Mapping GraphQL to the Haskell Type System

Bachelor Thesis

Daviti Nalchevanidze

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences (MIN)
B.Sc. Human-Computer Interaction (HCI)

08.03.2021

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Table of Contents

Introduction

Current challenges

Modern web applications must be adaptable, available anytime, anywhere, and support multiple devices [1]. This poses complexity, performance, and reliability challenges for development.

Haskell

Haskell has proven to be safe and capable of managing the complexity.

GraphQL

GraphQL enables clients to query only the data they need, while servers can address multiple devices.

Motivation

GraphQL Haskell libraries

Existing GraphQL Haskell libraries either do not provide sufficient type safety or straightforward mapping, making them cumbersome to use.

Objectives

Our goal is to provide a library that requires minimal effort to use and provides a good developer experience while still guaranteeing reliability.

Haskell

Haskell is a non-strict, purely functional language with static typing and algebraic data types. Many developers think that Haskell programs look nice [2].

Haskell is lazy

> Laziness is a primary concern in the design of Haskell [2].

Haskell is pure

> The lazy evaluation requires a pure design since a function call can no longer guarantee the reliable execution of the side-effects. However, Haskell permits side effects with monads [2].

Algebraic Data Types and Records I

An algebraic data type is the sum of one or more alternatives, where each alternative is a product of zero or more fields [2].

Listing 1: Algebraic data types [2]

```
data Maybe a = Nothing | Just a
```

Records simplify programming with data structures because fields are accessed by name rather than position [3].

Listing 2: Haskell records

```
data Deity = Deity
  { name :: String,
    power :: String
  }
```

Algebraic Data Types and Records II

Listing 3: Haskell record values

```
deity :: Deity
deity =
  Deity
    { power = "Shapeshifting",
      name = "Morpheus"
    }
```

Algebraic data types and records and are not extensible [3, 4].

GraphQL

What is GraphQL?

GraphQL is a query language for APIs. It was developed by Facebook and published in 2016 [5, 6].

GraphQL as a current trend

It has a rich open-source ecosystem and the trust of companies in various industries such as GitHub, Netflix, PayPal, and others [7].

Typed query language

It is a strongly typed language. The type system provides a solid contract between client and server, which eliminates possible invalid requests [8].

Advantages of GraphQL

Performance and efficiency improvement

- reduces numbers of API calls [9].
- reduces the response size [9].

API versioning

- new fields added to a type do not result in client changes [9].

Facilitates rapid product development

- familiar syntax and semantics [10].
- real-time query validation and auto-completion [10, 9].
- Instant API exploration through introspection [9].

GraphQL Language Components

Schema definition language (SDL)

> a domain-specific language, used by the server to define the schema [9].

Query language

> a domain-specific language, used by the client to retrieve data from the server [5].

Resolver functions

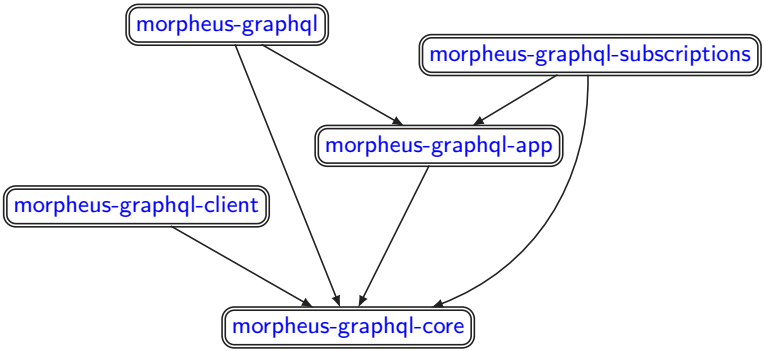
> Resolver functions return values from the underlying data structures based on client queries [9].

Requirements

- **Maintainability**: a Maintainable software can adapt to continuous changes.
  - Low boilerplate
  - Familiarity
  - Modularity
- **Reliability**: We aim to avoid runtime failures.
  - Type safety
  - Compliance with GraphQL specifications
- **Efficiency**: The library must be lazy and efficient.

Architecture Overview

Figure: Dependency graph of Morpheus GraphQL packages



Design Decisions

- **Code-first approach**: A single type change automatically updates the resolver types and GraphQL schema.
- **Embeded domain specific language**: familiar syntax of the native language [11].
- **Datatype-generic programming**: eliminates boilerplate code, while guaranteeing safety [12, 13].
- **Monadic resolvers**: All side effects in Haskell are performed with monads.
- **Parameterized resolver types**: Parametric polymorphism allows the modular definition of resolver types, where the type parameter determines the allowed operations.

Schema Derivation

GraphQL models the schema as a graph, where nodes are types and fields are edges [9]. We use deep-first search and derive all schema types from the root node while using cycle checking to avoid loop cycles.

Mapping Built-in Types

- **Wrapping Types**:
  - **Non-Null**: inverse representation with Maybe
  - **List**: List
- **Scalar Types**:
  - **Int**: Int
  - **Float**: Double
  - **String**: Text
  - **Boolean**: Bool
  - **ID**: custom data type ID

Mapping Type Definitions

- **Enums**: data types with only empty constructors.
- **Input object and field arguments**: data types with non-empty single constructors (records).
- **Objects**: We represent objects with single constructor parameterized data types (records), where fields can take monadic functions, with the monadic type bound to the type parameter.
- **Unions**: data types with multiple constructors, where at least one alternative is non-empty.
  - We create a new object for each constructor.
  - We unpack constructors, where the name is the concatenation of the type constructor name and the referencing type name.
  - We assign a field with unit type to each empty constructor.

Contributions

- **@Pygmalion**: library name and first inspirations.
- **@krisajenkins**: parser optimization and parameterized resolver types.
- **@theobat**: custom wrapping types.
- **many others**: 27 contributors.

Morpheus GraphQL Examples

Simple mapping example

`https://github.com/nalchevanidze/morpheus-graphql-uhh-example`

More sophisticated examples

- `https://github.com/nalchevanidze/morpheus-haxl-example`
- `https://github.com/morpheusgraphql/mythology-api`
- `https://github.com/dandoh/web-haskell-graphql-postgres-boilerplate`

Results

- **Maintainability**: The derivation approach is familiar and reduces boilerplate. The library architecture and parameterized resolver types allow modular API definitions.
- **Reliability**: The Haskell compiler guarantees type safety. The library fulfills all critical parts of the specifications. However, the interfaces and directives are not yet fully implemented.
- **Efficiency**: The library is lazy and only executes the resolvers necessary for the query.

Discussion I

Advantages

- The lazy and pure nature of Haskell frees developers from explicitly dealing with concurrency and laziness.
- easy-to-use interface while ensuring type safety.
- The library uses an intuitive design that can be quickly learned even by beginners.
- Importing schemas from SDL facilitates migration from other languages.

Disadvantages

- Not all GraphQL names can be represented with Haskell types and values.
- Since we use algebraic data types and records, we lose extensibility.

Discussion II

Alternative GraphQL Haskell libraries provide a more extensible schema definition than our approach. However, implementing resolvers for these schemas in our library is more convenient. Besides, our approach is more intuitive and requires less experience.

Outlook

- **Support interfaces**: Haskell does not provide interfaces, so we do not yet fully support them.
- **Support custom collections**: Sets and non-empty collections cannot be modeled in GraphQL because it provides only list types for collections.
- **Support directives**: GraphQL specification does not specify a particular implementation strategy for directives. [14].
- **Support input unions**: GraphQL does not support input unions [15].

Thanks!

Questions or suggestions?

d.nalchevanidze@gmail.com

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

References I

[1] Ardalis. *Characteristics of modern web applications*. Microsoft.com, Dec. 2020.

[2] Simon Peyton Jones. "A History of Haskell: being lazy with class". In: *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*. June 2007.

[3] Mark P Jones and Simon Peyton Jones. "Lightweight Extensible Records for Haskell". In: *Haskell Workshop*. Paris. ACM. ACM, Oct. 1999.

[4] Shayan Najd and Simon Peyton Jones. "Trees that grow". In: *Journal of Universal Computer Science (JUCS)* 23 (Jan. 2017), pp. 47–62.

[5] Olaf Hartig. "An Initial Analysis of Facebook's GraphQL Language". In: June 2017.

[6] *GraphQL*. Graphql.org, 2018.

[7] David Chaves-Fraga et al. "Exploiting Declarative Mapping Rules for Generating GraphQL Servers with Morph-GraphQL". In: *International Journal of Software Engineering and Knowledge Engineering* 30 (July 2020), pp. 1–19. DOI: 10.1142/S0218194020400070.

[8] Ying Guo, Fang Deng, and Xiudong Yang. "Design and Implementation of Real-Time Management System Architecture based on GraphQL". In: *IOP Conference Series: Materials Science and Engineering* 466 (Dec. 2018), p. 012015. DOI: 10.1088/1757-899X/466/1/012015.

References II

[9] Gleison Brito, Thais Mombach, and Marco Valente. "Migrating to GraphQL: A Practical Assessment". In: Jan. 2019. DOI: 10.1109/SANER.2019.8667986.

[10] Gleison Brito and Marco Valente. "REST vs GraphQL: A Controlled Experiment". In: Feb. 2020. DOI: 10.1109/ICSA47634.2020.00016.

[11] Benoit Bayol, Yuting Chen, and Paul-Henry Cournède. "Towards an EDSL to enhance good modelling practice for non-linear stochastic discrete dynamical models Application to plant growth models". In: July 2013. DOI: 10.5220/0004481101320138.

[12] Jeremy Gibbons. "Datatype-Generic Programming". In: vol. 4719. Nov. 2007, pp. 1–71. DOI: 10.1007/978-3-540-76786-2_1.

[13] Ralf Lämmel and Simon Peyton Jones. "Scrap your boilerplate: a practical approach to generic programming". In: *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*. ACM Press, Jan. 2003, pp. 26–37.

[14] GraphQL Tools. *Schema directives | GraphQL Tools*. Graphql-tools.com, 2021.

[15] GraphQL. *graphql/graphql-spec*. GitHub, Nov. 2020.

References III

[16] Ralf Hinze and Simon Peyton Jones. *Derivable type classes*. Tech. rep. NOTTCS-TR-00-1. Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI1990). Sept. 2000.

[17] José Magalhães et al. "Optimizing Generics Is Easy!" In: Jan. 2010, pp. 33–42. DOI: 10.1145/1706356.1706366.

[18] *GHC.Generics*. Haskell.org, 2011.

[19] Philip Wadler. "The Essence of Functional Programming". In: Prentice Hall, 1992, pp. 1–14.

Datatype-Generic Programming

Datatype-generic programming enables writing single functions that address various cases and types. [16]. It increases program reliability, reduces code duplication while guaranteeing safety [12, 17].

Listing 4: Generic Deriving

```
data Maybe a = Nothing | Just a
  deriving (Eq)
```

The generic derivation represents all types and values by generic representation types. The user can convert any types to their generic representation, operate on them, and convert them back to their original types [17, 18].

Monads

Haskell abstracts side effects with monads. monads are formed by a type constructor `M` and a pair of functions, `return` and `>>=` [2, 19]. The type `M a` is a computation that returns a value of type `a` and possibly performs some side effects [2]. The purpose of `return` and `>>=` is to push a value into computation and to evaluate a computation, yielding a value [19].
Since the monad allows the compiler to determine impure and pure operations, the Haskell language can use specific optimizations. However, the developer still has the freedom to define their execution order [2].

Mapping Positional Constructors

GraphQL fields must always have names [6]. That why we assume that a constructor without field selectors is enumerated like an array.

Listing 5: Positional Constructors

```
data Type
  = Cons1 Bool Int
  | Cons2 { _0 :: Bool, _1 :: Int }
```

Determining GraphQL Types in Haskell

1. **Scalars, Wrappers, Interfaces**: Derive scalar, wrapper, interface if the type has explicitly specified associated kinds.
2. **Field Arguments**: Derive field arguments if the type is used as an argument of the object field.
3. **Enums**: Derive enum if all data type constructors are empty.
4. **Objects**: Derive object if the type has a single non-empty constructor, and its parent is Object or Union.
5. **Input objects**: Derive input object if the type has a single non-empty constructor, and its parent is input object or Field Arguments.
6. **Unions**: Derive Union if the type has multiple constructors and its parent is Object or Union.
7. **Fail**: All other types are not supported.