A Mechanized Formalization of GraphQL

Tomás Díaz
IMFD Chile
tdiaz@imfd.cl

Federico Olmedo
Computer Science Department
University of Chile & IMFD Chile
folmedo@dcc.uchile.cl

Éric Tanter
Computer Science Department
U. of Chile & IMFD Chile & Inria Paris
etanter@dcc.uchile.cl

Abstract

GraphQL is a novel language for specifying and querying web APIs, allowing clients to flexibly and efficiently retrieve data of interest. The GraphQL language specification is unfortunately only available in prose, making it hard to develop robust formal results for this language. Recently, Hartig and Pérez proposed a formal semantics for GraphQL in order to study the complexity of GraphQL queries. The semantics is however not mechanized and leaves certain key aspects unverified. We present GraphCoQL, the first mechanized formalization of GraphQL, developed in the Coq proof assistant. GraphCoQL covers the schema definition DSL, query definitions, validation of both schema and queries, as well as the semantics of queries over a graph data model. We illustrate the application of GraphCoQL by formalizing the key query transformation and interpretation techniques of Hartig and Pérez, and proving them correct, after addressing some imprecisions and minor issues. We hope that GraphCoQL can serve as a solid formal baseline for both language design and verification efforts for GraphQL.

CCS Concepts • **Information systems → Web services**; *Query languages*; • **Theory of computation → Semantics and reasoning**.

Keywords GraphQL, mechanized metatheory, Coq.

ACM Reference Format:
Tomás Díaz, Federico Olmedo, and Éric Tanter. 2020. A Mechanized Formalization of GraphQL. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372885.3373822

*This work is partially funded by ERC Starting Grant SECOMP (715753).

CPP '20, January 20–21, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7097-4/20/01…$15.00
https://doi.org/10.1145/3372885.3373822

1 Introduction

GraphQL is an increasingly popular language to define interfaces and queries to data services. Originally developed internally by Facebook as an alternative to RESTful Web Services [21], GraphQL was made public in 2015, along with a reference implementation[1] and a specification—both of which have naturally evolved [16]. Since early 2019, as a result of its successful adoption by major players in industry, GraphQL is driven by an independent foundation[2]. The key novelty compared to traditional REST-based services is that tailored queries can be formulated directly by clients, allowing a very precise selection of which data ought to be sent back as response. This supports a much more flexible and efficient interaction model for clients of services, who do not need to gather results of multiple queries on their own, possibly wasting bandwidth with unnecessary data.

The official GraphQL specification, called Spec hereafter, covers the definition of interfaces, the query language, and the validation processes, among other aspects. The specification undergoes regular revisions by an open working group, which discusses extensions and improvements, as well as addressing specification ambiguities. Indeed, as often happens, the Spec is an informal specification written in natural language. Considering the actual vibrancy of the GraphQL community, sustained by several implementations in a variety of programming languages and underlying technologies, having a formal specification ought to bring some welcome clarity for all actors.

Recently, Hartig and Pérez [18] proposed the first (and so far only) formalization of GraphQL, called H&P hereafter. H&P is a formalization "on paper" that was used to prove complexity boundaries for GraphQL queries. Having a mechanized formalization would present many additional benefits, such as potentially providing a faithful reference implementation, and serving as a solid basis to prove formal results about the GraphQL semantics.

For instance, the complexity results of Hartig and Pérez rely on two techniques: *a)* transforming queries to equivalent queries in some normal form and, *b)* interpreting queries in a simplified but equivalent definition of the semantics. However, Hartig and Pérez do not provide an algorithmic definition of query normalization, let alone proving it correct and semantics preserving; nor do they prove that the

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

simplified semantics is equivalent to the original one when applied to normalized queries.

This work presents **the first mechanized formalization of GraphQL**, carried out in the Coq proof assistant [13], called GraphCoQL (|græf·co·kəl|). GraphCoQL currently includes the definition of the Schema DSL, query definition, schema and query validation, and the semantics of queries, defined over a graph data model as in H&P (§3).

As well as precisely capturing the semantics of GraphQL, GraphCoQL makes it possible to specify and prove the correctness of query transformations, as well as other extensions and optimizations made to the language and its algorithms. We illustrate this by studying H&P's notion of query normalization (§4). Specifically, we define an algorithmic presentation of normalization, which we then prove correct and semantics preserving. We also formalize and prove the equivalence between the original query evaluation semantics and the simplified semantics used by H&P for normalized queries. In the process, we address some imprecisions and minor issues that Coq led us to uncover.

We discuss the limitations and validation of GraphCoQL, including its trustworthiness, in §5. We hope that GraphCoQL can serve as a starting point for a formal specification of GraphQL from which reference implementations can be extracted. Although we have not yet experimented with extraction, GraphCoQL facilitates this vision by relying on boolean reflection as much as possible.

We briefly introduce GraphQL in §2. We discuss related work in §6 and conclude in §7. The fundamental challenge of the Coq formalization of GraphQL in comparison to similar formalization efforts (e.g. [1, 2, 4]) resides in the non-structural nature of most definitions related to GraphQL queries. In particular, as we will see, both query semantics and query normalization are defined by well-founded induction on a notion of query size (§ 3.3).

GraphCoQL and the results presented in this paper have been developed in Coq v.8.9.1 and are available online on GitHub[3]. GraphCoQL extensively uses the Mathematical Components [20], Ssreflect [15] and Equations [22] libraries. This work is based on the latest GraphQL specifications, dated June 2018 [16].

2 A Brief Introduction to GraphQL

We briefly introduce GraphQL by means of the running example that we use in the rest of the presentation. The example is about a fictional dataset Artists, containing information about artists and the artworks they have been involved in, particularly movies and books. We discuss the dataset schema, the underlying data model and the query evaluation.

[3]https://github.com/imfd/GraphCoQL

```
type Artist
{
  id: ID
  name: String
  artworks(role:Role): [Artwork]
}

enum Role {
        ACTOR
        DIRECTOR
        WRITER
}

union Artwork = Fiction
  | Animation
  | Book

interface Movie
{
  id: ID
  title: String
  year: Int
  cast: [Artist]
}

type Fiction implements Movie
{
  id: ID
  title: String
  year: Int
  cast: [Artist]
}

type Animation implements Movie
{
  id: ID
  title: String
  year: Int
  cast: [Artist]
  style: Style
}

enum Style {
        2D
        3D
        STOPMOTION
}

type Book
{
  id: ID
  title: String
  year: Int
  ISBN: String
  author: Artist
}

type Query {
  artist(id:ID): Artist
  movie(id:ID): Movie
}

schema {
  query: Query
}
```

Figure 1. Example of GraphQL schema.

GraphQL schema. The first step when defining a GraphQL API is to define the service's schema, describing the type system and its capabilities. The type system describes how data is structured and precisely what can be requested and received. A schema consists of a set of types, which can have declared fields. Only fields that are part of such type definitions can be accessed via a GraphQL query. In contrast to traditional data management systems, field accesses in GraphQL are handled by the backend via user-defined functions (called resolvers).

Figure 1 depicts the schema for the Artists dataset, which contains artists and their artworks. The definition is done using the GraphQL Schema Definition Language. The object type `Artist` declares three fields: an identifier, a name, and a list of artworks in which the artist has participated. The list of artworks is declared of type `[Artwork]`. Note that the field may receive an argument, `role`, which can be used to select only those artworks for which the author plays a certain role, such as being the director. Possible roles are modeled with the enum type `Role`, containing three scalar values. An `Artwork` is a disjoint union of fictional movies, animated movies and books. Both fictional and animated movies are defined as object types, namely `Fiction` and `Animation`, implementing the same interface type `Movie`.[4] An object implementing an interface may add more fields, such as the field `style` in the type `Animation`. To model the animation style, we use the enum type `Style` containing

[4]Note that GraphQL still requires objects to redeclare the implemented fields.

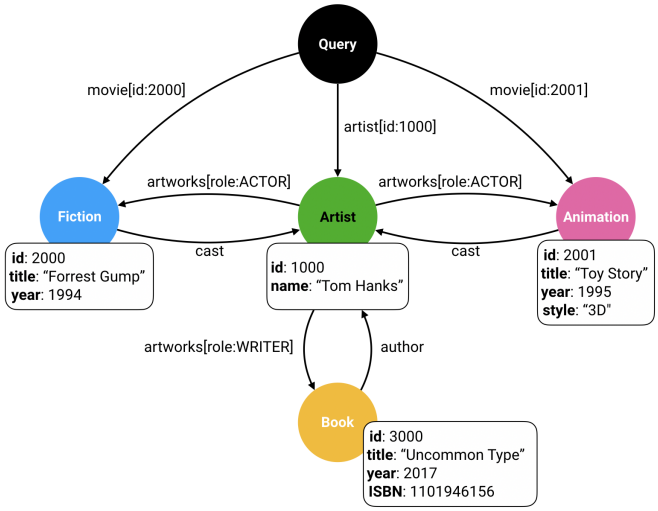

Figure 2. Example of GraphQL graph.

three scalar values. We define books with the object type Book which, for simplicity, does not implement any interface.

Finally, GraphQL requires that the schema includes a query type, which, for convenience, we call `Query`. This type is the same as a regular object type[5], however it is special in the sense that it represents the entry point from where users have to start querying the API and explore the dataset. In this API, a user may only access the Artists dataset by requesting a particular artist or movie (with a given id).

Graph data model. The complete Artists dataset can be represented with the graph in Figure 2. Nodes in the graph represent object values as defined in the schema; they are tagged with the corresponding object type and include a set of properties (key-value pairs) that describe the object's content. Edges between nodes are accompanied with a label that indicates the relationship they establish. For instance, the central node in the graph represents an object of type `Artist`. The node properties say that the artist is called *Tom Hanks* and his id is 1000. The node's outgoing edges indicate that the artist performed as an actor in the movies *Forrest Gump* and *Toy Story* (represented by the leftmost and rightmost nodes, respectively), and that he was the author of book *Uncommon Type* (represented by the bottommost node).

GraphQL query and response. With the schema and data defined, it is possible to query the service. As previously hinted, GraphQL queries are simply requests over the fields of types defined in the schema. They have a tree structure, similar to JSON, which follows the fields and relations established between the types in the type system.

[5]The query type can e.g. have any custom name and even implement interfaces.

```
query {
  artist[id:1000] {
    name
    artworks[role: ACTOR] {
      ... on Animation {
        title
        style
      }
      ... on Fiction {
        title
        releaseYear:year
      }
    }
  }
}
```

```
{
  "artist": {
    "name":"Tom Hanks",
    "artworks":[
      {
        "title": "Toy Story",
        "style": "3D"
      },
      {
        "title": "Forrest Gump",
        "releaseYear": 1994
      }
    ]
  }
}
```

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

To illustrate this, we define the query in Figure 3. Intuitively, the query is asking for the name of the artist with id 1000, as well as the title and some additional information of the artworks where he performed as an actor. The request starts with the keyword `query`, which allows proceeding with any of (and as many as) the dataset entry points declared by the query type `Query`. Concretely, we select field `artist` and include the value of argument `id` to indicate which artist we want to retrieve. Because the field is of an object type, namely `Artist`, we can continue making requests over the fields of this type, precisely specifying the desired information about the fetched artist. In this case, the query continues with the field `name` and the field `artworks`. The argument `role` of field `artworks` is used to restrict the search to those artworks where the artist performed as an actor. For such artworks, we request further information depending on the concrete kind of artwork. (Recall that union type `Artwork` is composed by the object types `Book`, `Fiction` and `Animation`). For animated movies, we request the movie's title and animation style, while for fictional movies, we request the title and year of release; for books, we require no related information. This is achieved by the pair of selections `... on Animation { title; style }` and `... on Fiction { title; releaseYear:year }`, called *inline fragments*. In the latter, the field selection `releaseYear:year` introduces a *field alias*, indicating that in the response the original field `year` should be renamed to `releaseYear`. Field aliasings are particularly relevant when validating and transforming queries, and originated one of the imprecisions uncovered in H&P's definitions (see § 4.5).

This query is then evaluated, resulting in the response depicted on the right of Figure 3. Observe that the response structure closely resembles the query, which is a key usability asset of GraphQL. While data accesses are handled by resolvers in the backend, the query evaluation process of GraphQL can be intuitively understood by considering a graph data model. In such a model, the selections in a query indicate what edges to traverse in the graph and what properties to access on each node. In this manner, starting from

```
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]
```

```
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.
```

Figure 4. Definition of GraphQL types: (left) Spec grammar; (right) GraphCoQL definition.
The $(\cdot)^?$ notation denotes optional attributes.

the node with type `Query`, the field `artist(id:1000)` indicates that we are looking for an adjacent node that can be reached after traversing an edge whose label matches the field and id. The first step then takes the evaluation process to the central node, of type `Artist`. From there, it accesses the value of the node's property matching the field `name` and then continues navigating the graph in search of the artist's artworks.

To summarize, in order to define a GraphQL service it is necessary to define the schema that describes the service's type system, to which both the underlying dataset and the queries must adhere. GraphQL queries consist of *selections* over fields and types defined in the schema, and their responses closely match the queries tree structure.

3 GraphCoQL

In this section we describe our formalization of GraphQL in Coq. We start by defining a schema and its properties (§3.1), then the graph data model (§3.2), and finally present queries (§3.3) and their semantics (§3.4). We highlight key connections with the Spec [16] and H&P [18] along the way, and end this section with a discussion of various design considerations (§3.5).

3.1 Schema

We formalize schemas and type definitions following the Spec. A schema is represented as a record, containing a list of type definitions and a root query type that specifies the entry points to the dataset. A `Name` is simply a string, according to the Spec.

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

Schemas may also include additional root types for specifying mutations and subscriptions (cf. [16, §3.2.1]). These operations are, however, out of the scope of this work.

Type definitions in GraphCoQL closely follow the Spec, as depicted in Figure 4. A type may be a scalar type, an object type, which possibly implements a set of interfaces, an interface type, a union type or an enumeration type. Object and interface type definitions comprise a set of fields; union types are defined by a set of type names and enumeration types by a set of scalar values.

The mere syntax of schema and types does not ensure that only valid schemas are defined. For instance, one can build an object that implements scalar types or use a nonexistent type as the root query type. To avoid this problem, the Spec provides several validation rules (cf. [16, §3]). We refer to these rules as the *well-formedness* condition of a GraphQL schema:[6]

Definition 3.1. A GraphQL schema is *well-formed* if:

- its query root type (is defined and) is an object type,
- there are no duplicated type names, and
- every type definition is *well-formed*.

In GraphCoQL, this is captured by the Boolean predicate below.

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

The notion of well-formedness for type definitions requires e.g. that the members of union types are existent object types and that object and interface types contain at least one field. Note that the Spec syntactically requires certain sets of elements to be non-empty (see the grammar on the left of Figure 4). This requirement is not enforced at the level of

[6] In H&P, this condition is referred to as the *consistency* of schemas.

a type definition in GraphCoQL, but is instead part of the well-formedness check. Full definitions can be found in our Coq development.

For convenience, we encapsulate schemas with their well-formedness proof in a single structure.

```
Record wfGraphQLSchema := WFGraphQLSchema {
    schema : graphQLSchema;
    _ : schema.(is_a_wf_schema);
}.
```

3.2 Data Model

Following H&P, we adopt a data model based on graphs, where datasets are modeled as directed property graphs, with labeled edges and typed nodes. Nodes contain a type and a set of properties (key-value pairs) and edges contain single labels. Properties and labels may contain a list of arguments (key-value pairs). Values are either scalars or lists of values.

Definition 3.2. A *GraphQL graph* is defined by:

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

To model graphs in Coq, we first model values. This requires fixing a domain of scalar values with decidable equality.

```
Variable Scalar : eqType.

Inductive Value : Type :=
| SValue : Scalar -> Value
| LValue : seq Value -> Value.
```

Labels, nodes and graphs are all represented as records:

```
Record label :=
  Label { lname : string; args : seq (string * Value) }.

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

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

Note that graphs are modeled using sequences—rather than sets—of edges. In our experience, this design decision led to a simpler formalization, as verifying edge unicity extrinsically is straightforward.

Intuitively, the dataset modeled by a GraphQL graph is built following a schema. However, the definition of GraphQL graph above is fully independent of any schema. To capture this relationship, we employ the following notion of *conformance*, partially based on H&P:

Definition 3.3. A GraphQL graph $\mathcal{G}$ *conforms* to a schema $\mathcal{S}$ if:

- the types of $\mathcal{G}$ root node and $\mathcal{S}$ query root coincide,
- every node of $\mathcal{G}$ *conforms* to $\mathcal{S}$, and
- every edge of $\mathcal{G}$ *conforms* to $\mathcal{S}$.

The conformance of nodes validates that a node's type is declared as an object type in the schema and that its properties conform. In turn, a property *conforms* if its key and arguments are defined in a field in the corresponding object type, and any value, either in an argument or the property's value, is *valid* w.r.t. the expected types described in the schema. For instance, if a field has type `Float`, the Spec dictates that a node property matching the field must have a value that represents a double-precision fractional value (cf. [16, §3.5.2]). To model this validation of values, we parameterize the Coq development with a Boolean predicate $\mathit{check\_scalar} : \mathit{graphQLSchema} \to \mathit{Name} \to \mathit{Scalar} \to \mathit{Bool}$.

The conformance of edges imposes some final natural restrictions on graphs. For instance, given an edge, it requires that the label match some field in the type of the source node, and that the type of the target node be compatible with the type of the matched field.

It is worth noting that, in accordance with the flexibility advocated by graph databases, the notion of conformance does not require that a graph contain full information about the represented objects. More precisely, a node need not provide values for all the fields in its type. For instance, in our running example in Figure 2, the bottommost node of type `Book` need not contain a property defining the book title.

With this in mind, the notion of conformance of a graph w.r.t. a schema is formalized as follows:

```
Variable check_scalar :
    graphQLSchema -> Name -> Scalar -> bool.

Definition is_a_conforming_graph
  (s : wfGraphQLSchema)
  (g : 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, containing a graph and a proof of its conformance to a particular schema.

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

3.3 Queries

To define queries we faithfully follow the Spec, as shown in Figure 5. A query consists of a list of selections, and can optionally be named. A *selection* σ can be a single field with arguments ($\mathtt{f}[\overline{\alpha}]$), a field with arguments followed by a set of subselections ($\mathtt{f}[\overline{\alpha}]\{\overline{\sigma}\}$) or an inline fragment comprising

a type condition and a set of subselections (`... on t {`$\overline{\sigma}$`}`). Fields can be aliased (`a:f[`$\overline{\alpha}$`]`, `a:f[`$\overline{\alpha}$`]{`$\overline{\sigma}$`}`). For notational simplicity, when a field selection contains an empty list of arguments, we omit it and simply write the field name.

Intuitively, a query has a tree structure, where leaves correspond to fields of scalar types and inner nodes correspond to either fields of some object type or abstract type (i.e. an interface or union), or to inline fragments that condition when their subselections are evaluated.

Observe that the definition of queries in Figure 5 is not bound to any schema, thus requiring a separate validation process to ensure that they adhere to a given schema. We introduce the notion of query *conformance*, based on a set of validation rules scattered throughout the Spec (cf. [16, §5]). The validity of queries depends on the validity of their selection sets, which in turn requires the notion of *type in scope* at a given selection location. To illustrate this, consider the query to the right with two occurrences of field `title`. In the first occurrence, the field is requesting information about the `Animation` type, while in the second it is requesting information about the `Fiction` type. The distinction is important because some field selections might be valid in some contexts but not in others. For instance, this is the case of field `style`, which is valid in the scope of the `Animation` type but it is invalid in the scope of the `Fiction` type, as the `Fiction` type does not contain any such field.

```
query {
  movie[id:2000] {
    ... on Animation {
      title
      style
    }
    ... on Fiction {
      title
      style
} } }
```

Now that we have clarified the notion of type in scope, we define the notion of conformance for selection sets.

Definition 3.4. A GraphQL selection set $\overline{\sigma}$ *conforms* to a schema $\mathcal{S}$ over a type in scope *ts* if:

- every selection in $\overline{\sigma}$ is well-formed w.r.t *ts*, and
- any two field selections in $\overline{\sigma}$ are type-compatible and renaming-consistent.[7]

The first rule ensures that every selection is well-formed on its own, w.r.t. the type in scope. This requirement depends on the kind of selection. For instance, if the selection is a field, the rule checks that the field is part of the type in scope and that its arguments are correctly provided; if the selection is an inline fragment, then the type condition must share at least one subtype with the type in scope. This rule also includes validating the values used in arguments, similarly to the case of graphs.

In the second rule, the type-compatibility requirement forbids the selection set to produce results of different types for the same key; e.g. the following query

[7] In the Spec, these two notions roughly correspond to *SameResponseShape* and *FieldsInSetCanMerge*, respectively (cf. [16, §5.3.2]).

$$
\begin{array}{rcl}
\varphi & ::= & \texttt{query}\ (name)^{?}\ \{\overline{\sigma}\} \\
\\
\sigma & ::= & \texttt{f}[\overline{\alpha}] \\
& | & \texttt{a:f}[\overline{\alpha}] \\
& | & \texttt{f}[\overline{\alpha}]\ \{\overline{\sigma}\} \\
& | & \texttt{a:f}[\overline{\alpha}]\ \{\overline{\sigma}\} \\
& | & \texttt{... on t}\ \{\overline{\sigma}\}
\end{array}
$$

```
Record query :=
     Query { qname : option string;
             selection_set : seq Selection }.

Inductive Selection : Type :=
| SingleField (name : Name)
               (arguments : seq (Name * Value))
| SingleAliasedField (alias : Name)
                     (name : Name)
                     (arguments : seq (Name * Value))
| NestedField (name : Name)
               (arguments : seq (Name * Value))
               (subselections : seq Selection)
| NestedAliasedField (alias : Name)
                     (name : Name)
                     (arguments : seq (Name * Value))
                     (subselections : seq Selection)
| InlineFragment (type_condition : Name)
                 (subselections : seq Selection).
```

Figure 5. Definition of GraphQL queries: (top) Spec grammar; (bottom) GraphCoQL definition.
In the Spec grammar, symbols `f`, `a` and `t` correspond to identifiers for field name, field alias, and type condition, respectively. Symbol α corresponds to a key-value pair.

```
query {                      // Possible invalid output
  artist[id:1000] {          {
    artworks[role:ACTOR] {     "artist": {
      ... on Animation {         "artworks": [
        title                     { "title": "Toy Story" },
      }                           { "title": 1994 }
      ... on Fiction {           ]
        title:year              }
} } } }                      }
```

is invalid because its evaluation produces results with the same key (`title`) but associated to values of different types (i.e. string and integer).

Formally, the definition of field *type-compatibility* is recursive. Two nested field selections are type-compatible if whenever they have the same response name, any two fields in the concatenation of their subselections (possibly reached traversing inline fragments) are also type-compatible. Two single field selections are always type-compatible, unless they have the same response name and different type.

Finally, the renaming-consistency condition ensures that fields with the same response name refer to the same piece of information. Consider, for instance, the following query:

```
query {
  movie[id:2000] {
          title
          ... on Animation {
            title:style
} } }
```

The query is considered invalid because the fields with response name `title` both refer to distinct pieces of information; the first occurrence refers to the field `title` of a `Movie`, while the second occurrence refers to the field `style` of an `Animation`.

Formally, two field selections are *renaming-consistent* if whenever they have the same response name and lie under types in scope with at least one common subtype, they have the same (actual) name, the same arguments and any two fields in the concatenation of their subselections (possibly reached traversing inline fragments) are also renaming-consistent. In the Spec, the last two rules are defined as a single validation rule (cf. [16, §5.3.2]). We chose to split them because it simplified both their implementation and reasoning about them.[8]

With the notion of conformance for selection sets at hand, the notion of conformance for queries is straightforward:

Definition 3.5. A GraphQL query φ *conforms* to a schema $\mathcal{S}$ if its selection set conforms to $\mathcal{S}$ over the query root type.

In GraphCoQL, this is captured by the following predicates:

```
Definition query_conforms
  (s : wfGraphQLSchema) (φ : query) : bool :=
    selections_conform s s.(query_type) φ.(selection_set).

Definition selections_conform (s : wfGraphQLSchema)
  (ts : Name) (σs : seq Selection) : bool :=
    let σs_with_scope := [seq (ts, σ) | σ <- σs] in
      all (is_consistent ts) σs &&
      σs_with_scope.(are_type_compatible) &&
      σs_with_scope.(are_renaming_consistent).
```

As a technical observation, most of our recursive definitions over selection sets are well-founded according to the following notion of size:

Definition 3.6 ([18])**.** The *size* of a selection σ and selection set $\overline{\sigma}$, noted $|\cdot|$, is recursively defined as:

$$|\mathtt{f}[\overline{\alpha}]| = |\mathtt{a{:}f}[\overline{\alpha}]| = 1$$

$$|\mathtt{f}[\overline{\alpha}]\ \{\overline{\sigma}\}| = |\mathtt{a{:}f}[\overline{\alpha}]\ \{\overline{\sigma}\}| = |\mathtt{...\ on\ t}\ \{\overline{\sigma}\}| = 1 + |\overline{\sigma}|$$

$$|\overline{\sigma}| = \textstyle\sum_{\sigma_i \in \overline{\sigma}} |\sigma_i|$$

3.4 Semantics

Now we have all the prerequisites to define the semantics of GraphQL queries and their selection sets. We begin by briefly examining the responses generated by executing queries and then we give an informal description of the semantics, finishing with the formal definition.

Roughly speaking, a GraphQL response maps keys (field names) to response values. We model response values with a tree structure, similar to JSON. Concretely, a response value can be either a value in `Scalar` or the distinguished value `null`, an object mapping keys to other response values, or an array of response values. Full definition is provided in Figure 6.

As a first step to define the semantics of queries, we observe that it is not compositional, in the sense that the result of a sequence of selections is not obtained by simply concatenating the results of individual selections. Therefore, the evaluation function takes as input a (whole) selection set, rather than single selections.

Informally, the evaluation of a selection set represents a navigation over a graph, starting from the root node, traversing its edges and collecting data from its nodes. To build the response, the values in nodes are coerced into proper response values. If this coercion fails or the data requested by the selection set is not present in the graph, the evaluation function simply returns `null`.

This validating transformation from values within graph nodes into values within responses is captured by the function *complete_value* (cf. [16, §6.4.3]). To carry out the transformation the function relies on two auxiliary functions: *coerce* that coerces scalar values and *check_scalar* that checks whether the resulting values have the expected type according to the schema.[9]

With this in mind, we can now define the evaluation function of selection sets.

Definition 3.7. Let $\mathcal{G}$ be a graph and $\overline{\sigma}$ a selection set, both conforming to a schema $\mathcal{S}$. The evaluation $[\![\overline{\sigma}]\!]^u_{\mathcal{G}}$ of $\overline{\sigma}$ over graph $\mathcal{G}$ from node $u \in \mathit{nodes}(\mathcal{G})$ is defined by the rules in Figure 7. The evaluation function is parametrized by a coercing function $\mathit{coerce}\colon \mathit{Scalar} \to \mathit{Scalar}$ and a value validation predicate $\mathit{check\_scalar}\colon \mathit{graphQLSchema} \to \mathit{Name} \to \mathit{Scalar} \to \mathit{Bool}$.

In order not to clutter the notation, we omit the underlying schema when defining $[\![\overline{\sigma}]\!]^u_{\mathcal{G}}$ in Figure 7. The schema is implicitly used e.g. when invoking function *complete_value*.

The definition starts with the base case of empty selection set (1), which results in an empty response. Single fields (2) correspond to accessing a node's property that matches the field name and using function *complete_value* to coerce and validate the requested value. This function is implemented by simply calling *coerce* and *check_scalar*.

Nested fields (3) represent a traversal to neighboring nodes: the evaluation function searches for nodes that are connected to the current node by an edge whose label matches the field

[8] GraphCoQL uses an optimized version of the algorithm in the Spec. Recently, a team at XING developed another optimized algorithm [29].

[9] Function *check_scalar* is also used for verifying the conformance of graphs w.r.t. schemas; see §3.2.

$$\begin{array}{rcl} \rho & ::= & \mathtt{v} \\ & | & \{(\mathtt{f} : \rho) \ldots (\mathtt{f} : \rho)\} \\ & | & [\rho \ldots \rho] \end{array}$$

```
Inductive ResponseValue : Type :=
| Leaf : option Scalar -> ResponseValue
| Object : seq (Name * ResponseValue) -> ResponseValue
| Array : seq ResponseValue -> ResponseValue.

Definition GraphQLResponse := seq (Name * ResponseValue).
```

Figure 6. Definition of GraphQL responses: (left) grammar à la JSON; (right) GraphCoQL definition.
The keywords v and f represent leaf values and keys in a key-value pair, respectively.

$$\boxed{[\![\cdot]\!]^u_{\mathcal{G}} : \mathit{seq\ Selection} \to \mathit{GraphQLResponse}}$$

$$[\![\cdot]\!]^u_{\mathcal{G}} = [\cdot] \tag{1}$$

$$[\![\mathtt{f}[\overline{\alpha}] :: \overline{\sigma}]\!]^u_{\mathcal{G}} = \mathtt{f}{:}(\mathit{complete\_value}(\mathit{ftype}(u.\mathit{type}, \mathtt{f}), u.\mathit{property}(\mathtt{f}[\overline{\alpha}]))) :: [\![\mathit{filter}(\mathtt{f}, \overline{\sigma})]\!]^u_{\mathcal{G}} \tag{2}$$

$$[\![\mathtt{f}[\overline{\alpha}]\ \{\overline{\beta}\} :: \overline{\sigma}]\!]^u_{\mathcal{G}} = \begin{cases} \mathtt{f}{:}[\mathit{map}\ (\lambda v_i.\ \{[\![\overline{\beta} +\!\!+ \mathit{merge}(\mathit{collect}(u.\mathit{type}, \mathtt{f}, \overline{\sigma}))]\!]^{v_i}_{\mathcal{G}}\})\ u.\mathit{neighbors}_{\mathcal{G}}(\mathtt{f}[\overline{\alpha}])] :: [\![\mathit{filter}(\mathtt{f}, \overline{\sigma})]\!]^u_{\mathcal{G}} \\ \qquad\qquad \text{if } \mathit{ftype}(u.\mathit{type}, \mathtt{f}) = \mathit{list} \text{ and } \{v_1, \ldots, v_k\} = \{v_i \mid (u, \mathtt{f}[\alpha], v_i) \in \mathit{edges}(\mathcal{G})\} \\ \mathtt{f} : \{[\![\overline{\beta} +\!\!+ \mathit{merge}(\mathit{collect}(u.\mathit{type}, \mathtt{f}, \overline{\sigma}))]\!]^v_{\mathcal{G}}\} :: [\![\mathit{filter}(\mathtt{f}, \overline{\sigma})]\!]^u_{\mathcal{G}} \\ \qquad\qquad \text{if } \mathit{ftype}(u.\mathit{type}, \mathtt{f}) \neq \mathit{list} \text{ and } (u, \mathtt{f}[\alpha], v) \in \mathit{edges}(\mathcal{G}) \\ \mathtt{f}{:}\mathtt{null} :: [\![\mathit{filter}(\mathtt{f}, \overline{\sigma})]\!]^u_{\mathcal{G}} \qquad \text{if } \mathit{ftype}(u.\mathit{type}, \mathtt{f}) \neq \mathit{list} \text{ and } \nexists v \text{ s.t. } (u, \mathtt{f}[\alpha], v) \in \mathit{edges}(\mathcal{G}) \end{cases} \tag{3}$$

$$[\![\ldots \text{ on } \mathtt{t}\ \{\overline{\beta}\} :: \overline{\sigma}]\!]^u_{\mathcal{G}} = \begin{cases} [\![\overline{\beta} +\!\!+ \overline{\sigma}]\!]^u_{\mathcal{G}} & \text{if } \mathit{fragment\_type\_applies}(u.\mathit{type}, \mathtt{t}) \\ [\![\overline{\sigma}]\!]^u_{\mathcal{G}} & \text{otherwise} \end{cases} \tag{4}$$

Figure 7. Semantics of GraphQL selection sets, adapted from H&P and the Spec.
As usual, notation $x :: y$ on the left denotes pattern matching deconstruction of a list into its head x and tail y; on the right, it denotes list construction. *property* and *type* are accessors to a node property and type. *neighbors* gets the neighbors of a node whose edge is labeled with the given field. *edges* gets the set of edges of a graph. *ftype* retrieves the type of a field from the underlying schema. *list* represents the list type (over any other type).

name and then evaluates the subselections on these nodes. If the field has list type, there is no constraint on the number of such neighboring nodes to recursively continue the evaluation. On the contrary, if the field does not have list type, then there should be exactly one neighboring node to successfully continue the recursive evaluation on this node (the case of multiple neighboring nodes is discarded by the conformance assumption); if there is no neighbor, the result is considered `null`.

To avoid the duplication of responses, rules (2) and (3) handling the evaluation of fields ensure that within a selection set, fields with the same response name are evaluated only once. This is achieved by collecting and merging all fields in subsequent selections that have the same response name as the current field being evaluated (using auxiliary functions *collect* and *merge*) and removing these fields from subsequent selections (using auxiliary function *filter*), before they are evaluated.

Finally, inline fragments (4) simply condition whether their subselections are evaluated (in the current node) or not. The decision is based on the *fragment_type_applies* predicate (cf. [16, §6.3.2]) that verifies whether the guard type matches the current node type or supertype thereof.[10]

For space reason, Figure 7 does not present aliased fields. They are evaluated the same way as unaliased fields, but differ in that the produced key-value pairs are renamed accordingly.

Definition 3.8. Let $\mathcal{G}$ be a graph and φ a query, both conforming to a schema $\mathcal{S}$. The result of evaluating φ over $\mathcal{G}$ is obtained by evaluating the selection set of φ from the root node of $\mathcal{G}$, that is, $[\![\varphi]\!]_{\mathcal{G}} = [\![\varphi.\mathit{selection\_set}]\!]^{\mathcal{G}.\mathit{root}}_{\mathcal{G}}$.

In GraphCoQL we have:

```
Definition execute_query (s : wfGraphQLSchema)
  (g : conformedGraph s check_scalar)
  (φ : query) :=
    execute_selection_set s check_scalar g coerce
      g.(root) φ.(selection_set).
```

3.5 Design Considerations and Discussion

We now discuss some design considerations that manifested when developing GraphCoQL, in particular with respect

[10] Function *fragment_type_applies* is called *does_fragment_type_applies* in the Spec.

to the two existing sources of information on GraphQL, namely the Spec and H&P.

Representation choices. The definition of schemas in H&P relies on defining sets of field names, types and values, and using a number of functions to relate the different elements. For example, the object types of the schema from Figure 1 are defined as the set $O_T = \{Artist, Fiction, Animation, Book, Query\}$ and their fields are defined with a function $fields_S : O_T \to 2^F$, where F is a set of field names. In contrast, GraphCoQL uses a more data-structure centric approach, in which schemas are records. While H&P's approach simplifies many validation rules, our approach is closer to the schema definition language from the Spec (which did not exist yet when H&P was published). Another advantage of the approach we take is that it is closer to an actual implementation and hence the definitions could be directly extracted as a reference implementation. The same observation holds for the data model.

Schema validation. The definition of well-formedness of a schema given by H&P does not capture all the validation rules established by the Spec. In particular, it does not properly account for arguments of implemented fields in objects implementing interfaces. A corrected version of H&P has been proposed recently by Hartig and Hidder [17], which coincides with both the Spec and GraphCoQL.

Data model. We observe that both H&P and GraphCoQL have a slightly limited graph model that does not entirely describe the universe of a GraphQL service: graphs cannot model *nested* lists, when the underlying type is an object or abstract type.[11] For instance, these semantics cannot account for a selection over a field whose type is `[[Artist]]`. The underlying question is how to interpret such nested lists in a graph-based data model.

The importance of this limitation in practice remains to be assessed. Recent empirical studies that analyze the structure of GraphQL schemas over a collection of industrial and open-source projects provide some insights such as the most common object types, which indeed do not have nested list types [19, 27]. Regardless, we believe that while this limitation might turn out problematic for an extracted reference implementation of GraphQL, it should play little role in metatheoretical results, as those studied in §4.

Responses. The Spec only states that responses are a map from keys to values, but encourages the ordering of selections to be preserved (cf. [16, §7.2.2])—the similarity between query selections and responses being one of GraphQL's attraction for programmers. We embrace this recommendation: modeling of responses as trees allows us to preserve similarity w.r.t. selections and ordering of response values.

[11] The situation is worse in H&P, which cannot model nested list types for scalar types.

The downsides of this representation choice are possible duplication of response names, and access cost. We establish unicity of names by extrinsic proofs, and consider that the access cost is not of primary importance in this work.

Query semantics. The query semantics of GraphCoQL (Figure 7) is similar to the Spec in that it performs a collection of fields at the level of *selections*. In contrast, H&P adopt a different formulation in which collection is done at the level of *responses*, i.e. as a post-processing phase. Initially, we experimented with the H&P approach in order to be as close as possible to their formalization. However, the non-structurally recursive nature of both the transformations and the post-processing function made reasoning about semantic equivalence of queries very hard. In contrast, we found that following the Spec approach made such reasoning in Coq much more convenient.

Finally, while the Spec presents the query evaluation semantics in two separate phases, first grouping fields by name and then evaluating each group, we define the whole query evaluation in a single pass. The technical reason is that following the Spec approach makes Equations produce a reasoning principle for the query evaluation function that is too large and inconvenient to work with. With our presentation, the reasoning principle is more concise and adequate, due to a more localized handling of inline fragments.

4 Case Study: Normalization

To illustrate how GraphCoQL can be used to reason about query transformations, we study the *normalization* process proposed by Hartig and Pérez (H&P) [18], which is fundamental for the complexity results they prove. These results are based on two premises: *a)* every query can be normalized to a semantically-equivalent query; *b)* on such queries, one can use a simplified evaluation function. For normalization, H&P provide a set of equivalence rules to establish the existence of a normal form. However they do not provide an algorithm for query normalization, let alone prove it correct and semantics preserving. Likewise, they do not prove the equivalence of the simplified semantics when applied to normalized queries.

In this section, we show how to define the property of being in *normal form* in GraphCoQL, define a normalization procedure, and prove it both correct and semantics preserving. Finally, we define the simplified semantics given by H&P, and prove it equivalent to the original semantics (§ 3.4) when applied to normalized queries.

4.1 Defining Normal Forms

The notion of *normal form* introduced by Hartig and Pérez consists of the conjunction of two conditions: being in *ground-typed normal form* and being *non-redundant*.

Ground-typed normal form. Informally, a query is in ground-typed normal form, or *grounded* for short, if any two selections within the same selection set are of the same kind, either field selections or inline fragments. Consider for instance the queries below:

```
// Not grounded query
query {
  movie[id:2000] {
    title
    ... on Fiction {
          year
    }
  }
}
```

```
// Grounded query
query {
  movie[id:2000] {
    ... on Animation {
      title
    }
    ... on Fiction {
      title
      year
} } }
```

The query on the left is not grounded because the selection set under field `movie` contains two selections of different kind. The query on the right represents a grounded version thereof.

Definition 4.1 ([18]). A GraphQL selection set $\overline{\sigma}$ is in *ground-typed normal form* if it can be generated by the following grammar, where `t` is an object type:

$$\begin{array}{lcl} \overline{\sigma} & ::= & \chi \ldots \chi \\ & | & \psi \ldots \psi \\ \\ \psi & ::= & \texttt{... on t } \{\chi \ldots \chi\} \end{array} \qquad \begin{array}{lcl} \chi & ::= & \texttt{f}[\overline{\alpha}] \\ & | & \texttt{a:f}[\overline{\alpha}] \\ & | & \texttt{f}[\overline{\alpha}]\,\{\overline{\sigma}\} \\ & | & \texttt{a:f}[\overline{\alpha}]\,\{\overline{\sigma}\} \end{array}$$

A GraphQL query φ is in *ground-typed normal form* if its selection set is in ground-typed normal form.

Non-redundancy. Informally, the notion of non-redundancy further constraints that of groundedness by forbidding queries that induce a repeated evaluation of selections. For example, consider the two queries below:

```
// Redundant query
query {
  movie[id:2000] {
    ... on Fiction {
      title
      title
    }
    ... on Fiction {
      year
} } } }
```

```
// Non-redundant query
query {
  movie[id:2000] {
    ... on Fiction {
      title
      year
    }
  }
}
```

The query on the left is redundant for two reasons: it requests the field `title` twice, and it contains multiple inline fragments with the same type condition. Conversely, the query on the right is semantically equivalent and non-redundant.

Definition 4.2 (Adapted from [18]). A GraphQL selection set $\overline{\sigma}$ is *non-redundant* if:

- there is at most one field selection with a given response name,
- at most one inline fragment with a given type condition, and
- subselections are non-redundant.

A GraphQL query φ is *non-redundant* if its selection set is non-redundant.

4.2 Defining Normalization

The normalization procedure of selection sets is described in Figure 8. It is parametrized by a type in scope and acts as follows. Whenever a field selection is encountered (2-3), normalization removes any field that shares the same response name. This step ensures the non-redundancy of the resulting selection set. However, in order not to lose information during filtering, normalization collects fields with the same response name and merges their subselections in the first occurrence (3). This also serves to preserve the order of selections.

Finally, to obtain a selection in ground-typed normal form, normalization performs two separate steps, depending on whether the selection is a field (3) or an inline fragment (4). For the former, the process either directly normalizes the subselections or wraps them with inline fragments, based on the field's type. For the latter, the process either removes fragments or lifts their subselections, depending on whether they apply to the given type in scope.

To normalize a query, we simply normalize its selection set, setting the initial type in scope to the query root type of the underlying schema.

Example. We illustrate the normalization procedure with the example below.

```
// Query not in normal form
query {
  ... on Query {
    movie[id:2000] {
      title
    }
  }
  movie[id:2000] {
    title
} }
```

```
// Normalized query
query {
  movie[id:2000] {
    ... on Animation {
      title
    }
    ... on Fiction {
      title
    }
} }
```

On the left we have the original query and on the right its normalized version. Subselections from fragment with type condition `Query` are lifted and the multiple occurrences of field `movie` are merged into a single occurrence. Since the type of the field `movie` is the abstract type `Movie`, the subselections are wrapped in fragments for each concrete object subtype, namely `Animation` and `Fiction`. The normalized query is both non-redundant and in ground-typed normal form.

4.3 Correctness and Semantic Preservation

We now establish two fundamental results about the normalization procedure. The first result states that the normalization procedure is *correct* in that it does indeed produce queries in normal form.

$\boxed{\mathcal{N}_{ts}(\cdot) : \mathit{seq\ Selection} \to \mathit{seq\ Selection}}$

(1) $$\mathcal{N}_{ts}(\cdot) = [\cdot]$$

(2) $$\mathcal{N}_{ts}(\mathtt{f}[\overline{\alpha}] :: \overline{\sigma}) = \mathtt{f}[\overline{\alpha}] :: \mathcal{N}_{ts}(\mathit{filter}(\mathtt{f}, \overline{\sigma}))$$

(3) $$\mathcal{N}_{ts}(\mathtt{f}[\overline{\alpha}]\ \{\overline{\beta}\} :: \overline{\sigma}) = \begin{cases} \mathtt{f}[\overline{\alpha}]\ \{\mathcal{N}_{\mathit{ftype}(ts,\mathtt{f})}(\overline{\beta} +\!\!+ \mathit{merge}(\mathit{collect}(ts, \mathtt{f}, \overline{\sigma})))\} :: \mathcal{N}_{ts}(\mathit{filter}(\mathtt{f}, \overline{\sigma})) & \text{if } \mathit{is\_object\_type}(\mathit{ftype}(ts, \mathtt{f})) \\ \mathtt{f}[\overline{\alpha}]\ \{\mathit{map}\ (\lambda t_i.\ \ldots\ \mathtt{on}\ \ t_i\ \ \{\mathcal{N}_{t_i}(\overline{\beta} +\!\!+ \mathit{merge}(\mathit{collect}(ts, \mathtt{f}, \overline{\sigma})))\})\ \mathit{get\_possible\_types}(ts)\} :: \mathcal{N}_{ts}(\mathit{filter}(\mathtt{f}, \overline{\sigma})) & \text{otherwise} \end{cases}$$

(4) $$\mathcal{N}_{ts}(\ldots\ \mathtt{on}\ \mathtt{t}\ \{\overline{\beta}\} :: \overline{\sigma}) = \begin{cases} \mathcal{N}_{ts}(\overline{\beta} +\!\!+ \overline{\sigma}) & \text{if } \mathit{fragment\_type\_applies}(ts, \mathtt{t}) \\ \mathcal{N}_{ts}(\overline{\sigma}) & \text{otherwise} \end{cases}$$

Figure 8. Normalization procedure for GraphQL selections.
is_object_type checks whether the given type is an object type in the schema. *get_possible_types* get all object subtypes of the given type.

```
Lemma normalized_selections_are_in_nf
  (s : wfGraphQLSchema) (ts : Name) (σs : seq Selection) :
    are_in_normal_form s (normalize_selections s ts σs).

Corollary normalized_query_is_in_nf
  (s : wfGraphQLSchema) (φ : query) :
    is_in_normal_form s (normalize s φ).
```

The proof of the main lemma proceeds by well-founded induction over the size of the selection set and relies on some auxiliary lemmas about subtyping. Each of the two conditions (groundedness and non-redundancy) are established separately.

The second result states that the normalization procedure is *semantics-preserving* in that a normalized query has the same evaluation semantics as the original query from which it was derived. Formally, this requires proving that evaluating a query and its normalized version from the root node of a graph both yield the same result. To be able to establish this equivalence, we must however consider a generalized statement that quantifies over every node of the graph.

```
Lemma normalize_selections_preserves_semantics
  (s : wfGraphQLSchema) (g : conformedGraph s)
  (σs : seq Selection) (u : node) :
    u \in g.(nodes) ->
    execute_selection_set s check_scalar g coerce
      u (normalize_selections s u.(ntype) σs) =
    execute_selection_set s check_scalar g coerce
      u σs.

Corollary normalize_preserves_query_semantics
  (s : wfGraphQLSchema) (g : conformedGraph s)
  (φ : query) :
    execute_query s check_scalar g coerce (normalize s φ) =
    execute_query s check_scalar g coerce φ.
```

Similarly to the previous result, the proof of the main lemma above proceeds by well-founded induction over the size of the selection set.

4.4 Simplified Semantics of Normalized Queries

One of the main properties of queries in normal form is that they produce non-redundant responses. This in turn permits defining a simplified evaluation function which H&P crucially use to establish their complexity results. However, H&P do not formally prove that this simplified semantics is equivalent to the original, when considering normalized queries.

We define the simplified semantics $(\!|\cdot|\!)_{\mathcal{G}}$ of H&P as shown in Figure 9 and prove that, for queries in normal form, both $(\!|\varphi|\!)_{\mathcal{G}}$ and $[\![\varphi]\!]_{\mathcal{G}}$ produce the same response.

```
Lemma exec_sel_eq_simpl_exec
  (s : wfGraphQLSchema) (g : conformedGraph s)
  (σs : seq Selection) (u : node) :
    are_in_normal_form s σs ->
    execute_selection_set s check_scalar g coerce
      u σs =
    simpl_execute_selection_set s check_scalar g coerce
      u σs.

Corollary exec_query_eq_simpl_exec
  (s : wfGraphQLSchema) (g : conformedGraph s)
  (φ : query) :
    is_in_normal_form s φ ->
    execute_query s check_scalar g coerce φ =
    simpl_execute_query s check_scalar g coerce φ.
```

The proof is once again performed by induction over the size of the selection set.

4.5 Observations

Mechanizing normalization and the simplified semantics, as well as associated properties and proofs, led us to identify some issues in H&P's definitions. While these are admittedly minor, they confirm the value of mechanized formalization.

First, some queries are considered non-redundant by H&P although they actually produce redundant results. A simple example is the following query:

```
query {
  movie[id:2000] { title }
  movie:movie[id:2000] { title } }
```

$$(\!|\cdot|\!)^{u}_{\mathcal{G}} : \mathit{seq\ Selection} \to \mathit{GraphQLResponse}$$

$$(1)\quad (\!|\cdot|\!)^{u}_{\mathcal{G}} = [\cdot]$$

$$(2)\quad (\!|\mathtt{f}[\overline{\alpha}] :: \overline{\sigma}|\!)^{u}_{\mathcal{G}} = \mathtt{f}{:}(\mathit{complete\_value}(\mathit{ftype}(u.\mathit{type}, \mathtt{f}), u.\mathit{property}(\mathtt{f}[\overline{\alpha}]))) :: (\!|\overline{\sigma}|\!)^{u}_{\mathcal{G}}$$

$$(3)\quad (\!|\mathtt{f}[\overline{\alpha}]\ \{\overline{\beta}\} :: \overline{\sigma}|\!)^{u}_{\mathcal{G}} = \begin{cases} \mathtt{f}{:}[\mathit{map}\ (\lambda v_i.\ \{(\!|\overline{\beta}|\!)^{v_i}_{\mathcal{G}}\})\ u.\mathit{neighbors}_{\mathcal{G}}(\mathtt{f}[\overline{\alpha}])] :: (\!|\overline{\sigma}|\!)^{u}_{\mathcal{G}} \\ \qquad\qquad \text{if } \mathit{ftype}(u.\mathit{type}, \mathtt{f}) = \mathit{list} \text{ and } \{v_1, \ldots, v_k\} = \{v_i \mid (u, \mathtt{f}[\overline{\alpha}], v_i) \in \mathit{edges}(\mathcal{G})\} \\ \mathtt{f}{:}\{(\!|\overline{\beta}|\!)^{v}_{\mathcal{G}}\} :: (\!|\overline{\sigma}|\!)^{u}_{\mathcal{G}} \qquad \text{if } \mathit{ftype}(u.\mathit{type}, \mathtt{f}) \neq \mathit{list} \text{ and } (u, \mathtt{f}[\overline{\alpha}], v) \in \mathit{edges}(\mathcal{G}) \\ \mathtt{f}{:}\mathtt{null} :: (\!|\overline{\sigma}|\!)^{u}_{\mathcal{G}} \qquad \text{if } \mathit{ftype}(u.\mathit{type}, \mathtt{f}) \neq \mathit{list} \text{ and } \nexists v \text{ s.t. } (u, \mathtt{f}[\overline{\alpha}], v) \in \mathit{edges}(\mathcal{G}) \end{cases}$$

$$(4)\quad (\!|\ldots \mathtt{on\ t}\ \{\overline{\beta}\} :: \overline{\sigma}|\!)^{u}_{\mathcal{G}} = \begin{cases} (\!|\overline{\beta} +\!\!+ \overline{\sigma}|\!)^{u}_{\mathcal{G}} & \text{if } \mathit{fragment\_type\_applies}(u.\mathit{type}, \mathtt{t}) \\ (\!|\overline{\sigma}|\!)^{u}_{\mathcal{G}} & \text{otherwise} \end{cases}$$

Figure 9. Simplified semantics for selections in normal form, adapted from [18].

which produces two repeated values. This occurs because H&P's definition of non-redundancy does not consider the case when an unaliased field and an aliased field share the same response name. This cornercase is properly handled in GraphCoQL, by grouping fields by their response names. We discovered this case when proving the equivalence of the simplified semantics.

Second, using H&P's equivalence rules, some queries cannot be normalized. For instance:

```
query {
  movie[id:2000] { title }
  ... on Query { movie[id:2000] { title } } }
```

In H&P there are no rules that explicitly consider and use the type in scope to transform selections. In this example, by considering the `Query` type, either the fragment's contents could be lifted or the field could be wrapped in a fragment, and then the existing rules would apply.

We uncovered this issue when working on the semantics preservation proof of the normalization function. In fact, at the time we were formalizing the query semantics following H&P's approach. Working on fixing this issue and the others discussed in §3.5 led us to change the approach of the query semantics, and to adjust the definition of responses.

5 Discussion

Limitations. Currently, GraphCoQL captures the core of GraphQL, but not the full specification. While this is already valuable to study key aspects of GraphQL—as illustrated in §4—more work is needed to fully cover the specification. For the sake of exhaustiveness, we now list all missing features, referring to the Spec sections [16] where they are defined:

- *Executable definitions:* mutations (§2.3), subscriptions (§2.3), fragments (§2.8).
- *Types and type operations:* Non-null types (§2.11-3.4.1-3.12), schema and type extensions (§3.2.2-3.4.3), argument default values (§3.6.1), input object types (§3.10), directives (§3.13).
- *Queries:* input object values (§2.9.8), variables (§2.10), variable and argument coercion (§6.1.2-6.4.1), normal and serial execution (§6.3.1), error handling (§6.4.4).
- *Introspection:* fully unsupported (§4).

Considering GraphCoQL's current status, integrating some of these missing features should be straightforward: fragments, schema and type extensions, argument default values, input object types and values. For directives, some of the builtin cases have a clear semantics and should be simple to include as well. However, custom directives would be quite hard to integrate because they arbitrarily alter the semantics of queries. It is not entirely clear how these should be modeled and how normalization would be affected by it, regardless of GraphCoQL's current implementation. Similarly, it is not clear how mutation and subscription should be modeled, and what reasoning can be performed on these operations.

Contrarily to custom directives, mutation and subscription, features such as variables, non-null values and error handling, can be properly modeled and implemented, though at some engineering cost. Their implementation would require several changes to GraphCoQL, such as including environments and error propagation, among others.

Supporting introspection would be challenging, given the amount of details involved. Many adjustments would probably be based on case analyzing string values in search of certain patterns, in order to identify introspection selections.

Note that many of the challenges in implementing these features arise from either the fact that it is not entirely clear how to model them (without recurring to generalizations that might compromise valuable reasoning), or that most definitions require nested well-founded recursion as result of the tree structure of queries, possibly increasing the difficulty in reasoning about them.

Trustworthiness. We have striven to establish a direct "eyeball correspondence" between GraphCoQL and the Spec whenever possible—though this correspondence has not

(yet) been as seriously and systematically established as in other language formalization efforts such as JSCert [6] and CoqR [7]. In particular, the GraphCoQL definitions closely follow the algorithmic definitions of the Spec. Also, whenever it makes sense, we explicitly reference the corresponding specific sections from the Spec, inside of comments in the Coq definitions.

As mentioned earlier, GraphCoQL adopts a graph data model in order to give semantics to data access. This approach, which follows H&P, goes beyond what the Spec mandates. Indeed, the Spec defers to *resolvers* to give meaning to data accesses, which are essentially arbitrary pieces of code. However, reasoning about GraphQL requires some sensible data model. In this respect, GraphCoQL follows H&P almost literally, while the query evaluation algorithm of GraphCoQL can be traced closely to Spec.

Validation. In order to further validate that GraphCoQL adequately captures the semantics of GraphQL, we implemented several examples, coming from different sources. In all examples we define the values used in arguments and properties of nodes as elements of an inductive type, which wraps standard Coq types such as integers and strings, and a coercion function to transform these values into the corresponding JSON format.

First, we implement all of the examples used in the Spec validation section (cf. §5 [16]), for features that GraphCoQL currently supports. These correspond to tests over fields, such as valid definition in given types in scope, proper use of arguments and whether they are are type-compatible and renaming-consistent. Also, the examples include validation of inline fragments and whether they can be used in certain contexts.

Second, we implemented the main example used in H&P, from its schema to its graph, query and corresponding response. Note that this example is quite close to the running example of this paper.

Finally, we also implemented the Star Wars example defined in the reference implementation of GraphQL,[12] up to the currently-supported features in GraphCoQL. For instance, we include the complete schema definition except for the `secretBackstory` field, which resolves to an error, as these are not yet supported.

6 Related Work

To the best of our knowledge, the only formalization efforts around GraphQL are H&P [18] and [17], which we have already discussed. The rest of the GraphQL literature focuses on practical issues such as creating GraphQL services and migrating REST-based web services GraphQL [10, 25, 26], automatic migration [28], and testing techniques [24]. A couple of empirical studies analyze the structure of GraphQL schemas in commercial and open-source projects [19, 27], shedding interesting insights on GraphQL in practice.

Despite being used mostly for web services, there are efforts to extend notions used in GraphQL to other areas of database specification and querying. Hartig and Hidders [17] use the GraphQL schema definition DSL to define the structure of property graphs, which can be linked to similar efforts to define the structure of graph databases [9]. Taelman et al. study the transformation of GraphQL queries to SPARQL [23]; however we are not aware of any mechanized formalization of SPARQL.

The mechanized formalization of data management systems has received a lot of attention in the traditional relational data model [3], SQL and its semantics [2, 5, 11, 12], as well as related query languages and engines [1, 4]. Coq is the proof assistant of choice for all these efforts. The tree-based nature of GraphQL queries and response differs significantly from the tuple-based semantics in traditional query languages, requiring different models and techniques. Doczkal and Pous [14] develop a mechanization of graph theory in Coq, including simple graphs, digraphs, and their properties. Their work could possibly be extended to deal with property graphs, and used for GraphCoQL. Bonifati et al. [8] build a Coq incremental graph view maintenance and evaluation engine; they experimentally assessed it on synthetic graphs generated from real-world schemas. The engine supports Regular Datalog queries and does not entirely fit the GraphQL setting; it could however serve as a base to extend GraphCoQL with mutation.

7 Conclusion

We have presented GraphCoQL, the first mechanized formalization of GraphQL, implemented in the Coq proof assistant. GraphCoQL currently covers most of the schema definition DSL, the query definition language, validity checking, and the query semantics over a graph data model. We study the query transformation proposed and exploited by Hartig and Pérez to establish their complexity results. Specifically, we provide an algorithmic definition of query normalization, proving it correct and semantics preserving. In the process we uncover and address some minor issues in the original definitions.

This work is a first step towards a mechanization of all of GraphQL, leaving several open venues for future work. The most pressing are supporting mutation, directives and non-null types, and experimenting with extraction in order to (ideally) derive a reference implementation directly from the mechanized specifications. We anticipate that extending GraphCoQL to support new features should be fairly straightforward. More engineering work needs to be done to adequately prepare the Coq development for such extensions, in particular through better modularity and automation. Finally, we would like to study a more abstract

[12] https://github.com/graphql/graphql-js/tree/master/src/__tests__

evaluation function that is not tied to the graph data model, which should then be derivable as a specific instance.

References

[1] Auerbach, J. S., Hirzel, M., Mandel, L., Shinnar, A., and Siméon, J. Q*cert: A platform for implementing and verifying query compilers. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 1703–1706.

[2] Benzaken, V., and Contejean, E. A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019* (2019), pp. 249–261.

[3] Benzaken, V., Contejean, E., and Dumbrava, S. A Coq formalization of the relational data model. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings* (2014), pp. 189–208.

[4] Benzaken, V., Contejean, E., and Dumbrava, S. Certifying standard and stratified Datalog inference engines in Ssreflect. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings* (2017), pp. 171–188.

[5] Benzaken, V., Contejean, E., Keller, C., and Martins, E. A Coq formalisation of SQL's execution engines. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings* (2018), pp. 88–107.

[6] 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.

[7] 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.

[8] Bonifati, A., Dumbrava, S., and Arias, E. J. G. Certified graph view maintenance with regular Datalog. *TPLP 18*, 3-4 (2018), 372–389.

[9] Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., and Voigt, H. Schema validation and evolution for graph databases. *CoRR abs/1902.06427* (2019).

[10] Bryant, M. GraphQL for archival metadata: An overview of the EHRI GraphQL API. In *2017 IEEE International Conference on Big Data (Big Data)* (2017), IEEE, pp. 2225–2230.

[11] Chu, S., Cheung, A., and Suciu, D. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proceedings of VLDB 11*, 11 (2018), 1482–1495.

[12] Chu, S., Weitz, K., Cheung, A., and Suciu, D. HoTTSQL: proving query rewrites with univalent SQL semantics. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (2017), pp. 510–524.

[13] development team, T. C. The Coq proof assistant. https://coq.inria.fr/, 1984. [Online; accessed 2019].

[14] Doczkal, C., and Pous, D. Graph theory in Coq: Minors, treewidth, and isomorphisms, May 2019. Available at https://hal.archives-ouvertes.fr/hal-02127698.

[15] Gonthier, G., Mahboubi, A., and Tassi, E. A small scale reflection extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2016.

[16] GraphQL Foundation. GraphQL specification. https://graphql.github.io/graphql-spec/June2018/, 2018.

[17] 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.

[18] Hartig, O., and Pérez, J. Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference* (Republic and Canton of Geneva, Switzerland, 2018), WWW '18, International World Wide Web Conferences Steering Committee, pp. 1155–1164.

[19] Kim, Y. W., Consens, M. P., and Hartig, O. An empirical analysis of GraphQL API schemas in open code repositories and package registries. In *AMW* (2019).

[20] Mahboubi, A., and Tassi, E. Mathematical components. https://math-comp.github.io/mcb/, 2018.

[21] Richardson, L., Amundsen, M., Amundsen, M., and Ruby, S. *RESTful Web APIs: Services for a Changing World.* " O'Reilly Media, Inc.", 2013.

[22] Sozeau, M., and Mangin, C. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang. 3*, ICFP (Aug. 2019), 86:1–86:29.

[23] Taelman, R., Sande, M. V., and Verborgh, R. GraphQL-LD: Linked data querying with GraphQL. In *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018.* (2018).

[24] Vargas, D. M., Blanco, A. F., Vidaurre, A. C., Alcocer, J. P. S., Torres, M. M., Bergel, A., and Ducasse, S. Deviation testing: A test case generation technique for GraphQL APIs.

[25] Vázquez-Ingelmo, A., Cruz-Benito, J., and García-Peñalvo, F. J. Improving the OEEU's data-driven technological ecosystem's interoperability with GraphQL. In *Proceedings of the 5th International Conference on Technological Ecosystems for Enhancing Multiculturality* (2017), ACM, p. 89.

[26] Vogel, M., Weber, S., and Zirpins, C. Experiences on migrating RESTful web services to GraphQL. In *International Conference on Service-Oriented Computing* (2017), Springer, pp. 283–295.

[27] Wittern, E., Cha, A., Davis, J. C., Baudart, G., and Mandel, L. An empirical study of GraphQL schemas, 2019.

[28] Wittern, E., Cha, A., and Laredo, J. A. Generating GraphQL-wrappers for REST(-like) APIs. In *Web Engineering* (Cham, 2018), T. Mikkonen, R. Klamma, and J. Hernández, Eds., Springer International Publishing, pp. 65–83.

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