GraphCoQL
A mechanized formalization of GraphQL in Coq

Tomás Díaz       Federico Olmedo       Éric Tanter

Millennium Institute
Foundational Research on Data

Certified Programs and Proofs
New Orleans, USA – January 2020
GraphQL

Language for specifying the interfaces of web data services and their query mechanism
Language for specifying the interfaces of web data services and their query mechanism
GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```graphql
query {
  artist(id: 1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

HTTP GET

Cloud to Cloud Service
GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```graphql
query {
  artist(id:1000) {
    name
    artworks {
      role: ACTOR
      title
    }
  }
}
```

HTTP GET
GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```graphql
query {
  artist(id: 1000) {
    name
  }
}
```

HTTP GET
GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```graphql
query {
  artist(id:1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

HTTP GET
GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```graphql
query {
  artist(id:1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

HTTP GET

```json
{}
```
GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```graphql
query {
  artist(id:1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

HTTP GET

```
{
  "artist": {
    "name": "Tom Hanks",
    ...
  }
}
```
GraphQL

Language for specifying the interfaces of web data services and their query mechanism

```graphql
query {
  artist(id:1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

HTTP GET

```
{
  "artist": {
    "name": "Tom Hanks",
    "artworks": [
      {
        "title": "Toy Story",
      },
      {
        "title": "Forrest Gump",
      },
      ...
    ]
  }
}
```
Industry involvement with GraphQL

2012

FACEBOOK

2015+

[Logos of various companies and technologies]
Paper & pencil formalization to study complexity properties.

Figure 5: Semantics of a GraphQL query.
First language formalization [Hartig & Pérez, WWW’18]

Paper & pencil formalization to study complexity properties.

Missing proofs about fundamental properties

![Figure 5: Semantics of a GraphQL query.](image)
Our contribution

First *mechanized* formalization of *GraphQL* in the *Coq* proof assistant
Schema

Describes how data is structured and queried
Describes how data is structured and queried

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

```graphql
enum Role {
    ACTOR
    DIRECTOR
    WRITER
}
```

```graphql
union Artwork = Fiction | Animation | Book
```

```graphql
type Fiction implements Movie {
    …
}
```

```graphql
type Animation implements Movie {
    …
    style: Style
}
```

```graphql
type Book {
    …
}
```

```graphql
type Query {
    artist(id: ID): Artist
    movie(id: ID): Movie
}
```
Describes how data is structured and queried

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

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

enum Role {
  ACTOR
  DIRECTOR
  WRITER
}

union Artwork = Fiction | Animation | Book

type Fiction implements Movie {
  ...
}

type Animation implements Movie {
  ...
  style: Style
}

type Book {
  ...
}

type Query {
  artist(id: ID): Artist
  movie(id: ID): Movie
}
```
Describes how data is structured and queried

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

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

type Fiction implements Movie {
  ...
}

type Animation implements Movie {
  ...
    style: Style
}
```
Describes how data is structured and queried

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

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

type Fiction implements Movie {
  ...
}

type Animation implements Movie {
  ...
    style: Style
}

enum Role {
  ACTOR
  DIRECTOR
  WRITER
}

type Query {
  artist(id: ID): Artist
  movie(id: ID): Movie
}
```
Describes how data is structured and queried

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

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

type Fiction implements Movie {
  ...
}

type Animation implements Movie {
  ...
    style: Style
}

enum Role {
  ACTOR
  DIRECTOR
  WRITER
}

union Artwork = Fiction
               | Animation
               | Book

type Book { ...
}
Describes how data is structured and queried

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

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

type Fiction implements Movie {
  ...
}

type Animation implements Movie {
  ...
    style: Style
}

enum Role {
  ACTOR
  DIRECTOR
  WRITER
}

union Artwork = Fiction |
  Animation |
  Book

type Book { ... }

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

**Schema**

- **object type**
  - `type Artist {` ...
  - `interface Movie {` ...
  - `type Fiction implements Movie {` ...
  - `type Animation implements Movie {` ...

- **interface type**
  - `enum Role {` ...
  - `union Artwork = Fiction | Animation | Book`

- **enumeration type**
  - `type Book { ... }

- **union type**
  - `type Query {` ...

- **entry points for querying the dataset**
  - `...`
Describes how data is structured and queried

**TypeDef**
- ScalarTypeDef
- ObjectTypeDefinition
- InterfaceTypeDef
- UnionTypeDef
- EnumTypeDef
- InputObjectTypeDefinition

```plaintext
Inductive TypeDefinition : Type :=
| ScalarTypeDef (name : Name)
| ObjectTypeDefinition (name : Name)
  (interfaces : seq Name)
  (fields : seq FieldDefinition)
| InterfaceTypeDef (name : Name)
  (fields : seq FieldDefinition)
| UnionTypeDef (name : Name)
  (members : seq Name)
| EnumTypeDef (name : Name)
  (members : seq EnumValue).
```
Datasets are modeled as directed property graphs, with labeled edges and typed nodes.
Datasets are modeled as directed property graphs, with labeled edges and typed nodes.

**Artist**

- **id**: 1000
- **name**: “Tom Hanks”
Datasets are modeled as directed property graphs, with labeled edges and typed nodes.
Datasets are modeled as directed property graphs, with labeled edges and typed nodes.
Datasets are modeled as directed property graphs, with labeled edges and typed nodes.
Datasets are modeled as directed property graphs, with labeled edges and typed nodes.

**Graph data model**

Datasets are modeled as directed property graphs, with labeled edges and typed nodes.
Queries are evaluated by traversing the graph and collecting nodes’ properties.

**Query**

```json
query {
  artist(id:1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

**Response (à la JSON)**

```json
{
}
```
Query evaluation

Queries are evaluated by traversing the graph and collecting nodes’ properties

Query

```graphql
query {
  artist(id:1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

Response (à la JSON)

```json
{
}
```

Dataset
Query evaluation

Queries are evaluated by traversing the graph and collecting nodes’ properties

Query

```json
query {
  artist(id: 1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

Response (à la JSON)

```json
{
  "artist": {
  ...
  }
}
```
Query evaluation

Queries are evaluated by traversing the graph and collecting nodes’ properties

Query

```json
query {
  artist(id:1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

Response (à la JSON)

```json
{
  "artist" : {
    "name" : "Tom Hanks",
  }
}
```
Query evaluation

Queries are evaluated by traversing the graph and collecting nodes’ properties

Query

```
query {
  artist(id:1000) {
    name
    artworks(role: ACTOR) {
      title
    }
  }
}
```

Response (à la JSON)

```
{
  "artist" : {
    "name" : "Tom Hanks",
    "artworks" : [
      {
        "title" : ...
      },
      {
        "title" : ...
      }
    ]
  }
}
```
Query evaluation is not compositional
Query evaluation is not compositional

```graphql
query {
  artist(id:1000) {
    name
  }
  artist(id:1000) {
    artworks(role: ACTOR) {
      title
    }
  }
}
```
Query evaluation is not compositional

```graphql
query {
  artist(id: 1000) {
    name
  }
  artist(id: 1000) {
    artworks(role: ACTOR) {
      title
    }
  }
}
```
Query evaluation is not compositional

query {
  artist(id:1000) {
    name
  }
  artist(id:1000) {
    artworks(role: ACTOR) {
      title
    }
  }
}

{
  "artist" : {
    "name" : "Tom Hanks",
  },
  "artist" : {
    "artworks" : [
      {
        "title": ...
      },
      {
        "title": ...
      }
    ]
  }
}
Query evaluation is not compositional

Selections are “factored-out” in between the recursive calls

```perl
query {
  artist(id:1000) {
    name
  }
  artist(id:1000) {
    artworks(role: ACTOR) {
      title
    }
  }
}
```
Query evaluation is not compositional

Selections are “factored-out” in between the recursive calls

This makes reasoning significantly harder

```
query {
  artist(id:1000) {
    name
  }
  artist(id:1000) {
    artworks(role: ACTOR) {
      title
    }
  }
}
```

```
{
  "artist" : {
    "name" : "Tom Hanks",
    "artworks" : [
      {
        ...
      },
      {
        ...
      }
    ]
  }
}
```
Application
Queries admit a normal form that can be evaluated purely compositionally and significantly simplifies reasoning.
Queries admit a **normal form** that can be evaluated purely compositionally and significantly simplifies reasoning

**But....**

👎 Normalization procedure not provided
👎 No correctness proof
Query normalization
Query normalization

- Certified normalization algorithm

\[\text{Theorem normalized_query_is_in_nf :} \]
\[\forall (\varphi : \text{query}) (s : \text{wfGraphQLSchema}),
\text{is_in_normal_form} s (\text{normalize} s \varphi).\]

\[\text{Theorem normalize_preserves_semantics :} \]
\[\forall (\varphi : \text{query}) (s : \text{wfGraphQLSchema}) (g : \text{conformedGraph} s),
\text{eval_query} (\text{normalize} s \varphi) g s = \text{eval_query} \varphi g s.\]
Query normalization

- Certified normalization algorithm

\textbf{Theorem} \texttt{normalized\_query\_is\_in\_nf}:
\[
\forall (\varphi : \text{query}) \text{ (s : \text{wfGraphQLSchema})},
\text{is\_in\_normal\_form} \text{ s (normalize s} \varphi).\]

\textbf{Theorem} \texttt{normalize\_preserves\_semantics}:
\[
\forall (\varphi : \text{query}) \text{ (s : \text{wfGraphQLSchema}) (g : \text{conformedGraph} s)},
\text{eval\_query (normalize s} \varphi) \text{ g s = eval\_query} \varphi \text{ g s}.\]

- Simplified evaluation for queries in normal form

\textbf{Theorem} \texttt{simpl\_eval\_correctness}:
\[
\forall (\varphi : \text{query}) \text{ (s : \text{wfGraphQLSchema}) (g : \text{conformedGraph} s)},
\text{is\_in\_normal\_form} \text{ s} \varphi \rightarrow
\text{eval\_query} \varphi \text{ g s = simpl\_eval\_query} \varphi \text{ g s}.\]
Formalization evaluation and details
Effectivity
Uncovered two issues in H&P formalization:

• Flawed definition of normal form
• Incomplete set of equivalence rules for normalization
Evaluation

Effectivity
Uncovered two issues in H&P formalization:

- Flawed definition of normal form
- Incomplete set of equivalence rules for normalization

Faithfulness
Validated with a series of examples from different sources:

- Examples (41) from the SPEC validation section*
- Star Wars example from GraphQL reference implementation
- Example used in H&P

* [https://graphql.github.io/graphql-spec/June2018/#sec-Validation](https://graphql.github.io/graphql-spec/June2018/#sec-Validation)
Conclusion

Contribution

• First mechanized formalization of GraphQL in the Coq proof assistant
• Certified query normalization algorithm
• Uncover issues in initial formalization [H&P, WWW18]
Conclusion

Contribution

• First mechanized formalization of GraphQL in the Coq proof assistant
• Certified query normalization algorithm
• Uncover issues in initial formalization [H&P, WWW18]

Future work

• Further GraphQL features
• Extraction (certified reference implementation)
• More general data models
Conclusion

Contribution

- First mechanized formalization of GraphQL in the Coq proof assistant
- Certified query normalization algorithm
- Uncover issues in initial formalization [H&P, WWW18]

Future work

- Further GraphQL features
- Extraction (certified reference implementation)
- More general data models

Thanks!