it is one of the selling points for GraphQL (queries and their responses are very similar and easy to read). We use option types to represent null values in the leaves of the response tree.

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

Variable (Vals: eqType).

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

Moving onto the semantics of GraphQL queries. We chose to model it similarly to HP, in the sense that our data model is a graph. Therefore, a query represents a navigation over an underlying graph. At top level, our query starts from the root node and then moves around its edges and nodes, collecting data along its way. In this sense:

- A field selection represents one of two things: accessing a node's property or traversing an edge to a neighboring node. On the neighboring nodes we recursively evaluate the subqueries.
- An inline fragment conditions whether we access some value of a node or if we use it to traverse to other nodes.

Figure 6 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 or when it is evaluating an inline fragment. Aliased fields are omitted for brevity.

There are two major aspects that we need to address about our formalization; errors and completeness.

The first one is that we currently do not handle errors during execution. This is due to two main reasons: our semantics assumes it receives valid queries and we have not yet implemented non-null types. These relates to the two kinds of errors one may encounter: validation and execution errors. The first ones are captured before execution and displayed to the user. Our semantics has to deal with a case which would be ruled out by the validation process. We believe both cases can be covered by including X (monad/reasonably exceptional type theory/etc) TD ►*rewrite*◄ .

The second major aspect refers to completeness. Our semantics does not cover all possible responses expected by a GraphQL service. In particular, it does not account for list types of depth bigger than one, when its inner type is not a scalar type[5]. For instance, one might want to get information about friends but grouped by their age. This could be modeled as a field with type `[[Human]]`, where the list type has depth 2. A response for this query would look something like `"friends":[[...], ..., [...]]`. This response cannot be generated by our semantics[6].

The main challenge in this case is to define what this nested list types represent in a graph. If we take a simple case of a field with type `[Human]`, we can model it as neighbors of a node. However, if we increase the nesting such as `[[Human]]`, it becomes harder to model. What does this represent in the graph? Should we introduce blank nodes in between the source node and the `Human` nodes? Are these inner edges labeled? Should there be a blank node per each level of nesting or a single one with edges to itself? All these questions do not have a straightforward answer. Our semantics, as the one definded in PH, simply ignores any nesting bigger than one. TD ►*This is where it can be modelled using Functors. The Spec checks if it received a collection and applies map to eventually get to the concrete values. Not sure how to put this out there.*◄

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.

4.1 Normal form

The notion of *normal form* is defined by the conjunction of two other properties; being *grounded*[7] and being *non-redundant*.

Groundness

Informally, the *groundness* property refers to whether queries are completely specified down to the objects and scalar types.

[5]HP goes a step further and does not allow any type of nested list result.

[6]It can be defined with the `Response` structure but not generated with the semantics.

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

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

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

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

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

Figure 6. Semantics for GraphQL queries. TD ►*This looks bad but I don't know how to format it :/*◄

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.
```

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[8]. 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

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

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. 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 [1]

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] 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.