

Creating Adaptive GraphQL Client for Github Repo Analysis Using Scala Macros

by

Ashesh Kumar Singh
Bachelor of Engineering, University of Mumbai, 2016

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2021

Chicago, Illinois

Defense Committee:

Mark Grechanik, Chair and Advisor
Ugo Buy
Jason Polakis

ACKNOWLEDGEMENTS

I express my sincere gratitude to my advisor, Prof. Mark Grechanik, for his aid, advice, and supervision in completing this thesis document and carrying out the related project. Besides that, I thank him for introducing me to functional programming and the Scala programming language.

I am also indebted to my thesis committee members, Prof. Ugo Buy and Prof. Jason Polakis. I thank them for so generously offering their time in review of this work. In the past, I thoroughly enjoyed their courses — CS540 (Advanced Topics in Software Engineering) by Prof. Buy and CS568 (Advanced Computer Security and Online Privacy) by Prof. Polakis. The lessons from those two subjects will stay with me for a long time.

Lastly, I recognize and applaud the support of my family, friends, and flatmates. They have given me the strength to keep persevering through my Master's study at UIC.

AS

TABLE OF CONTENTS

1	Introduction	1
1.1	Problem Statement	4
1.2	Contributions	5
2	Solution Overview	7
2.1	Task 1 — Finding Subgraphs	7
2.1.1	Source 1: Using GraphQL Schema	7
2.1.2	Source 2: Using GraphQL Response	8
2.1.3	Source 3: Using GraphQL Query	8
2.2	Task 2 — Generating Representations	8
2.3	Finding Subgraphs and Generating Representations	9
2.3.1	Approach 1: Through IDE Tooling	9
2.3.2	Approach 2: Through Low-level Bytecode Manipulation	9
2.3.3	Approach 3: Through Run-time Metaprogramming	9
2.3.4	Approach 4: Through Compile-time Metaprogramming	10
2.4	Ensuring GraphQL Query Correctness	10
2.5	Implementation Overview	10
3	GitHub’s GraphQL API	12
3.1	GraphQL Schema	12
3.2	GraphQL Query	14
4	Caliban-client Plugin	16
4.1	Setup	16
4.2	Code generation	16
4.3	Type-checked GraphQL Queries in Scala	19
4.4	Deserializing GraphQL Response to Scala Types	21
5	Scala Macros	24
5.1	Environment, Universes, and Mirrors	24
5.2	Abstract Syntax Tree	24
5.2.1	Subclasses of <code>TermTree</code>	28

5.2.2	Subclasses of <code>TypTree</code>	28
5.2.3	Subclasses of <code>SymTree</code>	28
5.2.4	Inspecting Trees	29
5.2.5	Traversing Trees	30
5.2.6	Manipulating Trees — Quasiquotes	31
5.3	Writing Macros	32
5.3.1	Def Macro — <code>log</code>	34
5.3.2	Annotation Macro — <code>benchmark</code>	36
6	APIs	39
6.1	What is an API?	39
6.2	Why do we need APIs?	40
6.3	Types of APIs	41
6.4	Web API Data Exchange Format — JSON	41
6.5	Web API Common Terms and Definitions	44
6.5.1	API Description	44
6.5.2	API Documentation	45
6.5.3	Data Description Specification	45
6.6	RESTful Web APIs	48
6.6.1	REST architectural constraints	49
6.6.2	Interpreting REST constraints	50
6.6.3	The OpenAPI Specification	50
6.7	RESTful vs GraphQL Web APIs	51
6.7.1	API Endpoints	53
6.7.2	Over-fetching	53
6.7.3	Under-fetching and N+1 Requests	53
7	Implementation	54
7.1	Hard-coding the <code>mapSelection</code> Annotation Macro	54
7.2	Parsing Constructor and Method Parameters	56
7.3	Generalizing the <code>mapSelection</code> Annotation Macro	61
7.4	Traversing <code>Select</code> ASTs	62
7.5	Traversing <code>Apply</code> ASTs	64
7.6	Traversing <code>Apply-TypeApply</code> ASTs	66

7.7	Traversing Apply-Apply-TypeApply ASTs	68
7.8	Accounting for Selection Combination ASTs	69
7.9	Testing the mapSelection Annotation Macro	71
8	Conclusion	74
8.1	Comparison with Existing Solutions	74
8.2	Limitations	75
	Cited Literature	77
	Vita	80

LIST OF FIGURES

1.1	Graph model for a social media domain	1
1.2	GraphQL query operation essentially retrieves a subgraph or tree	3
1.3	GraphQL query request and its response have similar structure	4
2.1	System overview	11
3.1	Graph model for partial GitHub GraphQL schema	13
3.2	GraphQL Object Type anatomy	14
3.3	GraphQL Query anatomy	15
5.1	All direct types of <code>TermTree</code>	27
5.2	All direct types of <code>TypTree</code>	27
5.3	All direct types of <code>SymTree</code>	27
6.1	<code>jsonschema2pojo</code> input	43
6.2	<code>jsonschema2pojo</code> output	44
7.1	Components of the <code>Select</code> AST	63
7.2	Components of the <code>Apply</code> AST	65
7.3	Components of the <code>Apply-TypeApply</code> AST	67
7.4	Components of the <code>Apply-Apply-TypeApply</code> AST	68
7.5	Components of a Compound Selection	70
8.1	False positive “missing type or symbol” warning in IntelliJ IDE	75

LIST OF ABBREVIATIONS

API	Application Programming Interface
REST	REpresentational State Transfer
ORM	Object-Relational Mapping
IDE	Integrated Development Environment
AST	Abstract Syntax Tree
DSL	Domain-Specific Language
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
CSV	Comma Separated Values
FTP	File Transfer Protocol
SMTP	Simple Mail Transfer Protocol
RAML	RESTful API Modeling Language
RADL	RESTful API Description Language
IDL	Interface Description Language
DTD	Document Type Definition
RMM	Richardson Maturity Model

SUMMARY

GraphQL is a data query language and specification originally developed by Facebook. There are GraphQL runtimes on both — client-side and server-side in several languages that implement the GraphQL Specification. Basic client implementations in general-purpose languages like Java/Javascript mostly deal with GraphQL queries as strings; hence, the language’s underlying type system remains underutilized. More sophisticated client implementations provide Domain Specific Language (DSL) based on some input GraphQL schema (e.g., GitHub) to present increased type safety. However, the mismatches between data schemata and how programs use the data elements known as impedance mismatch make programs written with DSLs challenging to update. The language data types often do not match the corresponding GraphQL schema types. These problems make the creation, maintenance, and evolution of GraphQL clients difficult. Addressing these problems means a significant investment of programmers’ time since they would have to manually map each type in the GraphQL Schema to a corresponding type in the programming language. Not only is this process error-prone, but it might not even be feasible if the input schema is large.

We address these issues by implementing a client that provides a custom DSL and automatically maps types from GraphQL schema to the corresponding types in the programs written using this DSL. As a result of both, we improve type-safety of GraphQL programs using our client and reduce the development effort on the user’s end. We accomplished this by using experimental Scala metaprogramming capabilities (macros). Using our client on GitHub’s GraphQL schema, it is possible to efficiently conduct studies such as vulnerability dependency analysis of software repositories.

1 Introduction

GraphQL is a data query language and specification originally developed by Facebook in 2012 before being open-sourced in 2015 and later donated to the non-profit Linux Foundation in 2018 [1, 2]. Many perceive it as a modern approach to designing Web APIs and an alternative to the more traditional, currently ubiquitous REST approach. Since its public debut in 2015, GraphQL has witnessed tremendous interest, including adoption by companies like GitHub, Intuit, PayPal, and the New York Times [3].

At its core, GraphQL is a way to model a business domain as a graph using a schema [4]. This idea is significant since a graph is a powerful means to represent relationships between a set of entities called nodes connected by edges. The figure 1.1 shows an example graph with three social media

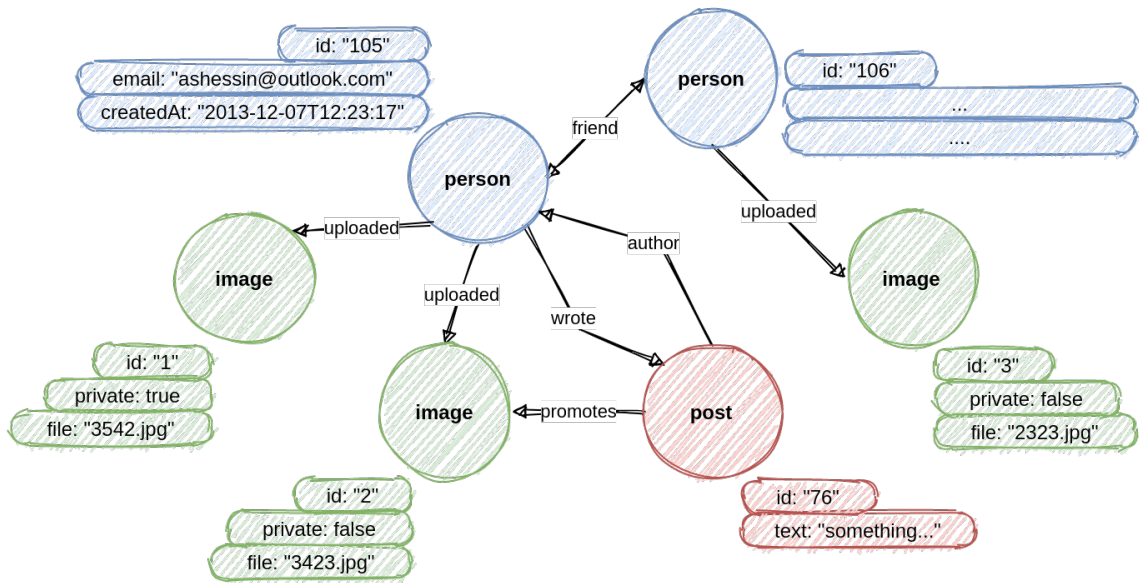


Figure 1.1: Graph model for a social media domain

domain entities as nodes: people, image, and post, along with the relationships as directed edges. An accompanying GraphQL schema would describe the GraphQL service's type system (constituting the entities and their relationships) in a language-agnostic way. This schema, in turn, serves as the contract to a client and defines how it may access data on the server. However, client application developers often find themselves at odds with this language-agnostic paradigm since they must design applications in a specific programming language. Such competing views give rise to the

impedance mismatch problem, more commonly seen in the object-relation world and addressed by Object-relational mapping (ORM) solutions [5].

Using a GraphQL query, clients can retrieve exactly the data they need from a GraphQL service endpoint. Typically, developers use a string to represent the query and write ad-hoc code to deserialize the retrieved data. This basic approach poses two fundamental problems for client application development:

- Statically checking query correctness is not possible.
- Deserialization of the query response is often complex.

To understand these limitations, consider a fictional social media GraphQL service and accompanying code fragment in listing 1.

```

1 String endpoint = "https://example.com/socialmedia/graphql?query=";
2 String query = "{perso(id:`105`){email,images{id,file}}}".replace("`", "");
3
4 JSONObject json = new JSONObject(IOUTils.toString(new URL(endpoint + query)));
5
6 List<Image> images = new ArrayList<>();
7 for (Object i : (JSONArray) json.query("/data/person/images")) {
8     images.add(new Image(
9         ((JSONObject) i).query("id").toString(),
10        ((JSONObject) i).query("id").toString()));
11 }
12 Person person = new Person("105", json.query("/data/person/email").toString(), images);
13
14 /* further processing */

```

Listing 1: Basic GraphQL client Java code

The example retrieves a user's email along with details of their uploaded photos as illustrated in figure 1.2, figure 1.3 and deserializes them for further processing. It shows:

- A string type that specifies the GraphQL query on line 2.
- Further, to create the `Person` and `Image` instance, the response is deserialized on lines 6–12.

Although the code in listing 1 will compile, it has some serious deficiencies:

- The execution would fail at runtime because the GraphQL query embedded as string is syntactically incorrect (`person` misspelled as `perso`). The compiler or the IDE will in no way warn the developer against this.

```
String query = "{person(id:`105`){email,images{id,file}}}".replace("`", ""); // fixed query
```

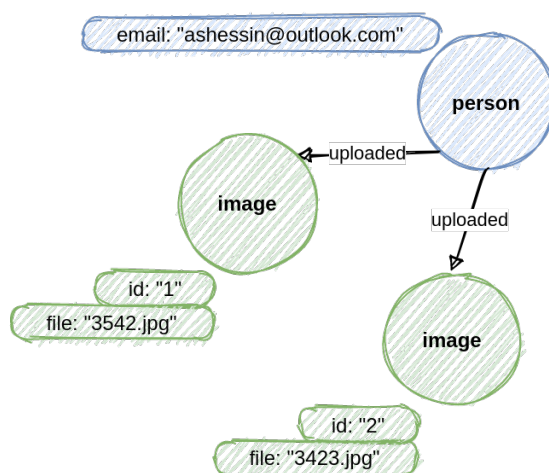


Figure 1.2: GraphQL query operation essentially retrieves a subgraph or tree

- Creating the `Person` and `Image` instances is a laborious process with additional scope for human error. For example, one can easily mix-up the positions for `id` and `email` arguments in the `Person`'s constructor. The compiler will not catch such a mistake since both arguments are of compatible type (i.e. `String`). This mix-up can cause errors at runtime down the road or may even go undetected.

```
public Person(String id, String email, List<Image> images) { /* constructor */ }
Person person1 = new Person("105", json.query("/data/person/email").toString(), images); //
↳ right
Person person2 = new Person(json.query("/data/person/email").toString(), "105", images); //
↳ wrong
```

- Maintaining the `Person` and `Image` classes required for precise deserialization is difficult as the GraphQL request-response might change to accommodate new business requirements or upstream GraphQL service changes. Fields might get added, removed, renamed, or even reordered. Any such change would demand a corresponding change to the existing classes or the creation of new ones. For example, due to a change of requirements, the client now needs to retrieve the user's email along with details of their posts and the image those posts promote. The original query will no longer serve this requirement and hence will need alteration. Along with this change, there is a need to add a new `Post` class and edit the original `Person` class to mimic the result's updated hierarchy.

```
String query = "{person(id:`105`){email,posts{id,text,image{id,file}}}}".replace("`", "");
public Post(String id, String text, Image image) { /* new Post class constructor */ }
public Person(String id, String email, List<Post> posts) { /* edited constructor */ }
```

- Reusing standard behavior across `Person` and `Image` classes is tricky since they have different structures and require specialized code to handle their fields.

These problems only intensify as the GraphQL schema size and complexity grows or as multiple endpoints get included, requiring developers to formulate more involved queries and deserialization processes. It might not even be feasible in specific scenarios to perform these tasks following the approach illustrated above.

1.1 Problem Statement

Unlike most RESTful Web API implementations, GraphQL takes a client dictated declarative approach to data fetching. This characteristic makes deserialization of response difficult. Every GraphQL response follows precisely the structure defined in the query, as shown in figure 1.3. Fields can appear in any order with many levels of nesting. They may even repeat and take zero or more arguments. Hence, it is challenging to develop a one-size-fits-all class definition that would precisely apply to multiple queries' nested structure. In this work, we approach such impedance-mismatch problem that

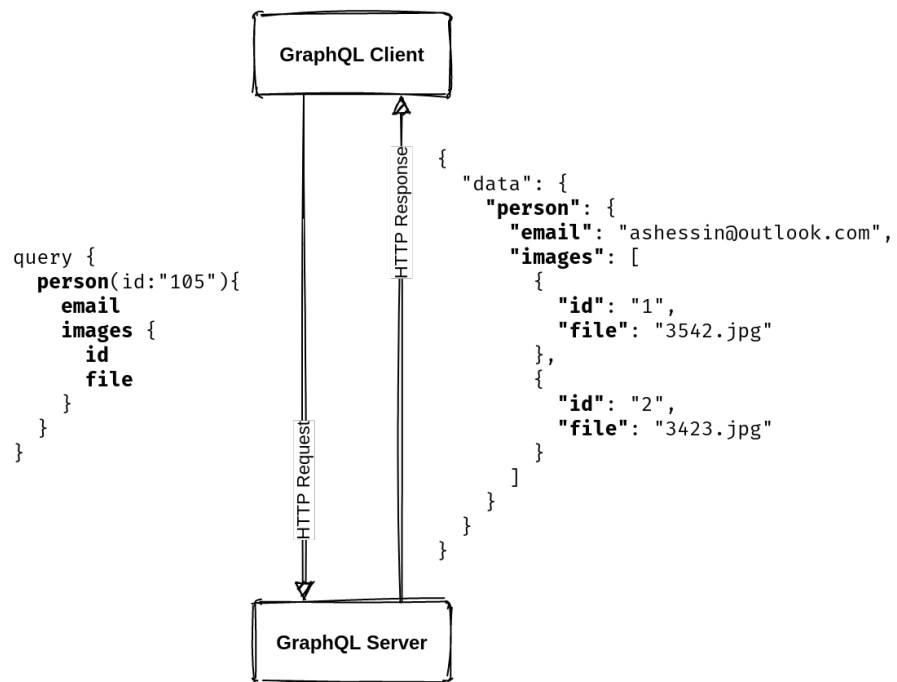


Figure 1.3: GraphQL query request and its response have similar structure

client applications and their developers face while working with language-agnostic GraphQL schema. More specifically, we aim to ease the process of representing hierarchies in a GraphQL query response

to corresponding types in a selected programming language. Four fundamental constraints define the problem space:

On Completeness First, the solution should capture all available details present in each GraphQL response hierarchy and represent them by appropriate types in the underlying programming language of choice. This requirement ensures that the mapping will be complete and fit well for deserialization.

On Correctness Second, aside from any necessary initial setup, the solution should be convenient and safe. A convenient solution requires minimal intervention. Simultaneously, a safe solution guarantees a high level of type-checking robustness in its underlying programming language. In other words, it should not be possible to produce incorrect representations leading to run-time errors.

On Testability Third, the solution should do its job in a reasonable amount of time, even for large response hierarchies, and one should be able to write tests to validate its correctness and completeness. Tests are an essential factor in software quality assurance, so it is wise to have this requirement.

On Generality Lastly, any chosen solution should adequately apply to all valid schemas following the GraphQL specification [6]. A solution that works only for a limited number of GraphQL schemas risks being ineffective if the original schema changes or as entirely new schemas become available.

Implementing a solution that offers a complete, correct, testable, and general approach to deserialize GraphQL query response would significantly improve the developer experience. It would facilitate the agile development of GraphQL client applications that are more robust and reduce runtime failures.

1.2 Contributions

To address the impedance-mismatch issue, we produce a solution that can deserialize GraphQL query response automatically. We accomplish this using experimental compile-time metaprogramming features in the Scala language, thus achieving high compile-time safety guarantees. In doing so, we identify some key contributions of our work:

- We develop a working implementation of our solution as a Scala program that can auto deserialize retrieved GraphQL query response. To the best of our knowledge, there is only one such analogous program but written in the Rust programming language with a set of drawbacks that do not

take form in our implementation. Additionally, a **Scala**-based solution is meaningful given its interoperability with **Java** (a very popular programming language).

- We offer, through examples, a practical approach to **Scala** metaprogramming using macros. **Scala** macros, in all regards, are a niche topic, and there are a limited number of publications that explain its details.
- We exhibit, through comparison, the difference between **GraphQL** and traditional RESTful Web API. Unlike some previous publications, we take a practical approach to this comparison based on GitHub's v3 and v4 API implementations.

2 Solution Overview

This section explains the key ideas or rationale behind our approach to the GraphQL response deserialization problem and gives an overview of our solution. At a very high-level, we can solve the deserialization problem by doing two sequential tasks — finding subgraphs (or extracting details for making type definitions, e.g., Class hierarchies in an Object-oriented language) and generating representations (or code generation, i.e., actually making the required type definitions).

2.1 Task 1 — Finding Subgraphs

As illustrated by figure 1.2 and the accompanying discussion in chapter 1, every GraphQL query essentially retrieves a subgraph from a larger graph, i.e., the business domain graph. The GraphQL schema describes this graph’s overall shape and, by extension, valid queries and response structures. Two defining constraints on the ideal solution are completeness and generality (section 1.1), which requires that the solution be able to deserialize all possible responses from any GraphQL schema. In the context of a graph structure, we can postulate these two constraints as finding all required subgraphs. To satisfy this requirement, we have the option to use one of several sources discussed next.

2.1.1 Source 1: Using GraphQL Schema

In theory, as the schema describes the graph’s overall structure, we can use it to discover all subgraphs. However, in practice, it is not easy to do so for a large graph (or schema) in a reasonable amount of time.

This limitation is a dealbreaker since real-world GraphQL schemas can be complex with many definitions [7], and the time requirements would go against the constraint for testability (section 1.1). Even if we could figure out all subgraphs, most practical client applications use queries that target only a fraction of this set. As a result, most subgraph representations, would go unused, polluting the codebase. Alternatively, an approximate solution using some heuristics, approximation, or depth limiting can not be guaranteed to be complete.

2.1.2 Source 2: Using GraphQL Response

Another approach for obtaining the required subgraphs is to target the GraphQL response on a case-by-case basis. In other words, capture a GraphQL query response, analyze its nested structure, and generate deserialization representations based on this analysis. The requirement can now be satisfied in a reasonable amount of time since we have significantly reduced the number of subgraph structures that we need to obtain.

However, any tangible solution that implements this approach would require run-time information (section 2.3.3), i.e., response, available only after GraphQL query execution. This risks being an unsafe option in case of a malicious server response [8, 9] or when the logic for deserialization of obtained subgraph is unsound. As a result of such unsafety, this source can not satisfy the correctness constraints required (section 1.1). Of course, there are solutions to mitigate this [10, 11], but it adds further to the approach's complexity.

2.1.3 Source 3: Using GraphQL Query

As shown earlier in figure 1.3, a GraphQL response precisely follows the request GraphQL query's overall structure. Additionally, this query is a part of the client application's source code. We can utilize these two pieces of information to obtain any response structure that would need deserialization. This indirect approach offers us the advantage of determining the subgraph before program execution since we no longer depend on the server's response. As a real-world application would consist a limited number of GraphQL queries, we can fulfill the requirement in a reasonable amount of time. Further, this approach guarantees safety against a malicious GraphQL server that may send an invalid response (as per the schema).

2.2 Task 2 — Generating Representations

After we obtain the details for generating type definitions, we need to create those definitions in a manner that can be utilize later for deserialization. More concretely, for example, in an Object-oriented language, we need to generate the class definitions with the appropriate name and constructor parameters. The constructor parameters, in turn, will have their own name and type (could be a primitive type or a reference type).

2.3 Finding Subgraphs and Generating Representations

At this point, we need to consider strategies that would allow us to accomplish both of the above tasks (sections 2.1 and 2.2) taken together. In other words, we need to conceptualize the means to implement them, i.e., finding subgraphs (by targeting GraphQL queries) and generating representations based on them. We will go into details of the concrete implementation in chapter 7. Right now, we are only concerned with the high-level details.

2.3.1 Approach 1: Through IDE Tooling

External tooling like an Integrated Development Environment (IDE) plugin/extension or a standalone program can be developed to perform these tasks. Such a tool would need the ability to parse source code and also manipulate them. Assuming that we are dealing with a target GraphQL project in Java, we could use libraries like Java Poet, JavaParser, or Eclipse JDT.

However, this is not a convenient approach since not everyone uses the same IDE (even for the same language), and the tool would need a separate re-run for each change made by the developer in the target project. Locating GraphQL queries in a large project could also be challenging. This goes against the correctness constraint (section 1.1).

2.3.2 Approach 2: Through Low-level Bytecode Manipulation

Some programming languages like Java and Python have a low-level representation of the source code generated as part of the compilation step. This low-level representation is the *bytecode*, a machine-readable, optimized format for execution. A program based on bytecode analysis and manipulation framework such as ASM (for Java) can be used in this case.

However, this approach would require a separate re-run for each compilation of the target project. Bytecode manipulation is also complicated to debug in case of any issues. There's also no widely accepted, formalized way to test them. These issues go against the testability and correctness constraints (section 1.1).

2.3.3 Approach 3: Through Run-time Metaprogramming

Many programming languages provide some level of support for *intercession* (the program's ability to examine its self-representation) and *introspection* (the program's ability to modify its self-representation) via a *reflection system* during run-time [12]. This characteristic allows for run-time metaprogramming, i.e., a program that treats other programs as data, at run-time. Again taking

Java as an example, we could use the Java Reflection API following this approach.

However, reflection can often lead to run-time errors, which are undesirable and goes against the correctness constraint (section 1.1). Another drawback is that reflection is known to have some performance overhead at run-time.

2.3.4 Approach 4: Through Compile-time Metaprogramming

Some programming languages have syntactic *macro systems* which operate on *Abstract Syntax Trees* (ASTs), i.e., abstract syntactic structure of source code at compile-time. Macros allow for specifying a set of input sequences mapped to their corresponding output sequences in a well-defined procedure [12]. This characteristic enables compile-time metaprogramming, i.e., a program that treats other programs as data, at compile-time. Macros can produce similar effects to that of using reflection. However, since it takes place at compile-time, they are safer and reduce the possibility of run-time errors. They additionally do not pose any performance overhead at run-time. A few programming languages that support syntactic macros include Scala, Rust and Prolog.

2.4 Ensuring GraphQL Query Correctness

The primary objective of this work is to address the deserialization problem when using GraphQL. However, we would ideally also want the overall implementation to ensure GraphQL query correctness, as detailed in chapter 1. Numerous GraphQL client runtimes solve this problem by providing a custom Domain-Specific Language (DSL) based on some input GraphQL schema (e.g., Github’s). GraphQL queries written using such DSL can be statically checked by the compiler (to ensure correctness) or used by the IDE to render hints/warnings. The Caliban-client plugin in the Scala programming language is one such example.

2.5 Implementation Overview

Based on the rationale detailed in this chapter (sections 2.3 and 2.4), we choose to implement a deserialization solution using Scala macros. Additionally, we employ the Caliban-client module to ensure GraphQL query correctness. We believe this way we can best satisfy the constraints defined in the problem statement (section 1.1).

The implementation provides a developer with an annotation class, technically an Annotation Macro — `mapSelection` that applies to immutable value definition for GraphQL queries (written using Caliban-client). This macro annotation undergoes expansion to produce correct representations for mapping

GraphQL response and hence deserialize the fetched data. This expansion happens automatically and is done by the Scala compiler. We only provide instructions on how the expansion should take place.

1. First, we ensure that the annotation application is valid by checking the AST of the definition to which it is applied (must be an AST for a immutable value definition). We define constraints that the AST musts satisfy to reduce the chances of failure during macro expansion at compile-time.
2. Next, we use a recursive algorithm that we devised to extract the structure of the GraphQL query. This algorithm basically parses the right-hand side (i.e., value assignment) of the value definition's AST to perform this task.
3. Then we generate the ASTs representing class definitions which can be used for the deserialization of the received GraphQL response.
4. Lastly, we update the original ASTs so that they are set to map or deserialize to these class definitions.
5. The implementation also provides a method, technically a Def Macro — `illTyped`, that can test macro expansion results. This macro integrates well with existing Behavior-driven testing mechanism in `ScalaTest` (a popular testing tool for Scala and Java).

The above step-by-step procedure is detailed in the implementation chapter (chapter 7), where we use GitHub's GraphQL schema as an example. The figure 2.1 shows an overview.

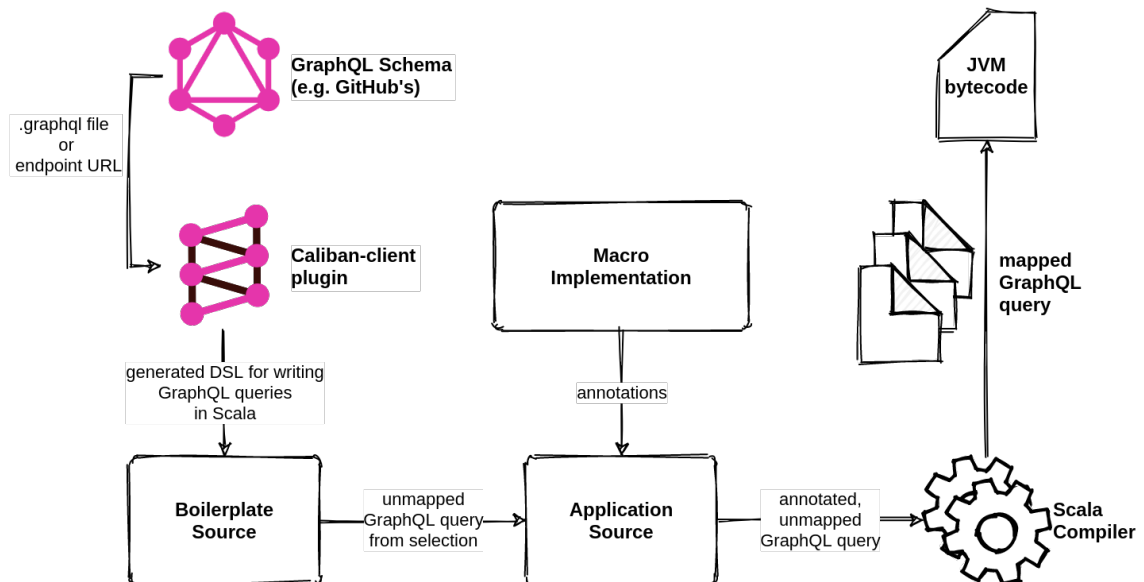


Figure 2.1: System overview

3 GitHub's GraphQL API

This chapter presents GitHub's GraphQL API based on its public schema. We use this schema to develop the implementation and for all examples from this point forward. This chapter explains the basic concepts behind GraphQL and shows how to construct GraphQL queries using GitHub's API as an example. Although the information presented here applies directly to GitHub and its API, the same concepts translate to other GraphQL APIs.

3.1 GraphQL Schema

GitHub's GraphQL schema is available for download at <https://docs.github.com/en/graphql/overview/public-schema> and was first introduced in the year 2016 [13]. Please note that the actual file has over

```
1  """An RFC 3986, RFC 3987, and RFC 6570 (level 4) compliant URI string."""
2  scalar URI
3
4  """The query root of GitHub's GraphQL interface."""
5  type Query {
6    """Lookup a user by login."""
7    user(login: String!): User
8    """Lookup a given repository by the owner and repository name."""
9    repository(name: String! owner: String!): Repository
10 }
11
12 """A user is an individual's account on GitHub that owns repositories and can make new
   ↳ content."""
13 type User {
14   id: ID!
15   avatarUrl(size: Int): URI!
16   status: UserStatus
17   repository(name: String!): Repository
18 }
19
20 """A repository contains the content for a project."""
21 type Repository{
22   id: ID!
23 }
24
25 """The user's description of what they're currently doing."""
26 type UserStatus {
27   id: ID!
28 }
```

Listing 2: Partial GitHub GraphQL schema

38k lines, and hence it is not feasible to discuss it here in its entirety. Therefore, we have reduced this

schema for the sake of simplicity in listing 2. However, the implementation project works with the schema in its entirety.

An online tool like GraphQL Voyager (<https://apis.guru/graphql-voyager/>) can visualize this schema as a graph. The figure 3.1 shows this visualization for easier understanding.

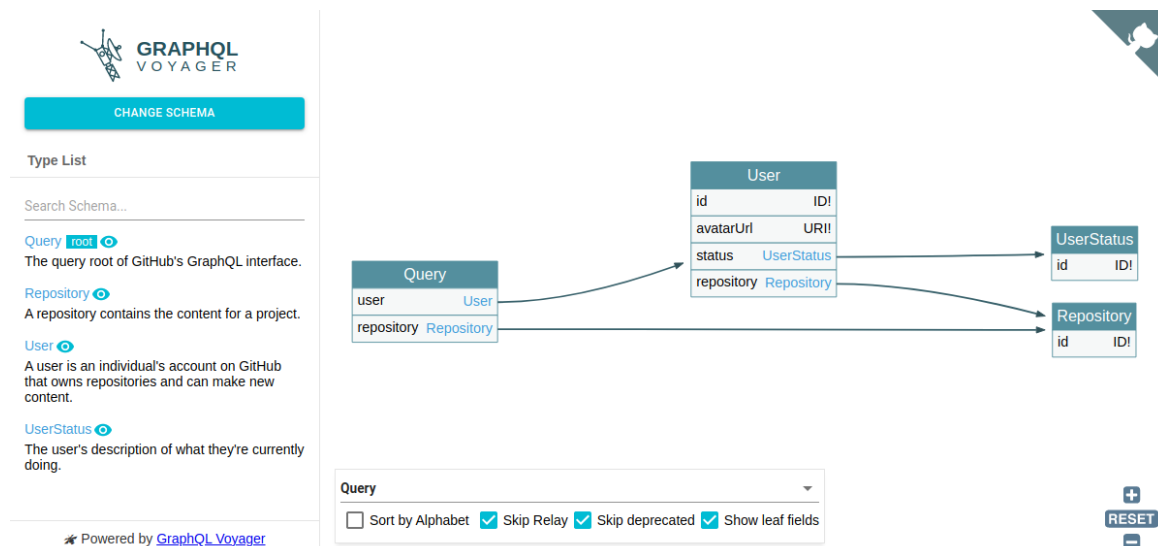


Figure 3.1: Graph model for partial GitHub GraphQL schema

Discussion (listing 2):

- The GraphQL schema describes the graph model for GitHub’s business domain. This business domain is mainly comprised of users (people having a registered account) and their repositories (for saving code).
- GraphQL schemas are written according to the syntactical rules described in the GraphQL Specification [6, 3. Type System] .
- Types are the building block of a GraphQL schema and can be of many kinds (Scalar, Object, Interface, Union, Enum, Input Object) [6, 3.4 Type].
- In listing 2, we have one Scalar type definition — URI and four Object type definitions — Query, User, Repository, UserStatus.
- Scalar types represent primitive *leaf values* in a GraphQL type system. There are also five other primitive scalars — Int, Float, String, Boolean, and ID in addition to custom schema-defined scalars [6, 3.5 Scalars].

- Object types definitions describe the *intermediate levels* of a hierarchy. Consider the `User` object type in figure 3.2. By GraphQL terminology, these have fields (blue) that may take zero or more typed (green) arguments (purple) and return a type (red).
- All field arguments and return types are nullable by default; unless specified as non-nullable by an exclamation mark (!).

<pre> type User { id: ID! avatarUrl(size: Int): URI! status: UserStatus repository(name: String!): Repository } </pre>	<table border="0"> <tr><td>RED</td><td>- Types</td></tr> <tr><td>BLUE</td><td>- Fields</td></tr> <tr><td>PURPLE</td><td>- Arguments</td></tr> <tr><td>GREEN</td><td>- Argument Types</td></tr> <tr><td>!</td><td>- Non-Nullable</td></tr> </table>	RED	- Types	BLUE	- Fields	PURPLE	- Arguments	GREEN	- Argument Types	!	- Non-Nullable
RED	- Types										
BLUE	- Fields										
PURPLE	- Arguments										
GREEN	- Argument Types										
!	- Non-Nullable										

Figure 3.2: GraphQL Object Type anatomy

- The specification mandates each GraphQL schema to have a root node type named `Query`.
- In-turn, the `Query` root node fields identify the top-level entry points for queries that clients can execute. In this case, we can write direct queries for the `User` and `Repository`. However, access to `UserStatus` will need to go through the `User` type.
- Note that each node type has a field named `id`. The GraphQL specification mandates this. All `id` fields should be unique string values on a GraphQL server denoted by the primitive scalar `ID`. This unique field is significant since it allows to establish object equivalence relation post deserialization by overriding the `equals` method of the class definition in `Scala/Java`. This way, two objects with the same `id` field are equivalent even if other fields vary.

3.2 GraphQL Query

GraphQL APIs support three operations — query, mutation and subscription. Our concern is the query operation which can retrieve data from a GraphQL endpoint. Consider the GitHub GraphQL query in listing 3.

```

1 query {
2   user(login:"ashessin") {
3     id
4     avatarUrl(size:42)
5   }
6 }

```

Listing 3: GitHub GraphQL Query

```

1 {
2   "data": {
3     "user": {
4       "id": "MDQ6VXNlcjYxMjk1MTc=",
5       "avatarUrl": "https://avatars.githubusercontent.com/u/6129517?s=42&v=4"
6     }
7   }
8 }

```

Listing 4: GitHub GraphQL Query Response

Discussion (listings 3 and 4)

- GraphQL queries are a selection of fields on a particular GraphQL type. In the listing, there are two selections — one on the Query type and the other on the User type.
- Selections may be nested, as shown in the figure 3.3.

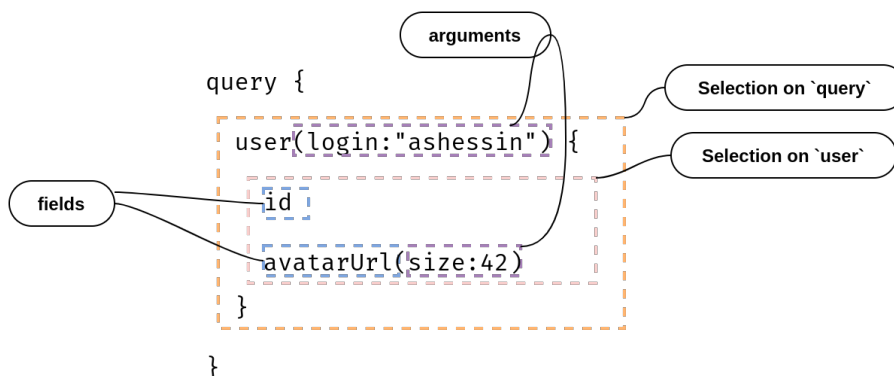


Figure 3.3: GraphQL Query anatomy

- Additionally, the fields may accept zero or more arguments.
- Note that all fields are optional for a given type in the GraphQL schema from the perspective of writing a GraphQL Query. In this case, the type User has many more fields but the query retrieves only its `id` and `avatarUrl`.
- A more complex GraphQL query for the GitHub API is shown in listing 11 of chapter 4.

4 Caliban-client Plugin

Caliban-client is a library that makes it possible to write GraphQL queries using Scala code by providing an easy-to-use embedded Domain-specific language (DSL) [14, 15]. GraphQL queries that are written using this DSL can be type-checked by the Scala compiler for correctness. This chapter illustrates the usage of Caliban-client library and explain how to write type-checked GraphQL queries directly in Scala instead of embedding them as strings.

4.1 Setup

The Caliban-client library offers an sbt plugin command (`calibanGenClient`) for boilerplate code generation (for writing GraphQL queries) given an input GraphQL schema. The sbt dependency manager allows extending a Scala project's build definition through plugins, most commonly by adding new settings. The new settings could be new tasks, as it is in the case of the Caliban-client library. This plugin is activated by following three steps:

1. Define Caliban-client as a Scala project dependency in the project's `build.sbt` file

```
libraryDependencies += "com.github.ghostdogpr" %% "caliban-client" % "0.9.2"
```

2. Declare the Caliban-client plugin in the project/`plugins.sbt` file

```
addSbtPlugin("com.github.ghostdogpr" % "caliban-codegen-sbt" % "0.9.2")
```

3. Lastly, activate the plugin in the project's main `build.sbt` file

```
enablePlugins(CodegenPlugin)
```

4.2 Code generation

The Caliban-client plugin provides a code generation command `calibanGenClient`.

```
calibanGenClient schemaPath outputPath [--scalafmtPath path] [--headers name:value,name2:value2]
```

schemaPath Is the file path or URI to an input GraphQL schema file.

outputPath Is the file path to an output Scala object file which will have the generated boilerplate Scala code.

--scalafmtPath path Is an optional path to a Scalafmt configuration used for code formatting.

--headers name:value,name2:value2 These are additional request header name value pairs to use if the `schemaPath` is an URI.

Command to generate the boilerplate code for the GitHub GraphQL schema (chapter 3), is in listing 5.

```
sbt "calibanGenClient common/src/main/resources/github/schema.docs.graphql
  ↪ common/src/main/scala/com/Github.scala --scalafmtPath .scalafmt.conf"
```

Listing 5: Usage of `calibanGenClient` for GitHub GraphQL input schema

Post `calibanGenClient` command execution, the boilerplate Scala code (listing 6) is available for use in the project's `common/src/main/scala/com/Github.scala` file with package name `com.Github`.

Discussion (listing 6):

RootQuery Represents the root query operation in a GraphQL schema.

SelectionBuilder Is a sealed trait that represents a selection from parent type ``Origin`` that returns a result of type ``A``. Some of its important methods are:

map Maps the result of a selection to a new type ``B``.

mapN Maps a tupled result to a type ``Res`` using a function ``f`` with 2 to 22 parameters.

toGraphQL Transforms a root selection into a GraphQL request.

toRequest Transforms a root selection into an HTTP request ready to be run (using the STTP library).

In addition, there's `~` operator which combines one selection with another selection (both with the same ``Origin``), returning a tuple of both results.

Field Is a case class that represents a single field that returns a result of type ``A`` belonging to the parent type ``Origin``. Two of its parameters are:

name Is the name of field, a `String`.

builder Is the type of field, it could be any of — `Scalar` (represents a GraphQL schema scalar type), `Obj` (represents a GraphQL schema object type), `ListOf` (represents a list of GraphQL schema types), `OptionOf` (represents a nullable GraphQL schema type), `ChoiceOf` (represents a GraphQL schema interface type).

Argument Is a trait that represents an argument in a GraphQL query.

```

1 package com
2
3 import caliban.client._
4
5 object Github {
6
7     type URI = String
8
9     type Query = RootQuery
10    object Query {
11        // Lookup a user by login.
12        def user[A](login: String)(innerSelection: SelectionBuilder[User, A]):
13            ↪ SelectionBuilder[RootQuery, Option[A]] =
14            Field("user", OptionOf(Obj(innerSelection)), arguments = List(Argument("login", login)))
15
16        // Lookup a given repository by the owner and repository name.
17        def repository[A](name: String, owner: String)(innerSelection:
18            ↪ SelectionBuilder[Repository, A]): SelectionBuilder[RootQuery, Option[A]] =
19            Field(
20                "repository",
21                OptionOf(Obj(innerSelection)),
22                arguments = List(Argument("name", name), Argument("owner", owner))
23            )
24    }
25
26    // A user is an individual's account on GitHub that owns repositories and can make new
27    ↪ content.
28    type User
29    object User {
30        def id: SelectionBuilder[User, String] = Field("id", Scalar())
31        def avatarUrl(size: Option[Int] = None): SelectionBuilder[User, URI] =
32            Field("avatarUrl", Scalar(), arguments = List(Argument("size", size)))
33        def status[A](innerSelection: SelectionBuilder[UserStatus, A]): SelectionBuilder[User,
34            ↪ Option[A]] =
35            Field("status", OptionOf(Obj(innerSelection)))
36        def repository[A](
37            name: String
38        )(innerSelection: SelectionBuilder[Repository, A]): SelectionBuilder[User, Option[A]] =
39            Field("repository", OptionOf(Obj(innerSelection)), arguments = List(Argument("name",
40            ↪ name)))
41    }
42
43    // A repository contains the content for a project.
44    type Repository
45    object Repository {
46        def id: SelectionBuilder[Repository, String] = Field("id", Scalar())
47    }
48
49    // The user's description of what they're currently doing.
50    type UserStatus
51    object UserStatus {
52        def id: SelectionBuilder[UserStatus, String] = Field("id", Scalar())
53    }
54 }

```

Listing 6: Caliban-client generated Scala boilerplate for partial GitHub GraphQL schema

```

1  import com.Github._
2  import caliban.client.SelectionBuilder
3  import zio.console.putStrLn
4  import zio.{ App, ExitCode, URIO }
5
6  object Main extends App {
7    private val githubGraphqlEndpoint: String = "https://api.github.com/graphql"
8    private val githubOAuthToken: String      = sys.env("GITHUB_OAUTH_TOKEN")
9
10   override def run(args: List[String]): URIO[zio.ZEnv, ExitCode] = {
11
12     val userFields: SelectionBuilder[User, (String, Option[String])] =
13       User.id ~ User.status(UserStatus.id)
14
15     val selection: SelectionBuilder[RootQuery, (String, Option[String])] =
16       Query.viewer(userFields)
17
18     val request = RequestWrapper(githubGraphqlEndpoint, githubOAuthToken)
19     val result  = request.execute(selection)
20
21     println(selection.toGraphQL())
22
23     result.tap { r =>
24       putStrLn(
25         s"""GraphQL Response Body [{r.getClass.getSimpleName}]:
26           |   $r""".stripMargin
27       )
28     }.exitCode
29   }
30 }

```

Listing 7: GitHub GraphQL query in Scala using Caliban-client

4.3 Type-checked GraphQL Queries in Scala

As evident from listing 6 and listing 2, the code generation process primarily creates a Scala object with accompanying methods for each corresponding GraphQL type and its fields in the input schema file. There are four Scala objects in listing 6 that correspond to the four GraphQL Object types in the original schema (listing 2). Further, these Scala objects have methods that correspond to the fields in input schema. Note that these Scala objects can only be used for writing queries and not deserialization of the response.

Discussion (listing 7):

- The listing shows an example program using the boilerplate code from listing 6.
- Lines 11,12 declare an immutable variable `userFields` with type `SelectionBuilder` on the Origin `User` and result type `Tuple` of `(String, Option[String])`.

```

1 GraphQLRequest(query{viewer{id status{id}}},Map())
2 GraphQL Response Body [Tuple2]:
3   (MDQ6VXNlcjYxMjk1MTc=,Some(8643811))
4
5 Process finished with exit code 0

```

Listing 8: Output for listing 7

- In this case, the `userFields` immutable variable is a combined selection on the users's id and status. The `User.id` method doesn't take any arguments while the `User.Status` method takes another selection, i.e., `UserStatus.id` as it's argument
- The `User.id` and `User.Status` could be combined using the tilde operator since they both have the same Origin i.e. `User`.
- Lines 14,15 declare another immutable value selection with type `SelectionBuilder` on Origin `RootQuery` and result type same as that of `userFields`, i.e., `Tuple of (String, Option[String])`. The `Query.viewer` method retrieves the authenticated user's details as defined by its arguments.
- Line 18 uses the `SelectionBuilder.toGraphQL` method to show the corresponding GraphQL query for the root selection which can be sent to the GraphQL endpoint.
- Lines 21,22,23 print the response of the GraphQL query result.
- Other lines in the listing deal with imports, authentication with the API, etc. and are not relevant to this discussion.
- Output is shown in listing 7.
 - The first line shows the GraphQL query i.e. `query{viewer{id status{id}}}`, formed from the root query selection.
 - Note that the return data is wrapped in a `Tuple`.
- The queries are type-checked. For example:
 - If `User.id` is misspelled as `User.idaho`, there's a compile-time error.


```

/playground/example/src/main/scala/Main.scala:11:12 value idaho is not a member of
↳ object com.Github.User      User.idaho ~ User.status(UserStatus.id)

```
 - If a required parameter to the `User.status` method is missing, there's a compile-time error.


```

/playground/example/src/main/scala/Main.scala:11:28 not enough arguments for method
↳ status: (innerSelection: caliban.client.SelectionBuilder[com.Github.UserStatus,A]):
↳ caliban.client.SelectionBuilder[com.Github.User,Option[A]]. Unspecified value
↳ parameter innerSelection.      User.id ~ User.status()

```

```

1 case class GhUserStatus(id: String)
2 case class GhUser(id: String, status: Option[GhUserStatus])
3
4 val userFields: SelectionBuilder[User, GhUser] =
5   (User.id ~ User.status(UserStatus.id.map(GhUserStatus))).mapN(GhUser)
6
7 val selection: SelectionBuilder[RootQuery, GhUser] =
8   Query.viewer(userFields)

```

Listing 9: Classes and mapping to deserialize GitHub GraphQL query response

```

1 GraphQLRequest(query{viewer{id status{id}}},Map())
2 GraphQL Response Body [GhUser$1]:
3   GhUser(MDQ6VXNlcjYxMjk1MTc=,Some(GhUserStatus(8643811)))
4
5 Process finished with exit code 0

```

Listing 10: Output for listing 9

4.4 Deserializing GraphQL Response to Scala Types

To deserialize the GraphQL response seen in listing 8; two case classes are required that the developer needs to manually create and map back to the selections as shown in listing 9.

Discussion (listing 9):

- Note the two case classes `GhUser` and `GhUserStatus` on lines 1,2.
- These case classes are mapped to the selection using the `map` and `mapN` functions on line 5.
- As a result of these changes, the return type of the selections are now set to `GhUser` instead of a tuple.
- Lines 2,3 of the output in listing 10 shows the string representation of a `GhUser` instance (post deserialization).
- However, if the selection changes i.e. the RHS of `userFields` changes; the `GhUser` and `GhUserStatus` classes will need an update or the developer will need to create additional classes.
- Here, the selection was very small consisting of only two fields. For more complex selections (listings 11 and 12), it's not efficient to manually create the Case Classes.
- The last two bullet points prove that the deserialization issues from chapter 1 are still very much prominent. Although GraphQL queries are type-checked when created through Caliban-client.

```

1  issue(number: 1169) {
2    timelineItems(first: 10) {
3      updatedAt
4      totalCount
5      nodes {
6        __typename
7        ... on AddedToProjectEvent {id}
8        ... on AssignedEvent {id}
9        ... on ClosedEvent {
10         id resourcePath url closer {
11           __typename
12           ... on Commit { id url }
13           ... on PullRequest { id url }
14         }
15       }
16       ... on CommentDeletedEvent {id}
17       ... on ConnectedEvent {id}
18       ... on ConvertedNoteToIssueEvent {id}
19       ... on CrossReferencedEvent {id}
20       ... on DemilestonedEvent {id}
21       ... on DisconnectedEvent {id}
22       ... on IssueComment {id}
23       ... on LabeledEvent {id}
24       ... on LockedEvent {id}
25       ... on MarkedAsDuplicateEvent {id}
26       ... on MentionedEvent {id}
27       ... on MilestonedEvent {id}
28       ... on MovedColumnsInProjectEvent {id}
29       ... on PinnedEvent {id}
30       ... on ReferencedEvent {id}
31       ... on RemovedFromProjectEvent {id}
32       ... on RenamedTitleEvent {id}
33       ... on ReopenedEvent {id}
34       ... on SubscribedEvent {id}
35       ... on TransferredEvent {id}
36       ... on UnassignedEvent {id}
37       ... on UnlabeledEvent {id}
38       ... on UnlockedEvent {id}
39       ... on UnmarkedAsDuplicateEvent {id}
40       ... on UnpinnedEvent {id}
41       ... on UnsubscribedEvent {id}
42       ... on UserBlockedEvent {id}
43     }
44   }
45   id url
46 }
47 }

```

Listing 11: A complex selection of fields on a repository — GraphQL

```

1 // A complex selection... it's not efficient or effective to manually create the Case Classes
2 val repositoryFields: SelectionBuilder[Repository, _] = Repository.issue(1169) {
3   Issue.timelineItems(None, None, Option(10)) {
4     IssueTimelineItemsConnection.updatedAt ~
5     IssueTimelineItemsConnection.totalCount ~
6     IssueTimelineItemsConnection.nodes(
7       AddedToProjectEvent.id,
8       AssignedEvent.id,
9       ClosedEvent.id ~ ClosedEvent.resourcePath ~ ClosedEvent.url ~ ClosedEvent.closer(
10        Commit.id ~ Commit.url,
11        PullRequest.id ~ PullRequest.url
12      ),
13      CommentDeletedEvent.id,
14      ConnectedEvent.id,
15      ConvertedNoteToIssueEvent.id,
16      CrossReferencedEvent.id,
17      DemilestonedEvent.id,
18      DisconnectedEvent.id,
19      IssueComment.id,
20      LabeledEvent.id,
21      LockedEvent.id,
22      MarkedAsDuplicateEvent.id,
23      MentionedEvent.id,
24      MilestonedEvent.id,
25      MovedColumnsInProjectEvent.id,
26      PinnedEvent.id,
27      ReferencedEvent.id,
28      RemovedFromProjectEvent.id,
29      RenamedTitleEvent.id,
30      ReopenedEvent.id,
31      SubscribedEvent.id,
32      TransferredEvent.id,
33      UnassignedEvent.id,
34      UnlabeledEvent.id,
35      UnlockedEvent.id,
36      UnmarkedAsDuplicateEvent.id,
37      UnpinnedEvent.id,
38      UnsubscribedEvent.id,
39      UserBlockedEvent.id
40    )
41   } ~ Issue.id ~ Issue.url
42 }

```

Listing 12: A complex selection of fields on a repository — Scala

5 Scala Macros

Scala is a statically typed language that combines functional and object-oriented programming and mainly runs on the JVM (Java Virtual Machine) platform. It supports both run-time and compile-time metaprogramming, based on the same API — `scala.reflect.api` which was first introduced as an experimental feature in Scala version 2.10. Scala compile-time metaprogramming is realized as the Scala macro system that directly works with Abstract Syntax Trees (ASTs) [16]. This chapter gives background on Scala Macros, focusing on Annotation and Def Macros, two distinct macros types that the Scala programming language supports [17].

5.1 Environment, Universes, and Mirrors

All reflective tasks, whether at compile-time (macros) or run-time require a proper *environment* in Scala which differs based on the flavor of reflection. The distinction between an environment to be used at run time or compile time is encapsulated in a so-called *universe* (`scala.reflect.api.Universe`) which in turn provides an interface to all the principal reflection concepts [16]. This information is made accessible through so-called *mirrors* (`scala.reflect.api.Mirrors`). Mirrors that translate names to symbols (via methods `staticClass/staticModule/staticPackage`) and are referred to as “classloader” mirrors and may be used at compile-time or run-time. There are also so called “invoker” mirrors that allow for reflective invocation (via methods `MethodMirror.apply`, `FieldMirror.get`, etc.) but their usage is limited to run-time.

5.2 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a syntactic representation of the source code for a particular programming language where each AST node corresponds to a source code construct. ASTs are generally easier to work with than plain source code text since they represent the code’s overall structure instead of concrete details (e.g., whitespace) which are inconsequential to the programs that need to manipulate them. In a compiled language like Scala, a dedicated compiler phase transforms source code to ASTs for internal use (listing 13). However, libraries such as `ScalaMeta` can also parse ASTs for multiple programming languages.

1	[ashesh@XPS-13-9300]~% scalac -Xshow-phases			
2	phase name	id	description	
3		--		
4	parser	1	parse source into ASTs, perform simple desugaring	
5	namer	2	resolve names, attach symbols to named trees	
6	packageobjects	3	load package objects	
7	typer	4	the meat and potatoes: type the trees	
8	patmat	5	translate match expressions	
9	superaccessors	6	add super accessors in traits and nested classes	
10	extmethods	7	add extension methods for inline classes	
11	pickler	8	serialize symbol tables	
12	refchecks	9	reference/override checking, translate nested objects	
13	uncurry	10	uncurry, translate function values to anonymous classes	
14	fields	11	synthesize accessors and fields, add bitmaps for lazy vals	
15	tailcalls	12	replace tail calls by jumps	
16	specialize	13	@specialized-driven class and method specialization	
17	explicitouter	14	this refs to outer pointers	
18	erasure	15	erase types, add interfaces for traits	
19	posterasure	16	clean up erased inline classes	
20	lambdalift	17	move nested functions to top level	
21	constructors	18	move field definitions into constructors	
22	flatten	19	eliminate inner classes	
23	mixin	20	mixin composition	
24	cleanup	21	platform-specific cleanups, generate reflective calls	
25	delambdafy	22	remove lambdas	
26	jvm	23	generate JVM bytecode	
27	terminal	24	the last phase during a compilation run	

Listing 13: Scala compiler phases include parsing source into ASTs

```

1 object Main extends App {
2   val greeting: String = "Hello World!"
3   def printGreeting(): Unit = println(greeting)
4
5   printGreeting()
6 }

```

Listing 14: Scala “Hello World” program

Scala ASTs may be *typed* or *untyped*. A typed AST has type information associated with its nodes, while an untyped AST does not contain any such information. Consider the sample “Hello World” program source in listing 14; listing 15 shows its untyped AST, while listing 16 shows the typed AST.

The `scala.reflect.api.Trees` is the base trait for all Scala abstract syntax trees and the common operations available on them. There are three broad categories of `Trees` — `TermTree` (section 5.2.1), `TypTree` (section 5.2.2), `SymTree` (section 5.2.3). Other types of trees are generally short lived constructs such as `CaseDef` that represent individual match cases in a pattern match. Such nodes are neither terms nor types, nor do they carry a symbol [18]. However, regardless of the type of a tree, all trees have a set of common fields.

```

1 ModuleDef(Modifiers(), TermName("Main"), Template(List(Ident(TypeName("App"))), noSelfType,
  ↳ List(DefDef(Modifiers(), termNames.CONSTRUCTOR, List(), List(List(), TypeTree(),
  ↳ Block(List(pendingSuperCall), Literal(Constant(()))), ValDef(Modifiers(),
  ↳ TermName("greeting"), Ident(TypeName("String")), Literal(Constant("Hello World!"))),
  ↳ DefDef(Modifiers(), TermName("printGreeting"), List(), List(List(),
  ↳ Ident(TypeName("Unit")), Apply(Ident(TermName("println")),
  ↳ List(Ident(TermName("greeting")))), Apply(Ident(TermName("printGreeting")), List()))))

```

Listing 15: Scala untyped AST for Hello World program (listing 14)

```

1 ModuleDef(Modifiers(), dummy.Main, Template(List(TypeTree(),
  ↳ TypeTree().setOriginal(Select(Ident(Scala), Scala.App))), noSelfType,
  ↳ List(DefDef(Modifiers(), termNames.CONSTRUCTOR, List(), List(List(), TypeTree(),
  ↳ Block(List(Apply(Select(Super(This(TypeName("Main")), typeNames.EMPTY),
  ↳ termNames.CONSTRUCTOR, List()), Literal(Constant(()))), ValDef(Modifiers(PRIVATE |
  ↳ LOCAL), TermName("greeting "), TypeTree().setOriginal(Select(Select(Ident(Scala),
  ↳ Scala.Predef), TypeName("String"))), Literal(Constant("Hello World!"))),
  ↳ DefDef(Modifiers(METHOD | STABLE | ACCESSOR), TermName("greeting"), List(), List(),
  ↳ TypeTree().setOriginal(Select(Select(Ident(Scala), Scala.Predef), TypeName("String"))),
  ↳ Select(This(TypeName("Main")), TermName("greeting ")), DefDef(Modifiers(),
  ↳ TermName("printGreeting"), List(), List(List(),
  ↳ TypeTree().setOriginal(Select(Ident(Scala), Scala.Unit)),
  ↳ Apply(Select(Select(Ident(Scala), Scala.Predef), TermName("println")),
  ↳ List(Select(This(TypeName("Main")), TermName("greeting")))),
  ↳ Apply(Select(This(TypeName("Main")), TermName("printGreeting")), List()))))

```

Listing 16: Scala typed AST for Hello World program (listing 14)

These common fields include:

isEmpty Denotes if the tree is empty.

isTerm Denotes if the tree represents a term.

isType Denotes if the tree represents a type.

pos Retrieves the position of the tree in the source code. Not all trees have a position. These can be used to display information, warning or errors with reference to the source code.

tpe Retrieves the type associated with the tree. Not all trees have a type and it may be set to null. These are assigned when a tree is typechecked.

symbol Retrieves the symbol associated with the tree. Anything that is defined and named in Scala, such as a class or a method will have an symbol. Symbols are used to establish bindings between a name and the entity it refers to.

children Retrieves all direct children for a Tree. Trees are nested structures.

duplicate Duplicates the tree keeping all details, except for the position information.

Trees are immutable except for three fields — `pos` (Position), `symbol` (Symbol), and `tpe` (Type) [19].

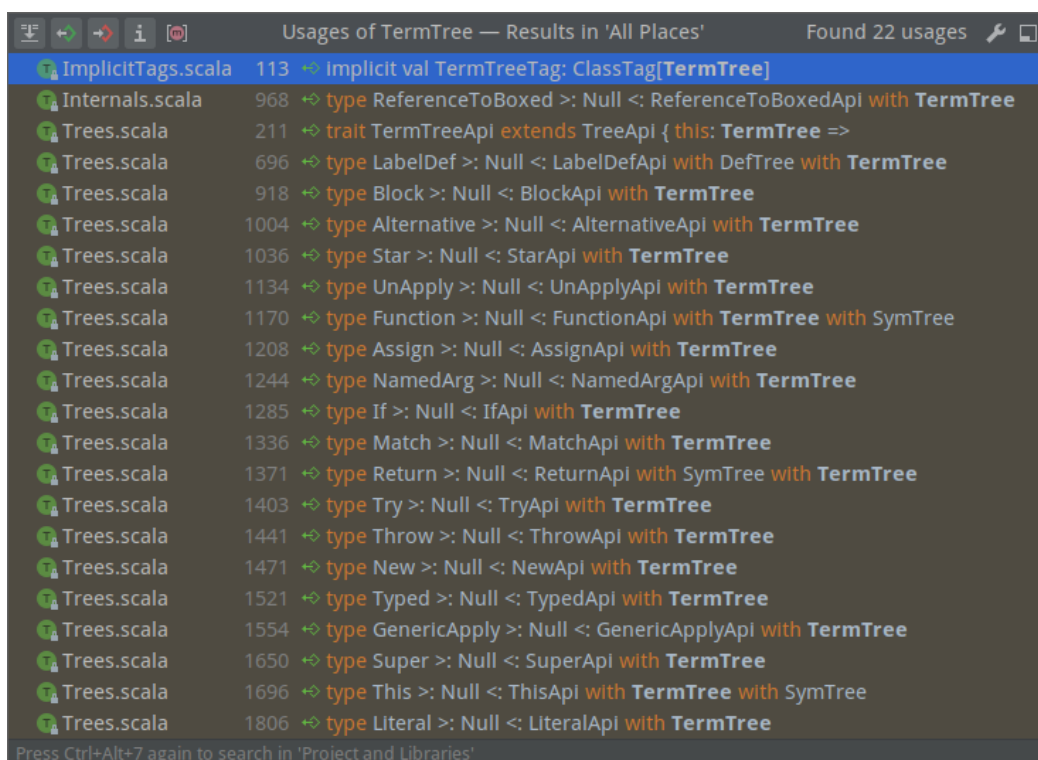


Figure 5.1: All direct types of TermTree

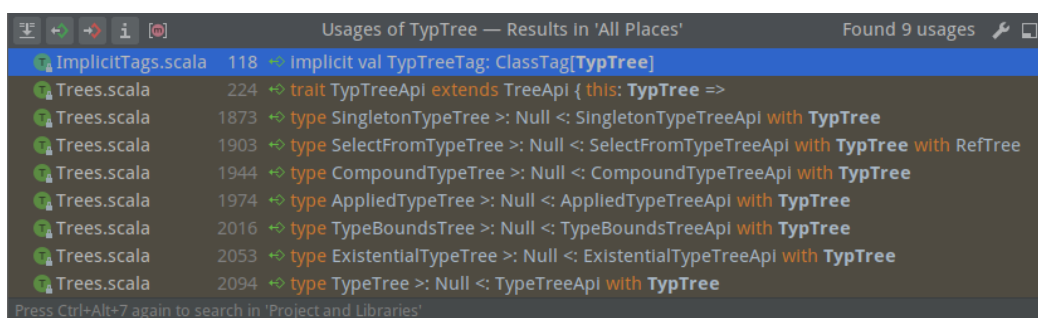


Figure 5.2: All direct types of TypTree

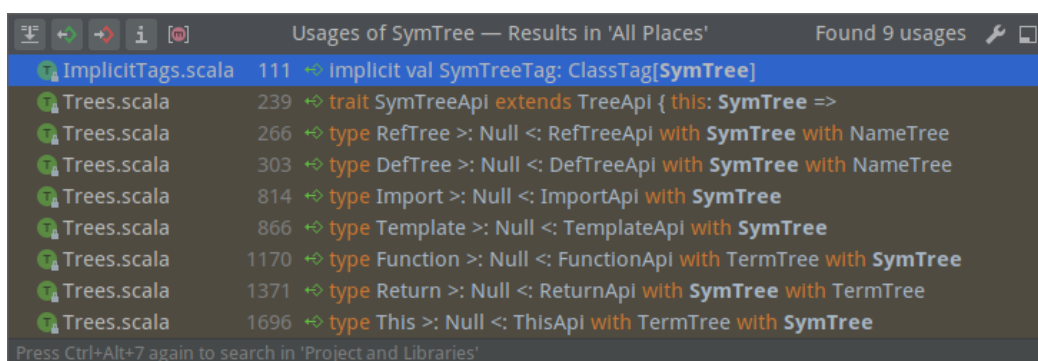


Figure 5.3: All direct types of SymTree

5.2.1 Subclasses of `TermTree`

These trees *represent a term* such as value application — `Apply` tree or a member selection — `Select` tree. Some other types of such trees include: `Block` (a block expression), `UnApply` (extractor object's unapply method), `If` (if-else style conditionals), `Literal`.

- In listings 15 and 16, method call `printGreeting()` is represented as an `Apply` tree.

```
Apply(Ident(TermName("printGreeting")), List())
Apply(Select(This(TypName("Main")), TermName("printGreeting")), List())
```

5.2.2 Subclasses of `TypTree`

These trees *represent a type* and can be used to identify type information explicitly provided in the source code. Some types of such trees are: `TypeTree` (a tree holding an arbitrary type), `CompoundTypeTree` (representing combination of many types with an optional refined member), `TypeBoundsTree` (type definitions with a lower and/or higher bound).

- In listing 16, type information for `App` is represented as a `TypeTree`.

```
TypeTree().setOriginal(Select(Ident(scala), scala.App))
```

- In listing 15, there are no trees of `TypeTree` (or its sub-types) since the AST is untyped.

5.2.3 Subclasses of `SymTree`

A tree that *carries a symbol*. Most likely, these symbols may either be defined ie. `DefTree` (and its subtypes) or referenced ie. `RefTree` (and its subtypes). Further these types of ASTs include: `ClassDef` (representing class definition), `ValDef` (representing both mutable and immutable value definitions), `ModuleDef` (representing object definitions).

- In listings 15 and 16, the method definition for `printGreeting` is represented as a `DefDef` tree.

```
DefDef(Modifiers(), TermName("printGreeting"), List(), List(List()), Ident(TypName("Unit")),
  ↳ Apply(Ident(TermName("println")), List(Ident(TermName("greeting")))))

DefDef(Modifiers(), TermName("printGreeting"), List(), List(List()),
  ↳ TypeTree().setOriginal(Select(Ident(scala), scala.Unit)),
  ↳ Apply(Select(Select(Ident(scala), scala.Pref), TermName("println")),
  ↳ List(Select(This(TypName("Main")), TermName("greeting")))))
```

- In listings 15 and 16, the object definition for `Main` is the root node of the AST and is represented as a `ModuleDef` tree.

```

1  import universe._
2
3  val expr: Expr[_] = reify {
4    object Main extends App {
5      val greeting: String = "Hello World!";
6      def printGreeting(): Unit = println(greeting);
7
8      printGreeting()
9    }
10 }
11
12 println(showRaw(expr.tree))

```

Listing 17: Use of `showRaw` to print Scala AST

```

1  Block(List(ModuleDef(Modifiers(), TermName("Main"), Template(List(Ident(scala.App)),
↪ noSelfType, List(DefDef(Modifiers(), termNames.CONSTRUCTOR, List(), List(List()),
↪ TypeTree(), Block(List(Apply(Select(Super(This(typeNames.EMPTY), typeNames.EMPTY),
↪ termNames.CONSTRUCTOR), List()), Literal(Constant(()))), ValDef(Modifiers(),
↪ TermName("greeting"), Select(Ident(scala.Predef), TypeName("String")),
↪ Literal(Constant("Hello World!"))), DefDef(Modifiers(), TermName("printGreeting"), List(),
↪ List(List(), Ident(scala.Unit), Apply(Select(Ident(scala.Predef), TermName("println")),
↪ List(Select(This(TypeName("Main"), TermName("greeting"))))),
↪ Apply(Select(This(TypeName("Main"), TermName("printGreeting")), List())))),
↪ Literal(Constant(())))

```

Listing 18: Output of `showRaw` to print Scala AST

5.2.4 Inspecting Trees

The reflection universe provides the method `scala.reflect.api.Printers#showRaw` that can display internal structure of a given reflection artifact as a Scala AST.

Discussion (listing 17):

- The universe in the import statement on line 1 may correspond to a run-time or compile-time universe.
- On line 2, `reify` takes the Scala expression, and returns a Scala `Expr` instance. These are mainly used to create typed abstract syntax trees for use in a macro.
- An `Expr` instance simply wraps an abstract syntax tree and tags it with its type.
- The `showRaw` method prints the tree associated with the expression as shown in listing 18.

```

1 val objectBody = expr.tree match {
2   case Block(l1 @ List(*), _) => l1.head match {
3     case ModuleDef(_, _, Template(_, _, body)) => body
4   }
5 }
6
7 println(showRaw(objectBody))

```

Listing 19: Tree traversal via pattern matching

```

1 List(DefDef(Modifiers(), termNames.CONSTRUCTOR, List(), List(List(), TypeTree(),
  ↳ Block(List(Apply(Select(Super(This(typeNames.EMPTY), typeNames.EMPTY),
  ↳ termNames.CONSTRUCTOR), List()), Literal(Constant(()))), ValDef(Modifiers(),
  ↳ TermName("greeting"), Select(Ident(scala.Predef), TypeName("String")),
  ↳ Literal(Constant("Hello World!"))), DefDef(Modifiers(), TermName("printGreeting"), List(),
  ↳ List(List(), Ident(scala.Unit), Apply(Select(Ident(scala.Predef), TermName("println")),
  ↳ List(Select(This(TypeName("Main"), TermName("greeting"))))),
  ↳ Apply(Select(This(TypeName("Main"), TermName("printGreeting")), List()))

```

Listing 20: Output of Tree Traversal via pattern matching

5.2.5 Traversing Trees

Traversing an AST is another common task that is often performed. There are two primary ways to traverse an AST — either via pattern matching or by overriding `scala.reflect.api.Trees.Traverser` methods.

Discussion (listing 19):

- The tree from listing 18 is pattern matched in this listing. Pattern matching is a mechanism for checking a value against a pattern [20, 21].
- This listing essentially uses pattern match to traverse from the root `Block` node to the nested `ModuleDef`'s body.
- On line 2, the tree is checked for a `Block` pattern and uses a binder (`@`).
- On line 3, the bound value obtained in the previous match is again pattern matched with a `ModuleDef` pattern, extracting the body which is returned.
- Ultimately, the AST for the object's body is printed using `showRaw` method, as shown in listing 20.

```

1  val x: List[Int] = List(1, 2) match { // List[Int] is a type, List(1, 2) is an expression
2    case List(a, b) => List(a + b) // List(a, b) is a pattern, List(a + b) is an expression
3  }

```

Listing 21: Multiple Syntactical Context for List [22]

5.2.6 Manipulating Trees — Quasiquotes

Quasiquotes are similar to string interpolators and are a common mechanism to manipulate trees. Manipulating ASTs involve operations such as creating new nodes, deleting them, changing the types or symbols associated with them.

Discussion (listing 21):

- The listing shows varying syntactical contexts for List.
- The construct `List(1, 2)` on line 1, and `List(a + b)` on line 2 are example expressions. Expressions, definitions and imports are covered by `q"..."` interpolator.

```

println(showRaw(q"List(1, 2)"))
// Apply(Ident(TermName("List")), List(Literal(Constant(1)), Literal(Constant(2))))

println(showRaw(q"List(a + b)"))
// Apply(Ident(TermName("List")), List(Apply(Select(Ident(TermName("a")), TermName("$plus")),
//   ↳ List(Ident(TermName("b"))))))

```

- On line 2 `List[Int]` is an example type. Types are covered by `tq"..."` interpolator.

```

println(showRaw(tq"List[Int]"))
// AppliedTypeTree(Ident(TypeName("List")), List(Ident(TypeName("Int"))))

```

- On line 2 `List(a, b)` is an example pattern. Patterns are covered by `pq"..."` interpolator.

```

println(showRaw(tq"List[Int]"))
// Apply(Ident(TermName("List")), List(Bind(TermName("a"), Ident(termNames.WILDCARD)),
//   ↳ Bind(TermName("b"), Ident(termNames.WILDCARD))))

```

- Syntactical similarity between different contexts doesn't imply similarity between underlying trees [22, 23].

```

println(q"List[Int]" equalsStructure tq"List[Int]") //false
println(showRaw(q"List[Int]"))
// TypeApply(Ident(TermName("List")), List(Ident(TypeName("Int"))))
println(showRaw(tq"List[Int]"))
// AppliedTypeTree(Ident(TypeName("List")), List(Ident(TypeName("Int"))))

```

```

println(q"List(a, b)" equalsStructure pq"List(a, b)") //false
println(showRaw(q"List(a, b)"))
// Apply(Ident(TermName("List")), List(Ident(TermName("a")), Ident(TermName("b"))))
println(showRaw(pq"List(a, b)"))
// Apply(Ident(TermName("List")), List(Bind(TermName("a"), Ident(termNames.WILDCARD)),
//   ↳ Bind(TermName("b"), Ident(termNames.WILDCARD))))

```

- Besides the three interpolators listed above, there's `cq"..."` (case clause interpolator) and `fq"..."` (for loop enumerator interpolator).

Splicing

Quasiquotes allow for unquote splicing, which is a way to retrieve (or unquote) a variable number of elements. These could then be modified or used in parts to create new trees. Dots before the unquotee annotate indicate a degree of flattening and are called a splicing rank; `..$` expects the argument to be an `Iterable[Tree]` and `...$` expects an `Iterable[Iterable[Tree]]` [22, 23].

```
val q"f(..$args)" = q"f(a, b)"
// args: List[universe.Tree] = List(a, b)

val q"f(...$argss)" = q"f(a, b)(c)"
// argss: List[List[universe.Tree]] = List(List(a, b), List(c))
```

Lifting

Lifting is an extensible way to unquote custom data types in quasiquotes. Its primary use-case is to support unquoting of literal values and a number of reflection primitives as trees and may be even combined with splicing [22, 23].

```
val two = 1 + 1
// two: Int = 2

val four = q"$two + $two"
// four: universe.Tree = 2.$plus(2)
```

Unlifting

Unlifting is the opposite of lifting and is used to take trees and recover a value from it [22, 23].

```
val q"${left: Int} + ${right: Int}" = q"2 + 2"
// left: Int = 2
// right: Int = 2

left + right
// res0: Int = 4
```

5.3 Writing Macros

As mentioned in the opening paragraph, our focus is on two particular kinds of macros that the Scala programming language provides — Def Macros and Annotation Macros. This section builds on the topics introduced in the previous sections to show how to write them.

An essential aspect of Scala macros is a separate compilation unit. To perform macro expansion, the Scala compiler needs a macro implementation in its executable, compiled form. Thus developers must compile macro implementations before they can use it [24]. Otherwise, the compilation would fail as shown in listing 22. This need for a separate compilation unit is generally not an issue when the

```

1 /playground/example/src/main/scala/dummy/Main.scala:6:13
2 macro implementation not found: apply
3 (the most common reason for that is that you cannot use macro implementations in the same
   ↪ compilation run that defines them)
4   helloWorld("Hi there!");

```

Listing 22: Macros usage error in a poorly configured project

```

1 scalaVersion in ThisBuild := "2.13.4"
2
3 // PROJECTS
4 lazy val global = project
5   .in(file("."))
6   .settings(settings)
7   .disablePlugins(AssemblyPlugin)
8   .enablePlugins(CodegenPlugin)
9   .aggregate(common, example)
10
11 lazy val common = project
12   .in(file("common")) // macro implementation
13   .settings(
14     name := "playground-common",
15     settings,
16     assemblySettings,
17     libraryDependencies ++= commonDependencies ++ Seq(dependencies.scalaReflect)
18   )
19
20 lazy val example = project
21   .in(file("example")) // macro usage
22   .settings(
23     name := "playground-example",
24     settings,
25     assemblySettings,
26     libraryDependencies ++= commonDependencies
27   )
28   .dependsOn(common)

```

Listing 23: Multi-project builds using sbt for Scala macros

macro is in a separate packaged dependency (.jar file). However, when working in a project, that implements and also uses the macro, two separate modules (or sub-projects) must be present — one with the macro implementation and other with the macro usage. Additionally, builds must be setup in a way that the second module (macro using) depends on the former (macro implementing). This can be easily configured via a dependency manager like sbt as in listing 23. With this setup in place, the next two subsections go over the implementation of a Def Macro (log) and an Annotation Macro (benchmark).

5.3.1 Def Macro — `log`

The listing 24 shows a program using a Def Macro — `log`. Def Macros look similar to method calls. In this case, the macro prints out a message to the console along with the details of the source code location where it was called.

```

1 package dummy
2
3 import macros.dummy.log
4
5 object Main extends App {
6   val greeting: String = "Hello World!"
7   def printGreeting(): Unit = println(greeting)
8
9   log("Log this!!!")
10  printGreeting()
11 }

```

Listing 24: Usage for `log` Def Macro

```

1 package macros.dummy
2
3 import scala.language.experimental.macros
4 import scala.reflect.macros.blackbox
5
6 /** A Def Macro which logs the message and its position in source code. */
7 object log {
8   def apply(msg: String): Unit = macro logMacro.impl
9 }
10
11 class logMacro(val c: blackbox.Context) {
12   import c.universe._
13
14   def impl(msg: c.Tree): c.Tree = {
15     c.info(c.macroApplication.pos, "expanding `log` def macro", force = false)
16
17     q"""
18       println(${msg.pos.source.path})
19       println(${msg.pos.line})
20       println($msg)
21     """
22   }
23 }

```

Listing 25: Implementation for `log` Def Macro

Discussion (listing 25):

- On line 3, the import statement signals the Scala compiler to enable macros. Alternatively, the compiler switch `-language:experimental.macros` can also be used.

- The import of `scala.reflect.macros.blackbox` on line 4 is required to use the macro `Context` on line 11 that wraps the universe during compile-time. Macros that faithfully follow their type signatures (eg. return type) are called blackbox macros as expressed by the import.
- The `apply` method on line 8 accepts a single parameter of type `String` and has no return type (`Unit`). It's right hand side uses the keyword macro and references the implementation for the macro — `logMacro.impl`. Since this method is automatically invoked, no parameters are passed to it in the source code.
- Note that `Def Macros` method definitions do not support `Named` or `Default` arguments.
- The `logMacro` class on line 11 is automatically instantiated with a `Context` by the compiler.
- The `logMacro.impl` method on line 12 receives the AST from the macro call site on line 9 of listing 24. In this case this AST is a `Literal` for “Log this!!!”.
- As the macro is expanded, `c.info` method on line 15 will print a message to the user as shown on lines 5,6 of listing 26.
- The lines 17–21 use quasiquotes to construct ASTs for three print line statements. The first one for printing the source file path. The second one for printing the macro invocation line number in the source file. The last one prints the original message to the macro invocation at the call site, in this case — “Log this!!!”.
- The run-time output is shown in listing 27.

```

1 sbt:playground> compile
2 [info] compiling 1 Scala source to /playground/common/target/scala-2.13/classes ...
3 [info] done compiling
4 [info] compiling 1 Scala source to /playground/example/target/scala-2.13/classes ...
5 [info] /playground/example/src/main/scala/dummy/Main.scala:9:6: expanding `log` def macro
6 [info]   log("Log this!!!")
7 [info]       ^
8 [success] Total time: 4 s, completed Feb 25, 2021, 3:53:49 PM

```

Listing 26: Compiler messages during `log` Def Macro expansion

```

1 /playground/example/src/main/scala/dummy/Main.scala
2 9
3 Log this!!!
4 Hello World!
5
6 Process finished with exit code 0

```

Listing 27: Output for listing 24 with Def Macro usage

5.3.2 Annotation Macro — **benchmark**

One limitation with Def macros is that they can not introduce top level definitions at their call site. Annotation macros in `Scala` can do this and hence are even more powerful. Macro annotations enable textual abstraction on definitions. Annotating any top-level or nested definition with something that `Scala` recognizes as a macro will let it expand, possibly into multiple members [25]. The listing 28 shows an example program using an Annotation Macro — `benchmark`. Annotation Macro look similar to annotations (i.e., prefixed with `@`). In this case, the macro modifies the definition of a method that it is annotated with to print the total execution time on each call.

```

1 package dummy
2
3 import macros.dummy.benchmark
4
5 object Main extends App {
6   val greeting: String = "Hello World!"
7
8   @benchmark
9   def printGreeting(): Unit = println(greeting)
10
11   printGreeting()
12 }

```

Listing 28: Usage for ‘benchmark’ Annotation Macro

Discussion (listing 31)

- The import of `scala.reflect.macros.whitebox` on line 5 is required to use the macro `Context` on line 13 that wraps the universe during compile-time. Unlike Def Macros, Annotation Macros are always whitebox since they can’t have precise signatures in `Scala`’s type system.
- All classes meant to be used as an annotation in `Scala`, extend the `scala.annotation.StaticAnnotation` class, shown on line 8.
- The name `macroTransform` and the signature `annottees: Any*` of that macro on line 10 are important as they tell the macro engine that the enclosing annotation is a macro annotation.
- Unlike Def Macros, Annotation Macros can indirectly use Named and Default arguments in their definition. These are however not presented here to keep the example simple.
- The `benchmarkMacro` class on line 13 is automatically instantiated with a `Context` by the compiler.

```

1 sbt:playground> compile
2 [info] compiling 1 Scala source to /playground/common/target/scala-2.13/classes ...
3 [info] done compiling
4 [info] compiling 1 Scala source to /playground/example/target/scala-2.13/classes ...
5 [info] /playground/example/src/main/scala/dummy/Main.scala:8:4: expanding `benchmark`
   ↳ annotation macro
6 [info]   @benchmark
7 [info]     ^
8 [info] done compiling
9 [success] Total time: 4 s, completed Feb 25, 2021, 4:05:10 PM

```

Listing 29: Compiler messages during **benchmark** Annotation Macro expansion

```

1 Hello World!
2 Method `printGreeting` took: 47ms
3
4 Process finished with exit code 0

```

Listing 30: Output for listing 28 with Annotation Macro usage

- The `logMacro.impl` method on line 16 receives the AST from the macro call site on line 8 of listing 28. In this case this AST is a method definition (tree type `DefDef`) for the `printGreeting` method. It's possible to receive multiple ASTs for a single annotation, depending on the annotated definition. For example, if a class definition is annotated and it has a companion (object), then both are passed into the macro [25].
- ASTs passed to macro annotations by default are untyped. They must be type-checked via call to the `c.typecheck` method if type information is required (not shown here).
- As the macro is expanded, `c.info` method on line 17 will print a informational message as shown on lines 5,6 of listing 26.
- The lines 19–31 use pattern matching on the AST.
- The annotated definition must be for a method, else the compilation is aborted through `c.abort` on line 30.
- The quasiquote on line 20 is for a method definition and uses lifting.
- The lines 21–28 use quasiquotes to reconstruct the method definition with modifications.
- This modified AST is stored in an immutable variable and returned on line 29 by wrapping it inside an `Expr` type.
- The run-time output is shown in listing 30.

```

1 package macros.dummy
2
3 import scala.annotation.{ compileTimeOnly, StaticAnnotation }
4 import scala.language.experimental.macros
5 import scala.reflect.macros.whitebox
6
7 @compileTimeOnly("Enable macro paradise to expand macro annotations.")
8 class benchmark extends StaticAnnotation {
9   //noinspection ScalaUnusedSymbol
10  def macroTransform(annottees: Any*): Any = macro benchmarkMacro.impl
11 }
12
13 class benchmarkMacro(val c: whitebox.Context) {
14   import c.universe._
15
16   def impl(annottees: c.Expr[Any]*): c.Expr[DefDef] = {
17     c.info(c.macroApplication.pos, "expanding `benchmark` annotation macro", force = false)
18
19     annottees.map(_._tree).toList match {
20       case q"$mods def $tname[.. $tparams](... $paramss): $tpt = $expr" :: Nil =>
21         val ast =
22           q"""$mods def $tname[.. $tparams](... $paramss): $tpt = {
23             val start = System.nanoTime()
24             val result = $expr
25             val end = System.nanoTime()
26             println("Method `" + ${tname.toString} + "` took: " + (end - start)/1000000 +
27               ↪ "ms")
28             result
29           }"""
30       case _ => c.abort(c.enclosingPosition, "Annotation @Benchmark can be used only with
31         ↪ method definition.")
32     }
33   }
34 }

```

Listing 31: Implementation for **benchmark** Annotation Macro [26]

6 APIs

This chapter gives an overview of APIs with a focus on Web APIs and related technologies. The last section of this chapter compares GitHub’s GraphQL API with its REST API to illustrate the main differences between them from a client’s perspective.

6.1 What is an API?

Application Programming Interface (API) as per NIST definition [27] is —

A system access point or library function that has a well-defined syntax and is accessible from application programs or user code to provide well-defined functionality.

Although the above definition is straightforward, an API can mean dissimilar things under varying contexts and to different audiences. This is why it’s important that we further discuss the question “What is an API?”. We do this below by using a modified example [28] from the book *The Design of Web APIs* by author Arnaud Lauret. Consider the task of capturing, editing, and uploading a photo to a social media website using a third party camera app:

1. First, to take a photo, the app would have to interact with the camera hardware on the smartphone. Camera features vary a lot across smartphones. Regardless of such variations, we are able to take a photo since functions like focus, capture, zoom etc. are provided by the camera driver (a low level system program). This driver is usually embedded in the Operating System (OS) and the functionality is available to the application via the *Operating System’s Camera API*.
2. To edit (eg. crop and rotate) this photo, the app could use an image manipulation library, say ImageMagick. This is generally accomplished by using a language specific implementation like MagickWand for C or Magick++ for C++. Regardless of the concrete choice of *Library Language API* (sometimes called *binding*), the end result would be the same since they both internally utilize ImageMagick library to complete the task at hand.
3. Finally, to upload the edited photo, the camera app will use another API; this time a *Remote API* provided by the social media site which is available over a network connection. Most likely,

this network is the internet. Further, if the remote API is designed to work with web standards, it's fair to assume that it's a *Web API*.

In the above example, we saw the involvement of different APIs at each step of the process. Note that this is by no means an exhaustive list of API types. However, an API, whatever its type, is first and foremost an interface (or contract) in computing that facilitates interaction between two separate software components. More strictly, it's an abstraction of the underlying implementation ie. the underlying code — which executes when the API is used. So, to summarize, an API is a computing interface that acts as an abstraction to facilitate interaction between two separate software components.

6.2 Why do we need APIs?

Simply put, good abstractions in a software system helps reduce its complexity and allows developers to write easily extendable programs. Let's go back to the example of clicking a photo from the previous section. Only this time, we are not aware about the concept of abstraction via APIs. This change has some serious implications on the entire flow:

1. To click a photo, the third party camera app would have to now know how to interact with and control several hundreds if not thousands of camera hardware modules through their respective driver. That too with varying sets of specification like zoom range, megapixels, ISO range, etc. This will require writing specialized code for each camera, within the app itself.
2. For editing the captured photo, the app may no longer be able to use `ImageMagick` if it is not written in the app's native programming language. This problem is further compounded when the same app is developed and maintained in different languages for different OSs. The developers may resort to using more than just one image manipulation library in each case which can lead to non-uniform results.
3. Finally, to put this edited photo on the social media website, we might have to use a separate app. Possibly the one provided by the Social Media site which shares access to the storage space also used by the camera app. This is not only inconvenient, but can also be a major security/privacy risk.

We can safely say that the complexity of the whole process and the camera app has drastically increased. It now has to do a lot more in comparison to accomplish the same task as before.

6.3 Types of APIs

There are several ways in which we could categorize APIs.

For example, *based on the release policy*, APIs can be labeled as private, partner and public:

- *Private*: For internal use by a team, company, etc.
- *Partner*: Shared with one or more partners.
- *Public*: Available to everyone for use. The Social media API that we discussed earlier in Section 2.1 is a good example of this. Other API examples, ie. The OS's Camera API and the Library's Language API are also likely public.

Another common way of categorizing APIs is *based on location*:

- *Local*, if the underlying API code is on the same system at which it is used. They share the same address space. In the example from section 6.1, the OS's Camera API and the Library's Language APIs are both on the smartphone; as is the third party camera app using them.
- *Remote* APIs on the other hand are designed to be used over a disjoint address space (usually over a network). A subcategory of Remote APIs are Web APIs. A Web API is a Remote API that is designed based on web standards. The social media API for uploading the photo is an example of Web API and by extension, is also a Remote API.

6.4 Web API Data Exchange Format — JSON

A serialization format that is used for communication by an API is called *data exchange* or *data-interchange* format. Serialization is the process of translating a data structure for use in communications protocols (like HTTP), data storage (as file or in a database), etc. There are a number of binary and text based serialization formats. For example:

- binary based serialization: Protocol Buffers, Avro, BSON, MessagePack
- text based serialization: JSON, YAML, XML, CSV

Binary based serialization formats are generally more compact but are not human readable. Text based serialization formats on the other hand are human readable and hence greatly eases debugging. Additionally, many application layer web protocols such as HTTP (excluding newer HTTP/2), FTP, SMTP, etc. are text oriented. As a result, text-based data-interchange is commonly used with them (specially true for public Web APIs).

JSON (JavaScript Object Notation) for data-interchange has gained widespread adoption in recent years. JSON's popularity can be attributed to a number of factors, however the most important one is that it's a subset of JavaScript which is mainly used for programming web pages in a web browser. JSON is also lightweight and easier to read (by humans) and parse (by machines) as compared to traditional XML (Extensible Markup Language).

JSON values can represent four primitive types (strings, numbers, booleans, null) and two structured types (objects and arrays). These are detailed below:

- string: A sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. Examples: "Ashe\"sh", "A", "a", ""
- number: A base 10 whole or fractional number with no extra leading zeros. When prefixed by a hyphen, the number is negative. Scientific notation can also be used. Examples: 42, 0.5, -0.5, 2.99792458e8
- boolean: A lowercase true or false literal.
- null: A null literal.
- object: An unordered set of name/value pairs. An object begins with a left brace and ends with a right brace. Each name (also called field, key) is followed by colon and the name/value pairs are separated by comma. All names within an object should be unique. Some examples:
 - { "id": 105, "email": "ashessin@outlook.com" }
 - { "address": { "city": "Chicago", "zip": 60607 } }
 - {}
- array: An ordered collection of values. An array begins with a left bracket and ends with a right bracket. Values are separated by comma. Some examples:
 - ["abc", 25, true, null, {}, []]
 - ["abc", 25, { "city": "Chicago", "zip": 60607 }]
 - [1, 2, 3]
 - []

A *JSON text* (or less formally *JSON document/instance*) is an object or array and can be used to construct a hierarchy through nesting (see examples above). The terms "object" and "array" come from the conventions of JavaScript but virtually all modern programming languages support such notions. Hence parsing and deserializing JSON is more straightforward than XML or CSV.

For example, the JSON text:

```
{ "id": 105, "email": "ashessin@outlook.com" }
```

Can be deserialized in Java as an instance of a `Person` class; the “ashesh” `Person` object below:

```
Person ashesh = new Person(105, "ashessin@outlook.com");
```

Where the `Person` class is:

```
public class Person {
    private Integer id;
    private String email;

    public Person(Integer id, String email) {
        this.id = id;
        this.email = email;
    }
}
```

Listing 32: Example `Person` class

There are a number of JSON serialization/deserialization and parsing libraries. Google’s `Gson` and FasterXML’s `Jackson` are prominent examples. Other libraries like `jsonschema2pojo` can even automatically generate the corresponding Java/Scala class(es) for an input JSON or JSON Schema. For example the `Person` class above can be auto generated (figs. 6.1 and 6.2).

jsonschema2pojo

Generate Plain Old Java Objects from JSON or JSON-Schema.

```
1 { "id": 105, "email": "ashessin@outlook.com" }
```

Star

5,372

Tweet

Package

com.example

Class name

Person

Target language:

☒ Java
 ☐ Scala

Source type:

☐ JSON Schema
 ☒ JSON
 ☐ YAML Schema
 ☐ YAML

Annotation style:

☐ Jackson 2.x
 ☐ Jackson 1.x
 ☐ Gson
 ☐ Moshi
 ☒ None

☐ Generate builder methods

☐ Use primitive types

☐ Use long integers

☐ Use double numbers

☐ Use Joda dates

☐ Use Commons-Lang3

☐ Include getters and setters

☒ Include constructors

☐ Include `hashCode` and `equals`

☐ Include `toString`

☐ Include JSR-303 annotations

☐ Allow additional properties

☐ Make classes serializable

☐ Make classes parcellable

☐ Initialize collections

Property word delimiters:

- _

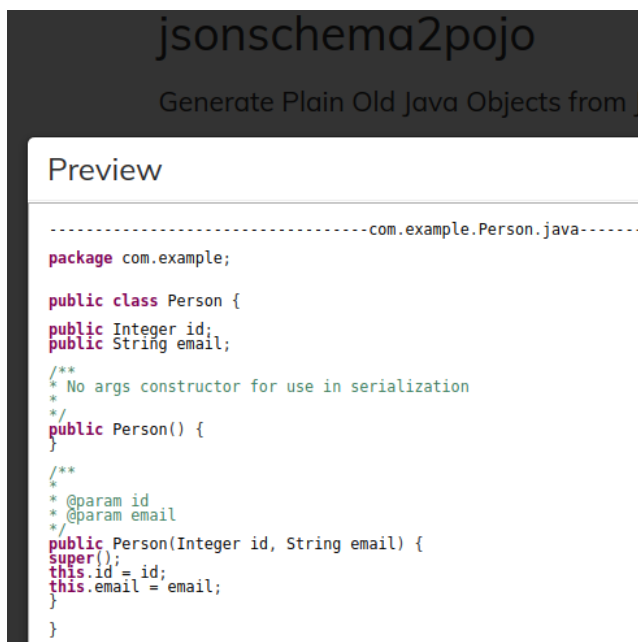
Preview

Zip

Use this tool offline:

[Maven plugin](#)
[Gradle plugin](#)
[Ant task](#)
[CLI](#)
[Jc](#)

Figure 6.1: `jsonschema2pojo` input



The screenshot shows the output of the jsonschema2pojo tool. At the top, the title 'jsonschema2pojo' is displayed in a large, bold font, followed by the subtitle 'Generate Plain Old Java Objects from JSON Schema'. Below this, a 'Preview' section shows a code snippet for a Java class named 'Person'. The code is enclosed in a dark-themed box with a light border. The code includes package declarations, class declarations, and method definitions with Javadoc comments.

```

-----com.example.Person.java-----
package com.example;

public class Person {
    public Integer id;
    public String email;

    /**
     * No args constructor for use in serialization
     */
    public Person() {
    }

    /**
     * @param id
     * @param email
     */
    public Person(Integer id, String email) {
        super();
        this.id = id;
        this.email = email;
    }
}

```

Figure 6.2: jsonschema2pojo output

6.5 Web API Common Terms and Definitions

In this section, we go over some common API terms in context of the web. We provide relevant examples wherever possible, however, note that such examples may often transcend their use case listed here and hence, others may label them differently.

6.5.1 API Description

An *API description* sometimes also called an *API definition* is a language-agnostic file (or set of files) that describes or defines the working of a specific API. Definitions often comply with a standard *API specification* like GraphQL, OpenAPI, RAML, API Blueprint or RADL. These specifications can be considered as *IDLs* (Interface Description Languages) designed to provide a structured description of an API and are generally meant for humans and/or machine consumption. The descriptions may include information like:

- Available API endpoints and operations
- Operation expected parameters input and output
- Authentication methods
- Licensing, versioning, etc.

These structured descriptions are precise enough to generate client and server libraries (in many programming languages), documentation, tests, and other software artifacts. For example, GitHub APIs are described in an OpenAPI 3.0 compliant document as well as GraphQL schema (chapter 3).

6.5.2 API Documentation

An *API documentation* is a technical content deliverable, containing instructions about how to effectively use an API [29]. These can be auto generated from API descriptions using a compatible program and are meant specifically for human consumption. Auto generated documentation can further be edited manually to include specific usages. Examples include GitHub’s v3 and v4 API documentation and Cisco WebEx REST API documentation.

6.5.3 Data Description Specification

A large portion of an API specification may be dedicated to the description of data for validation and/or hypermedia. The specification may provide its own requirements or use an existing specification (as a subset). We will focus on JSON Schema which is a JSON-based format for describing the structure of JSON data that can be used for validation and hypermedia.

For Validation

In the example listing 33, both JSON documents are syntactically correct ie. well-formed and represent the same real-world “Person”. However, the document on right is likely less formal since the value for the email field is not really an email address.

<pre>{ "id": 105, "email": "ashessin@outlook.com", "createdAt": "2013-12-07T12:23:17" }</pre>	<pre>{ "id": 105, "email": "some-random-text", "createdAt": 1386418997 }</pre>
---	--

Listing 33: Example JSON representation of a Person

Discussion(listing 33):

- In both cases, value for the email key is a string, however JSON provides no way to signal that the string must be an email address.
- Similar argument can be made for the createdAt key whose string value is formatted as an RFC 3339 date-time on the left. Whereas, on the right side, the same date-time is represented as a numeric value in the UNIX Epoch format.

- Not only that, it's entirely possible for any of the three fields (`id`, `email`, `createdAt`) to be optional.
- It's difficult to determine the right or intended representation of the data.

Such issues are resolved by using JSON Schema's Vocabulary for Structural Validation. For example, consider the accompanying JSON Schema document shown in listing 34.

```

1  {
2    "$schema": "http://json-schema.org/draft-07/schema",
3    "$id": "http://example.com/person.json",
4    "title": "Person",
5    "description": "Schema for JSON representing a Person.",
6    "type": "object",
7    "properties": {
8      "id": {
9        "type": "integer"
10     },
11     "email": {
12       "type": "string",
13       "format": "email"
14     },
15     "createdAt": {
16       "type": "string",
17       "format": "date-time"
18     }
19   },
20   "required": [
21     "id",
22     "createdAt"
23   ]
24 }
```

Listing 34: Example JSON Schema (for validation)

Discussion(listing 34):

- The `format` key on lines 13,17 of listing 34, allows for basic semantic validation on certain kinds of string values that are commonly used.
- More specifically, here, the `email` key must have a string value, formatted as an email address ([RFC 5322](#)).
- While the `createdAt` key must have a string value, formatted as datetime ([RFC 3339](#)).
- Alternatively, JSON Schema also allows regular expression patterns for validation of strings. This gives additional flexibility to the end user.
- We can now validate the two example JSON instances against this schema by using a program such as `ajv`.

Although, here we have taken a tiny example specific to JSON, other formats can also have similar solutions. For example the XML Schema languages — DTD, RelaxNG for validating XML documents and YAML Schema for validating YAML documents.

For Hypermedia

The Web is made up of diverse types of interlinked, nonlinearly accessed media types; collectively termed *hypermedia*. A JSON instance can be part of such hypermedia that is related to other JSON instances and media. However, JSON has no support for hyperlinks (unlike XML and HTML). Hyperlinks or links as they are often called, are important since they allow for easier navigation and manipulation of media in a hypermedia environment like HTTP. For example, three “Images” may be individually represented by the JSON documents in listing 35.

<pre>{ "id": 1, "personId": 105 "file": "3542.jpg", }</pre>	<pre>{ "id": 2, "personId": 105 "file": "3423.jpg", }</pre>	<pre>{ "id": 3, "personId": 106 "file": "2323.jpg", }</pre>
---	---	---

Listing 35: Example JSON representations for Images

Discussion(listing 35):

- These representations alone tell us nothing about how to retrieve them or any related representation.
- For instance we don’t know how to view the image’s owner ie. the person representation, despite knowing that the first two images are owned by `personId 105` while the last image is owned by `personId 106`.

This issue can be addressed by using JSON Schema’s Vocabulary for Hypermedia Annotation as show in listing 36. We can include such information in the JSON Schema under the links array in a standard way.

Discussion (listing 36):

- On lines 26,30 of listing 36, we have URI templates which points to self (ie. JSON representation of the image) and the image’s owner (ie. JSON representation of person) respectively.
- We can deduce the links by substituting concrete values for `id` and `personId` from the three JSON documents into these URI templates.

- By doing so, the first two image representations are from "http://example.com/images/1" and "http://example.com/images/2" while their owner is at "http://example.com/persons/105".
- Similarly, the third or the last image is from "http://example.com/images/3" and their owner is at "http://example.com/persons/105".

```

1  {
2    "$schema": "http://json-schema.org/draft-07/schema",
3    "$id": "http://example.com/image.json",
4    "title": "Image",
5    "description": "Schema for JSON representing an Image owned by a Person.",
6    "type": "object",
7    "properties": {
8      "id": {
9        "type": "integer"
10     },
11     "personId": {
12       "type": "integer"
13     },
14     "file": {
15       "type": "string",
16       "pattern": "/\\w*\\.jpg$/g"
17     }
18   },
19   "required": [
20     "id",
21     "personId",
22     "file"
23   ],
24   "links": [{
25     "rel": "self",
26     "href": "http://example.com/images/{id}"
27   },
28   {
29     "rel": "owner",
30     "href": "http://example.com/persons/{personId}"
31   }
32 ]
33 }
```

Listing 36: Example JSON Schema (for hypermedia)

This is just one solution where the links are defined in an external schema. Other popular hypermedia solutions for JSON are HAL, JSON-LD, Collection+JSON, SIREN, and JSON:API which take different standardized approaches to the problem.

6.6 RESTful Web APIs

REST APIs are based on the REST (REpresentational State Transfer) architectural style as first outlined [30] in Roy Fielding's PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures". As per Roy —

An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style [30, 31].

6.6.1 REST architectural constraints

Further, as per the dissertation, REST architectural constraints emphasize component interaction scalability, component independence, encapsulate legacy components, increased interface generality, reduced latency, and enforced security [30]. These constraints are summarized below:

Client-server architecture This is guided by the Separation of Concern (SoC) principle. The idea being that separation allows the components involved ie. client and server to evolve independently of each other. Thus ultimately improving the support for Internet-scale requirements of multiple organizational domains [30].

Statelessness Each client request must have all the information needed for the server to process it in isolation of any prior request. This improves visibility, reliability, and scalability [30].

Cacheability Each server response must be labeled as cacheable or non-cacheable. This unleashes the potential to reduce or eliminate some client-serve interactions and thus improve efficiency [30].

Uniform interface This is the core constraint of REST and is defined by four interface sub constraints:

Resource identification Server resources are distinguished by client requests and are separate from the representations returned [32, 30].

Resource manipulation through representations Clients receive replies or responses that represent resources. These representations must have adequate information to enable alteration or deletion [32, 30].

Self-descriptive messages Each returned response from a server to the client comprises enough information to specify how the client should process that information [32, 30].

Hypermedia as the engine of application state (HATEOAS) After obtaining a resource, the REST client should have the possibility to discover by hyperlinks all additional actions or associated resources currently available [32, 30].

Layered System Additional layers can mediate client-server interactions. These layers could offer extra features like load balancing, shared caches, or security [32, 30].

Code on demand This constraint is optional, which basically means servers can extend the functionality of a client by transporting executable code [32, 30].

6.6.2 Interpreting REST constraints

For an API to be labeled as REST API, it must follow the constraints listed above. However, this is not so common. Many people incorrectly call any HTTP-based interface a REST API; even though REST is not tied to a specific protocol (like HTTP). As a result of this, REST has become more of a buzzword in recent times and has varied interpretations [33]. It's difficult to determine the exact reason behind this, however, few blame the abstract nature and communication style of Fielding's dissertation [34].

People have come up with further explanations of REST in response to this varied interpretation; including Fielding himself [35]. The Richardson Maturity Model (RMM) developed by Leonard Richardson in the year 2008 is another such attempt [36]. RMM is a leveled metric to categorize Web APIs based on their adherence to the REST style. As per RMM, RESTful Web APIs should:

1. expose resources,
2. via HTTP, using HTTP verbs (GET, POST, PUT, DELETE)
3. and provide links to themselves, as well as links to related resources

Although the RMM metric is helpful, it still doesn't fill the void of a standardized way to describe RESTful Web APIs. In light of this, a number of early adopters introduced their own design guidelines or recommendations. For instance:

- Microsoft's REST API Guideline: <https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md>
- PayPal's REST API Guidelines: <https://github.com/paypal/api-standards/blob/master/api-style-guide.md>

This was fine to some extent but soon became a huge inconvenience to those who worked with multiple Web APIs; each governed by their own set of evolving guidelines.

6.6.3 The OpenAPI Specification

OpenAPI Specification (OAS) is a standard API description format for REST APIs, not tied to any particular vendor. It is a super-set of JSON Schema, discussed in section 6.5.3, and allows describing

an entire API [37]. Additionally, a large ecosystem of tools built around OAS, commonly referred to as **Swagger**, can help developers design, build, document, and consume such REST APIs. OAS is by far the most popular RESTful Web API description format written in either JSON or YAML.

6.7 RESTful vs GraphQL Web APIs

This section makes a comparison of traditional REST and modern GraphQL Web APIs from a client's perspective on three specific details — endpoints involved, over-fetching, and, under-fetching. There are many examples in literature, both — formal and informal, that directly compare REST with GraphQL [38, 39]. However, it is important to acknowledge that REST is an architectural style while GraphQL is a query language and specification. Our comparison takes a practical approach by contrasting GitHub's v4 GraphQL API with GitHub's v3 OAS based REST API. Since RESTful Web APIs can follow many different specifications (besides OAS), we note exceptions in our comparison when applicable.

Consider the task of retrieving a GitHub user's name along with the names of their repositories:

Using **RESTful Web API** this task requires two steps (or requests):

- To retrieve the user's name, make a GET request to the `/users/{login}` endpoint.

```
wget --method GET 'https://api.github.com/users/ashessin'
```

```
{
  "login": "ashessin",
  "id": 6129517,
  "node_id": "MDQ6VXNlcjYxMjk1MTc=",
  "avatar_url": "https://avatars.githubusercontent.com/u/6129517?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/ashessin",
  "html_url": "https://github.com/ashessin",
  "followers_url": "https://api.github.com/users/ashessin/followers",
  "following_url": "https://api.github.com/users/ashessin/following{/other_user}",
  "repos_url": "https://api.github.com/users/ashessin/repos",
  "type": "User",
  "site_admin": false,
  "name": "Ashesh Singh",
  "company": "University of Illinois at Chicago (UIC)",
  "blog": "https://ashessin.com",
  "location": "Chicago, IL, USA",
  "created_at": "2013-12-07T12:23:17Z",
  "updated_at": "2021-03-01T08:39:45Z"
}
```

- To retrieve user's repository names, make a GET request to the `/users/{login}/repos` endpoint.

```
wget --method GET 'https://api.github.com/users/ashessin/repos'
```

```
[
  {
    "id": 221371159,
    "node_id": "MDEwOlJlcG9zaXRvcnkyMjEzNzExNTk=",
    "name": "akka-chordsim",
    "full_name": "ashessin/akka-chordsim",
    "private": false,
    "html_url": "https://github.com/ashessin/akka-chordsim",
    "description": "Akka based Chord algorithm simulation.",
    "fork": false
  },
  {
    "id": 93999959,
    "node_id": "MDEwOlJlcG9zaXRvcnk5Mzk5OTk1OQ==",
    "name": "awesome-falsehood",
    "full_name": "ashessin/awesome-falsehood",
    "private": false,
    "html_url": "https://github.com/ashessin/awesome-falsehood",
    "description": ":pill: Curated list of falsehoods programmers believe in.",
    "fork": true
  },
  {
    "id": 235682912,
    "node_id": "MDEwOlJlcG9zaXRvcnkyMzU2ODI5MTI=",
    "name": "basic-crud",
    "full_name": "ashessin/basic-crud",
    "private": false,
    "html_url": "https://github.com/ashessin/basic-crud",
    "description": "CRUD operations using PHP, MySQL and Bootstrap 4.",
    "fork": false
  }
]
```

Using GraphQL Web API this task requires a single nested selection of fields on user and repository.

```
wget --method GET
↳ 'https://api.github.com/graphql?query={user(login:"ashessin"){name,repositories(first:
↳ 3){nodes{name}}}}'
```

```
{
  "data": {
    "user": {
      "name": "Ashesh Singh",
      "repositories": {
        "nodes": [
          {
            "name": "akka-chordsim"
          },
          {
            "name": "awesome-falsehood"
          },
          {
            "name": "basic-crud"
          }
        ]
      }
    }
  }
}
```

6.7.1 API Endpoints

As shown in the examples, RESTful APIs have multiple endpoints for each resource (e.g., users, repositories). Also, RESTful APIs utilize multiple HTTP verbs (GET, PUT, POST, DELETE) on the same endpoint. However, GraphQL takes a simpler approach by utilizing a single endpoint for requests and response and is not centered around HTTP verbs. Although GraphQL requests can utilize an HTTP GET or HTTP POST [40].

6.7.2 Over-fetching

Over-fetching occurs when the server sends a response that has data the client does not require. For the GitHub API responses, the green highlights on JSON key-value pairs denote the client-required data. In case of the REST API, there is much over-fetching since multiple responses contain many additional JSON key-value pairs along with the client-required data. GraphQL, in contrast, only retrieves the client-required data and nothing more.

We note that some other API specifications such as JSON:API support sparse fields [41] and hence may not face this issue. For example, a RESTful Web API based on JSON:API can support requests for a select number of fields on the resource (similar to GraphQL).

```
wget --method GET 'https://api.github.com/rest-
↳ new/users/ashessin?include=repositories&fields[user]=name&fields[repository]=name'
```

6.7.3 Under-fetching and N+1 Requests

Under-fetching occurs when a server response does not have all the data needed by the client. As a result, clients may need to make additional requests to get such data; this is the N+1 problem. In the GitHub example, a single GraphQL request could retrieve all necessary data, while with REST, the client made two requests. The REST API endures from the N+1 problem.

Again, we note that some other REST API specifications can, in theory, address under-fetching by allowing clients to specify exact queries for the data. However, this is not possible following OAS.

7 Implementation

This chapter presents the implementation of a macro-based tool to ease the GraphQL response deserialization problem using Scala. It describes how the Annotation Macro — `mapSelection` works to transform Scala ASTs for GraphQL queries written using Caliban-client boilerplate (chapter 4). We first start with a toy, hard-coded implementation and refine it to handle more complex selections (or queries). The chapter ends with a discussion on the `illTyped` Def Macro for writing test cases.

7.1 Hard-coding the `mapSelection` Annotation Macro

This section presents a hard-coded solution to mapping as simple GraphQL selection (or query) written in Caliban-client generated boilerplate code. The target selection is shown in listing 37, while the macro implementation is shown in listing 38.

```
1 // A simple Caliban-client GraphQL selection.
2 @mapSelection
3 val userFields1: SelectionBuilder[User, GhUser] = User.bio
4
5 // macro expansion should automatically generate the below case class representation
6 // for the selection identified by userFields1
7 // case class GhUser(bio: Option[String])
```

Listing 37: A basic GraphQL selection with `mapSelection` macro

Discussion (listing 37):

- The `userFields1` immutable variable on line 3 is a selection on the `User`'s `bio`. Further, since it has the `@mapSelection` annotation, its corresponding AST will undergo expansion by the Scala compiler. This expansion will use the procedure defined in the `impl` method on line 14 of listing 38 to transform the AST.
- A corresponding Scala case class for mapping this selection could be named `GhUser` with a single parameter named `bio` and type `Option[String]`. This is shown as a comment on line 6.
- Note that the type for `userFields1` is defined as `SelectionBuilder[User, GhUser]`. Generally this would result in a compiler error because the `GhUser` class is commented out. However, since the value undergoes un-typed macro expansion, the presence of `GhUser` doesn't produce an error right-away.

```

1 package com.ashessin.cs598.macros
2
3 import scala.annotation.{ compileTimeOnly, StaticAnnotation }
4 import scala.language.experimental.macros
5 import scala.reflect.macros.whitebox
6
7 /** Annotation macro to map Caliban Selection Builder value definitions to case classes. */
8 @compileTimeOnly("Enable macro paradise to expand macro annotations.")
9 class mapSelection extends StaticAnnotation {
10   //noinspection ScalaUnusedSymbol
11   def macroTransform(annottees: Any*): Any = macro mapSelectionMacros.impl
12 }
13
14 object mapSelectionMacros {
15   def impl(c: whitebox.Context)(annottees: c.Expr[Any]*): c.Expr[c.universe.Block] = {
16     import c.universe._
17
18     val result: Tree = annottees.head.tree match {
19       case valDef: ValDef =>
20         q"""case class GhUser(bio: Option[String])
21           ${valDef.mods} val ${valDef.name}: ${valDef.tpt} =
22           ↳ ${valDef.rhs}.${TermName("map")}.${TypeName("GhUser").toTermName}
23           """
24       case t => c.abort(t.pos, "Unexpected AST.")
25     }
26
27     c.Expr[Block](result)
28   }
29 }

```

Listing 38: A basic `mapSelection` macro implementation

Discussion (listing 38):

- This listing shows the `mapSelection` Annotation Macro class and its corresponding implementation in the `impl` method, line 14.
- The logic for AST transformation is mainly dictated by lines 17–23.
- Line 17 uses pattern matching to ensure that the annotated definition’s AST is a `ValDef` which is a type for abstract representation of value definition in Scala.
- If the AST is not for a value definition, the compilation and macro expansion aborts with the compiler error message “Unexpected AST” on line 22.
- Lines 19–21 use quasiquotes for two purposes.
 - First, it creates the class representation of `GhUser` (same as that of line 7 on listing 37).

- Second, it updates the annotated value definition's (i.e., `userFields1`) RHS to map it on the generated `GhUser` class. The fields used to do so are:

valDef.mods Are the modifiers (AST type `Modifiers`) for `userFields1`, none in our case.

valDef.name It is the name or identifier (AST type `TermName`) of the value definition i.e. `userFields1`.

valDef.tpt It is the AST for the type ascribed to the definition of `userFields1`, i.e., AST for type `SelectionBuilder[User, GhUser]`.

valDef.rhs It is the AST for the right hand side of `userFields1`, i.e., AST for `User.bio`.

- The resulting transformed AST is assigned to the result value identifier with type `Tree`.
- This transformed AST is then used to create an expression block on line 25 which is in turn supplanted at the macro call location, i.e, lines 2,3 of listing 37.

An equivalent Scala code is shown in listing 39 post expansion of the `mapSelection` Annotation Macro.

```
1 case class GhUser(bio: Option[String])
2 val userFields1: SelectionBuilder[User, GhUser] = User.bio.map(GhUser)
```

Listing 39: A basic GraphQL selection post expansion of `mapSelection` macro

Although, the above macro works; it does so for a single selection, i.e., `User.bio`. This implementation is in no way general as required by the problem statement (section 1.1). If the selection changes to something like `User.id` (or more complex), it would fail to compile. This is because the macro expansion no longer produces the right class for mapping.

7.2 Parsing Constructor and Method Parameters

Before we can move to generalize the hard-coded implementation in section 7.1, we need to rework the definition of the macro class to accept parameters in its constructor. The motivation to do this is to allow user configurable macro behavior during expansion. For example, enable or disable logging. To do this, we first update our `mapSelection` macro to use a separate helper class — `Util` as shown in listing 40 and add the parameters `withLogging`, `withPrefix` to the definition.

The next step is to implement a `extractArgs` method in the `Utils` class that can retrieve parameters for arguments — `withLogging`, `withPrefix` from the macro calls site. The `MapSelectionMacros.impl` method has `macro` keyword before it. As a result of this, it is automatically invoked by the com-

```

1  /** Annotation macro to map Caliban Selection Builder value definitions to case classes. */
2  @compileTimeOnly("Enable macro paradise to expand macro annotations.")
3  class mapSelection(val withLogging: Boolean = true, val withPrefix: Boolean = true) extends
4    ↳ StaticAnnotation {
5    //noinspection ScalaUnusedSymbol
6    def macroTransform(annottees: Any*): Any = macro MapSelectionMacros.impl
7  }
8
9  class MapSelectionMacros(val c: whitebox.Context) {
10    import c.universe._
11
12    val macroUtil: Util[c.type] = Util[c.type](c)
13
14    def impl(annottees: c.Expr[Any]*): c.Expr[Block] = {
15      /** ... */
16    }
17  }

```

Listing 40: The `mapSelection` macro with parameters

```

1  @mapSelection val userFieldsConfig1: SelectionBuilder[User, _] = User.bio
2  @mapSelection(withLogging = false, withPrefix = "Github") val userFieldsConfig2:
3    ↳ SelectionBuilder[User, _] = User.bio
4  @mapSelection(withPrefix = "Github", withLogging = false) val userFieldsConfig3:
5    ↳ SelectionBuilder[User, _] = User.bio
6  @mapSelection(withLogging = false) val userFieldsConfig4: SelectionBuilder[User, _] =
7    ↳ User.bio
8  @mapSelection(withPrefix = "Github") val userFieldsConfig5: SelectionBuilder[User, _] =
9    ↳ User.bio
10 @mapSelection(false, "Github") val userFieldsConfig6: SelectionBuilder[User, _] = User.bio
11 @mapSelection(withLogging = false, "Github") val userFieldsConfig7: SelectionBuilder[User, _]
12 ↳ = User.bio

```

Listing 41: Applying `mapSelection` macro with parameters

piller with ATs of the parameters encountered at the macro call site. The method hence doesn't accept user supplied parameters for `withLogging`, `withPrefix`. In other words, the method call, `MapSelectionMacros.impl(true, false)` will fail to compile. Let's first consider all the ways in which we can apply the selection macro in listing listing 41 (pay close attention to parameter names and position). The corresponding prefix trees (untyped) for each of these are shown in listing 42.

Discussion:

- Each one of these is an Apply Tree, which is expected. The first instinct on seeing this is to make parameter names mandatory within the macro expansion logic.
- Making parameter names mandatory will ensure that we are able to receive three pieces of information that we seek: the name of the parameter, its value and type.

Based on this discussion, we add the `extractArgs` method in the `Utils` class, shown in listing 43.

```

1 Apply(Select(New(Ident(TypeName("mapSelection"))), termNames.CONSTRUCTOR), List())
2 Apply(Select(New(Ident(TypeName("mapSelection"))), termNames.CONSTRUCTOR),
  ↳ List(AssignOrNamedArg(Ident(TermName("withLogging")), Literal(Constant(false))),
  ↳ AssignOrNamedArg(Ident(TermName("withPrefix")), Literal(Constant("Github"))))
3 Apply(Select(New(Ident(TypeName("mapSelection"))), termNames.CONSTRUCTOR),
  ↳ List(AssignOrNamedArg(Ident(TermName("withPrefix")), Literal(Constant("Github"))),
  ↳ AssignOrNamedArg(Ident(TermName("withLogging")), Literal(Constant(false))))
4 Apply(Select(New(Ident(TypeName("mapSelection"))), termNames.CONSTRUCTOR),
  ↳ List(AssignOrNamedArg(Ident(TermName("withLogging")), Literal(Constant(false))))
5 Apply(Select(New(Ident(TypeName("mapSelection"))), termNames.CONSTRUCTOR),
  ↳ List(AssignOrNamedArg(Ident(TermName("withPrefix")), Literal(Constant("Github"))))
6 Apply(Select(New(Ident(TypeName("mapSelection"))), termNames.CONSTRUCTOR),
  ↳ List(Literal(Constant(false)), Literal(Constant("Github")))
7 Apply(Select(New(Ident(TypeName("mapSelection"))), termNames.CONSTRUCTOR),
  ↳ List(AssignOrNamedArg(Ident(TermName("withLogging")), Literal(Constant(false))),
  ↳ Literal(Constant("Github")))

```

Listing 42: Untyped ASTs for `mapSelection` macro application with parameters

```

1 def extractArgs[A <: Any]: Either[(Position, String), Map[String, A]] = PrefixTree match {
2   case Apply(Select(New(Ident(TypeName(_))), _), arguments) =>
3     val result = accumulateRightSeq(arguments.map {
4       // only valid tree is a labeled literal constant
5       case AssignOrNamedArg(Ident(TermName(k)), v @ Literal(Constant(_))) => Right(Map(k ->
6         ↳ evaluate(v)))
7
8       // invalid trees
9       case AssignOrNamedArg(Ident(TermName(_)), t) => return Left(t.pos,
10        ↳ error.ExpectedLiteralConstant)
11       case t @ Literal(Constant(_)) => return Left(t.pos,
12        ↳ error.ProvideParameterLabel)
13       case t @ Ident(TermName(_)) => return Left(t.pos,
14        ↳ error.ProvideLabeledLiteral)
15       case t => return Left(t.pos, error.UnexpectedTree)
16     }).right.get.flatten.toMap
17     Right(result)
18   case _ => Left(PrefixTree.pos, error.UnexpectedTree)
19 }
20
21 def evaluate[A <: Tree, B](tree: A): B = c.eval(c.Expr[B](c.untypecheck(tree.duplicate)))
22
23 protected def accumulateRightSeq[A, B]: Seq[Either[A, B]] => Either[A, List[B]] =
24   _.foldRight(Right(Nil): Either[A, List[B]])((e, acc) => acc.right.flatMap(xs =>
25     ↳ e.right.map(x => x :: xs)))

```

Listing 43: A basic `extractArgs` method

However, this approach has certain drawbacks.

- The most obvious limitation/inconvenience is that it forces the end user to specify all the parameter names and values.
- The default values specified on the macro class definition is of no use, since they are never used.
- Only literal constants are valid. Other expressions would fail (e.g., variable, if-else, etc.).

To overcome these issues, we update the `extractArgs` method in listing 45, which unlike the previous implementation, now works with typed trees shown in listing 44.

```

1 Apply(Select(New(Select(Select(Select(Select(Ident(com), com.ashessin), com.ashessin.cs598),
  ↳ com.ashessin.cs598.macros), com.ashessin.cs598.macros.selectionmapper)),
  ↳ termNames.CONSTRUCTOR), List(Select(Select(This(TypeName("dummy")),
  ↳ com.ashessin.cs598.macros.selectionmapper), TermName("$lessinit$greater$default$1")),
  ↳ Select(Select(This(TypeName("dummy")), com.ashessin.cs598.macros.selectionmapper),
  ↳ TermName("$lessinit$greater$default$2")))))
2 Apply(Select(New(Select(Select(Select(Select(Ident(com), com.ashessin), com.ashessin.cs598),
  ↳ com.ashessin.cs598.macros), com.ashessin.cs598.macros.selectionmapper)),
  ↳ termNames.CONSTRUCTOR), List(Literal(Constant(false)), Literal(Constant("Github"))))
3 Block(List(ValDef(Modifiers(ARTIFACT), TermName("x$1"), TypeTree(),
  ↳ Literal(Constant("Github"))), ValDef(Modifiers(ARTIFACT), TermName("x$2"), TypeTree(),
  ↳ Literal(Constant(false)))), Apply(Select(New(Select(Select(Select(Ident(com),
  ↳ com.ashessin), com.ashessin.cs598), com.ashessin.cs598.macros),
  ↳ com.ashessin.cs598.macros.selectionmapper)), termNames.CONSTRUCTOR),
  ↳ List(Ident(TermName("x$2")), Ident(TermName("x$1")))))
4 Apply(Select(New(Select(Select(Select(Select(Ident(com), com.ashessin), com.ashessin.cs598),
  ↳ com.ashessin.cs598.macros), com.ashessin.cs598.macros.selectionmapper)),
  ↳ termNames.CONSTRUCTOR), List(Literal(Constant(false)),
  ↳ Select(Select(This(TypeName("dummy")), com.ashessin.cs598.macros.selectionmapper),
  ↳ TermName("$lessinit$greater$default$2"))))
5 Block(List(ValDef(Modifiers(ARTIFACT), TermName("x$3"), TypeTree(),
  ↳ Literal(Constant("Github"))), ValDef(Modifiers(ARTIFACT), TermName("x$4"), TypeTree(),
  ↳ Select(Select(This(TypeName("dummy")), com.ashessin.cs598.macros.selectionmapper),
  ↳ TermName("$lessinit$greater$default$1")))),
  ↳ Apply(Select(New(Select(Select(Select(Select(Ident(com), com.ashessin),
  ↳ com.ashessin.cs598), com.ashessin.cs598.macros),
  ↳ com.ashessin.cs598.macros.selectionmapper)), termNames.CONSTRUCTOR),
  ↳ List(Ident(TermName("x$4")), Ident(TermName("x$3")))))
6 Apply(Select(New(Select(Select(Select(Select(Select(Ident(com), com.ashessin),
  ↳ com.ashessin.cs598), com.ashessin.cs598.macros), com.ashessin.cs598.macros.dummy),
  ↳ com.ashessin.cs598.macros.selectionmapper)), termNames.CONSTRUCTOR),
  ↳ List(Literal(Constant(false)), Literal(Constant("Github"))))
7 Apply(Select(New(Select(Select(Select(Select(Select(Ident(com), com.ashessin),
  ↳ com.ashessin.cs598), com.ashessin.cs598.macros), com.ashessin.cs598.macros.dummy),
  ↳ com.ashessin.cs598.macros.selectionmapper)), termNames.CONSTRUCTOR),
  ↳ List(Literal(Constant(false)), Literal(Constant("Github"))))

```

Listing 44: Typed ASTs for `mapSelection` macro application with parameters

Discussion (listing 44), as per Scala documentation:

- For named arguments, the argument order does not have to match the parameter order of the method definition. To evaluate the argument expressions in call-site order, the method application is transformed to a block [42] as seen on lines 3,5.
- For every default argument expression the compiler generates a method computing that expression. These methods have deterministic names composed of the method name, the string "\$default\$" and a number indicating the parameter position. Each method is parameterized

by the type parameters of the original method and by the value parameter sections preceding the corresponding parameter [42] as seen on lines 1,4,5.

```

1 def extractArgs[A <: Any](tree: Tree): Either[(Position, String), Map[String, A]] = tree
  ⇨ match {
2   case Apply(select @ Select(_, _), l1) ⇒
3     val parameters: Seq[String] =
4       ⇨ select.symbol.typeSignature.paramLists.head.map(_.name.toString)
5     val arguments: Map[String, A] = accumulateRightSeq(l1.zipWithIndex map {
6       case (t: Literal, i: Int) ⇒ Right(Map(parameters(i) → evaluate(t)))
7       case (t: Select, i: Int)  ⇒ Right(Map(parameters(i) → evaluate(t)))
8       case (t: Apply, i: Int)   ⇒ Right(Map(parameters(i) → evaluate(t)))
9       case (t, _)              ⇒ println(showRaw(t)); return Left(t.pos,
10        ⇨ error.UnexpectedTree)
11    }).right.get.flatten.toMap
12    Right(arguments)
13   case Apply(TypeApply(select @ Select(_, _), _), l1) ⇒
14     val parameters: Seq[String] =
15       ⇨ select.symbol.typeSignature.paramLists.head.map(_.name.toString)
16     val arguments: Map[String, A] = accumulateRightSeq(l1.zipWithIndex map {
17       case (t: Literal, i: Int) ⇒ Right(Map(parameters(i) → evaluate(t)))
18       case (t: Select, i: Int)  ⇒ Right(Map(parameters(i) → evaluate(t)))
19       case (t: Apply, i: Int)   ⇒ Right(Map(parameters(i) → evaluate(t)))
20       case (t: TypeApply, i: Int) ⇒ Right(Map(parameters(i) → evaluate(t)))
21       case (t, _)              ⇒ println(showRaw(t)); return Left(t.pos,
22        ⇨ error.UnexpectedTree)
23    }).right.get.flatten.toMap
24    Right(arguments)
25   case Block(l1, Apply(select @ Select(_, _), l2)) ⇒
26     val parameters: Seq[String] =
27       ⇨ select.symbol.typeSignature.paramLists.head.map(_.name.toString)
28     val idents: Map[String, Name] = accumulateRightSeq(l2.zipWithIndex map {
29       case (t: Ident, i: Int) ⇒ Right(Map(parameters(i) → t.name))
30       case (t, _)           ⇒ println(showRaw(t)); return Left(t.pos, error.UnexpectedTree)
31    }).right.get.flatten.toMap
32     val valDefs: Map[Name, A] = accumulateRightSeq(l1 map {
33       case t: ValDef ⇒ Right(Map(t.name → evaluate(t.rhs)))
34       case t         ⇒ println(showRaw(t)); return Left(t.pos, error.UnexpectedTree)
35    }).right.get.flatten.toMap
36     val arguments = idents.flatMap(ident ⇒ Map(ident._1 → valDefs(ident._2)))
37     Right(arguments)
38   case t ⇒ println(showRaw(t)); Left(t.pos, error.UnexpectedTree)
39 }

```

Listing 45: Updated 1 to extractArgs method

Discussion (listing 45):

- The lines 2–10, handle Apply Trees 1, 2, 4, 6, 7 of listing 44.
- The lines 22–33, handle Block Trees of listing 44.

- Lastly, lines 12–21 of are for scenarios when the method or class definition has a type parameter in its signature — Polymorphic Methods and Generic Classes i.e., Parameterized Types.

This new approach overcomes the limitations of the previous one and can process or in other words extract name, value and type of named and default arguments with ease. However, this solution is messy and it's possible that the matches are non-exhaustive (i.e., some cases were missed). The listing 46 resolves these issues by directly evaluating the AST for parenthesized calls (i.e., method or class constructor calls).

```

1  /** Extracts values from a tree for method or constructor call.
2  *
3  * @param tree an Apply or Block tree
4  * @tparam A    type of the value
5  * @return      an ordered map of argument names and their values
6  */
7  def extractArgs[A <: Any](tree: Tree): Either[(Position, String), ListMap[String, A]] = {
8    def paramsFromApplyTree(select: Select, l: List[Tree]): ListMap[String, A] = {
9      val arguments = select.symbol.typeSignature.paramLists.head.map(_.name.toString)
10     l.zipWithIndex.map { case (t: Tree, i: Int) => ListMap(arguments(i) -> evaluate(t).get) }
11     .reduce(_ ++ _)
12   }
13   def paramsFromBlockTree(select: Select, l1: List[Tree], l2: List[Tree]): ListMap[String, A]
14   => = {
15     val arguments = select.symbol.typeSignature.paramLists.head.map(_.name.toString)
16     val valDefs   = l2.flatMap { case t: ValDef => ListMap(t.name -> evaluate(t.rhs).get) }
17     .toMap
18     l1.zipWithIndex.map { case (t: Ident, i: Int) => ListMap(arguments(i) -> evaluate(t).get) }
19     .toMap
20     .reduce(_ ++ _)
21     .flatMap(ident => ListMap(ident._1 -> valDefs(ident._2)))
22   }
23   util
24   .Try(tree match {
25     case Apply(s @ Select(_, _), l)           => paramsFromApplyTree(s, l)
26     case Apply(TypeApply(s @ Select(_, _), _), l) => paramsFromApplyTree(s, l)
27     case Block(l1, Apply(s @ Select(_, _), l2)) => paramsFromBlockTree(s, l1, l2)
28   })
29   .fold(fa => Left(tree.pos, fa.getMessage), Right(_))
30 }

```

Listing 46: Updated 2 to `extractArgs` method

7.3 Generalizing the `mapSelection` Annotation Macro

As seen on line 19 of listing 38, case classes generation is straightforward when all the required information (e.g. class name, parameter name/type) is readily available. However, extracting this information or the subgraph as discussed in section 2.1.3 is challenging since the implementation should be able to handle all types of selection. Despite this, one can make a few observations

regarding the structure of a selection AST. For example, the right hand side of a value definition with unit selection (ie. no use of `~` operator) and type `SelectionBuilder[Origin, A]` will always be either a member selection, eg. `User.id`, `User.bio` or a value application, eg. `User.avatarUrl(42)`, `User.status(UserStatus.id)`, `User.repository("repo-name")(Repository.id)`. These ASTs are of two distinct types:

Select Tree AST node that correspond to the following Scala code format `qualifier.name`. These are the boilerplate Object methods that do not take any parameters.

```
def id: SelectionBuilder[User, String] = Field("id", Scalar())
```

Apply Tree AST node that corresponds to the following Scala code format `fun(args)`. These are the boilerplate Object methods that take arguments. However, since ASTs are nested structures, there are many kinds of such trees depending on the field selection:

The simple Apply tree These are Monomorphic (ie. not parameterized by type) boilerplate methods that take zero or more arguments. eg. `User.avatarUrl()`, `User.avatarUrl(Option(36))`

```
def avatarUrl(size: Option[Int] = None): SelectionBuilder[User, URI] =  
  ↪ Field("avatarUrl", Scalar(), arguments = List(Argument("size", size)))
```

The Apply-Typeapply tree These are Polymorphic methods (ie. definition parameterized by type) that require argument and don't exhibit currying ie. have single argument list. eg. `User.status(UserStatus.id)`

```
def status[A](innerSelection: SelectionBuilder[UserStatus, A]): SelectionBuilder[User,  
  ↪ Option[A]] = Field("status", OptionOf(Obj(innerSelection)))
```

The Apply-Apply-TypeApply tree These are Polymorphic methods (ie. definition parameterized by type) that require argument and exhibit currying ie. have multiple argument lists. eg. `User.repository("repo-name")(Repository.id)`

```
def repository[A](name: String)(innerSelection: SelectionBuilder[Repository, A]):  
  ↪ SelectionBuilder[User, Option[A]] = Field("repository",  
  ↪ OptionOf(Obj(innerSelection)), arguments = List(Argument("name", name)))
```

The next few sections explore macro implementation procedure that handle Select, Apply ASTs and their nested types. To do this we create a new `traverse` method within the macro implementation that accepts a single typed AST as it's input.

7.4 Traversing Select ASTs

The listing 47 shows several unit Select ASTs which are of the format — `qualifier.name`. Scala's reflection API method — `scala.reflect.api.Trees.SelectExtractor#unapply` can extract the `qualifier` and `name`.

```

1 @mapSelection val userFields1a: SelectionBuilder[User, _] = User.id
2 // tree-code: com.Github.User.id
3 // tree-raw: Select(Select(Select(Ident(com), com.Github), com.Github.User), TermName("id"))
4
5 @mapSelection val userFields1b: SelectionBuilder[User, _] = User.createdAt
6 // tree-code: com.Github.User.createdAt
7 // tree-raw: Select(Select(Select(Ident(com), com.Github), com.Github.User),
8   ↪ TermName("createdAt"))
9
10 @mapSelection val userFields1c: SelectionBuilder[User, _] = User.company
11 // tree-code: com.Github.User.company
12 // tree-raw: Select(Select(Select(Ident(com), com.Github), com.Github.User),
13   ↪ TermName("company"))
14
15 @mapSelection val userFields1d: SelectionBuilder[User, _] = User.isEmployee
16 // tree-code: com.Github.User.isEmployee
17 // tree-raw: Select(Select(Select(Ident(com), com.Github), com.Github.User),
18   ↪ TermName("isEmployee"))

```

Listing 47: Select ASTs from GitHub boilerplate

Selection:

`User.id`

AST:

`Select(Select(Select(Ident(com), com.Github), com.Github.User), TermName("id"))`

Figure 7.1: Components of the **Select** AST

```

1 protected def traverse(tree: Tree): Unit = {
2   if (hasLoggingEnabled) println(s"tree-code: ${showCode(tree)}")
3   if (hasLoggingEnabled) println(s"tree-raw: ${showRaw(tree)}")
4
5   tree match {
6
7     // For Select AST
8     case select @ Select(qualifier, name) =>
9       val caseClassName      = qualifier.symbol.name
10      val caseClassParameter = (name.toTypeName, select.tpe.finalResultType.typeArgs.last)
11      println(caseClassName, caseClassParameter)
12
13   }
14 }

```

Listing 48: Pattern matching **Select** ASTs

Discussion (listing 48):

- The input tree is pattern matched to extract qualifier (with type `Trees#Tree`) and name (with type `Names#Name`). These are a part of the Scala Reflection API.

- Line 9 retrieves the name for the qualifier which will be the name for the case class representation.
- Line 10 retrieves the parameter name and type for the case class representation. The retrieved parameter type corresponds to the last type in the method definition's result type.

```
def id: SelectionBuilder[User, String] = Field("id", Scalar())

def createdAt: SelectionBuilder[User, DateTime] = Field("createdAt", Scalar())

def company: SelectionBuilder[User, Option[String]] = Field("company", OptionOf(Scalar()))

def isEmployee: SelectionBuilder[User, Boolean] = Field("isEmployee", Scalar())
```

- Extracted details are shown as console outputs in listing 49.

```
1 (User,(id,String))
2 (User,(createdAt,com.Github.DateTime))
3 (User,(company,Option[String]))
4 (User,(isEmployee,Boolean))
```

Listing 49: Output for pattern matching `Select` ASTs

7.5 Traversing `Apply` ASTs

This section details the processing of `Apply` ASTs which the macro application may encounter. The simplest `Apply` AST are for Monomorphic (ie. not parameterized by type) methods that take an optional argument as shown in listing 50.

```
1 @mapSelection val userFields2a: SelectionBuilder[User, _] = User.avatarUrl()
2 // tree-code: com.Github.User.avatarUrl(com.Github.User.avatarUrl$default$1)
3 // tree-raw: Apply(Select(Select(Select(Ident(com), com.Github), com.Github.User),
4   ↳ TermName("avatarUrl")), List(Select(Select(Select(Ident(com), com.Github),
5   ↳ com.Github.User), TermName("avatarUrl$default$1"))))
6
7 @mapSelection val userFields2b: SelectionBuilder[User, _] = User.avatarUrl(Option(42))
8 // tree-code: com.Github.User.avatarUrl(scala.Option.apply[Int](42))
9 // tree-raw: Apply(Select(Select(Select(Ident(com), com.Github), com.Github.User),
10   ↳ TermName("avatarUrl")), List(Apply(TypeApply(Select(Select(Ident(scala), scala.Option),
11   ↳ TermName("apply")), List(TypeTree()), List(Literal(Constant(42)))))
```

Listing 50: `Apply` ASTs from GitHub boilerplate

Discussion (listing 50)

- Lines 3 shows the raw AST for Scala code `User.avatarUrl()`.
- Lines 7 shows the raw AST for Scala code `User.avatarUrl(Option(42))`.
- Notice that both of these ASTs enclose a `Select` AST.

Selection:

```
User.avatarUrl()
```

AST:

```
Apply(Select(Select(Select(Ident(com), com.Github), com.Github.User),
TermName("avatarUrl")), List(Select(Select(Select(Ident(com), com.Github),
com.Github.User), TermName("avatarUrl$default$1"))))
```

(a) Apply AST, with default value

Selection:

```
User.avatarUrl(Option(42))
```

AST:

```
Apply(Select(Select(Select(Ident(com), com.Github), com.Github.User),
TermName("avatarUrl")), List(Apply(TypeApply(Select(Select(Ident(scala),
scala.Option), TermName("apply")), List(TypeTree()))),
List(Literal(Constant(42))))))
```

(b) Apply AST, with custom value

Figure 7.2: Components of the **Apply** AST

- We have already established how to parse ASTs with `Select` as root in section 7.4. Additionally, we showed how to extract value, name, and type of parameters from an `Apply` tree in section 7.2.

Based on the above discussion we update the `traverse` method to pattern match on `Apply` ASTs as shown in listing 51. As a result of these changes, the `traverse` function is now recursive. The output is shown in listing 52.

```

1  protected def traverse(tree: Tree): Unit = {
2    if (hasLoggingEnabled) println(s"tree-code: ${showCode(tree)}")
3    if (hasLoggingEnabled) println(s"tree-raw: ${showRaw(tree)}")
4
5    tree match {
6
7      // For Select AST
8      case select @ Select(qualifier, name) =>
9        val caseClassName      = qualifier.symbol.name
10       val caseClassParameter = (name.toTypeName, select.tpe.finalResultType.typeArgs.last)
11       println(caseClassName, caseClassParameter)
12
13      // For Apply AST
14      case apply @ Apply(fun @ Select(_, _), _) =>
15        println(macrosUtil.extractArgs(apply))
16        traverse(fun)
17
18    }
19  }

```

Listing 51: Pattern matching **Apply** ASTs

```

1  Right(ListMap(size -> None))
2  (User,(avatarUrl,com.Github.URI))
3  Right(ListMap(size -> Some(42)))
4  (User,(avatarUrl,com.Github.URI))

```

Listing 52: Output for pattern matching **Apply** ASTs

7.6 Traversing **Apply-TypeApply** ASTs

The next form of an Apply tree is one that accepts another selection as parameter. Consider this macro usage in listing 53.

```

1  @mapSelection val userFields3: SelectionBuilder[User, _] = User.status(UserStatus.id)
2  // tree-code: com.Github.User.status[String](com.Github.UserStatus.id)
3  // tree-raw: Apply(TypeApply(Select(Select(Select(Ident(com), com.Github), com.Github.User),
4  ↪ TermName("status")), List(TypeTree()), List(Select(Select(Select(Ident(com), com.Github),
5  ↪ com.Github.UserStatus), TermName("id"))))

```

Listing 53: **Apply-TypeApply** ASTs from GitHub boilerplate

Discussion (listing 53):

- Both User and UserStatus are GraphQL types (Objects).
- This tree is quite similar to the Apply tree in section 7.5. The only difference here is that User.status is a Polymorphic method unlike User.avatarUrl which was Monomorphic.

Selection:

```
User.status(UserStatus.id)
```

AST:

```
Apply(TypeApply(Select(Select(Select(Ident(com), com.Github), com.Github.User),
TermName("status")), List(TypeTree()), List(Select(Select(Select(Ident(com),
com.Github), com.Github.UserStatus), TermName("id")))))
```

Figure 7.3: Components of the **Apply-TypeApply** AST

- We can handle this by adding another match statement on line 19 and using recursive calls to the `traverse` method on lines 20,21.

Based on this discussion, we update the `traverse` function with an additional pattern match as shown in listing 54. This produces the desired output of listing 55

```

1  protected def traverse(tree: Tree): Unit = {
2    if (hasLoggingEnabled) println(s"tree-code: ${showCode(tree)}")
3    if (hasLoggingEnabled) println(s"tree-row: ${showRaw(tree)}")
4
5    tree match {
6
7      // For Select AST
8      case select @ Select(qualifier, name) =>
9        val caseClassName      = qualifier.symbol.name
10       val caseClassParameter = (name.toTypeName, select.tpe.finalResultType.typeArgs.last)
11       println(caseClassName, caseClassParameter)
12
13      // For Apply AST
14      case apply @ Apply(fun @ Select(_, _), _) =>
15        println(macroUtil.extractArgs(apply))
16        traverse(fun)
17
18      // For Apply-TypeApply AST
19      case Apply(TypeApply(fun, _), args @ List(_*)) =>
20        traverse(fun)
21        args.foreach(traverse)
22
23    }
24  }
```

Listing 54: Pattern matching **Apply-TypeApply** ASTs

```

1  (User,(status,Option[A]))
2  (UserStatus,(id,String))
```

Listing 55: Output for pattern matching **Apply-TypeApply** ASTs

7.7 Traversing Apply-Apply-TypeApply ASTs

This is the final Apply tree form that we need to parse. Consider this macro usage from listing 56

```
1 @mapSelection val userFields4: SelectionBuilder[User, _] =
  ↳ User.repository("repo-name")(Repository.id)
2 // tree-code: com.Github.User.repository[String]("repo-name")(com.Github.Repository.id)
3 // tree-raw: Apply(Apply(TypeApply(Select(Select(Select(Ident(com), com.Github),
  ↳ com.Github.User), TermName("repository")), List(TypeTree()))),
  ↳ List(Literal(Constant("repo-name"))), List(Select(Select(Select(Ident(com), com.Github),
  ↳ com.Github.Repository), TermName("id"))))
```

Listing 56: Apply-Apply-TypeApply ASTs from GitHub boilerplate

Selection:

```
User.repository("repo-name")(Repository.id)
```

AST:

```
Apply(Apply(TypeApply(Select(Select(Select(Ident(com), com.Github),
com.Github.User), TermName("repository")), List(TypeTree())),
List(Literal(Constant("repo-name"))), List(Select(Select(Select(Ident(com),
com.Github), com.Github.Repository), TermName("id"))))
```

Figure 7.4: Components of the Apply-Apply-TypeApply AST

Discussion (listing 56):

- This AST is an amalgamation of the Apply and Apply-TypeApply trees seen earlier (sections 7.5 and 7.6).
- Here, the polymorphic method `User.repository` has two parameter lists instead of one and the final result type depends on the parameter to the second list.
- To tackle this, we yet again add a match statement on line 24 and use recursion.

Based on this discussion, we present the updated traverse method in listing 57. The output is shown in listing 58.

```

1  protected def traverse(tree: Tree): Unit = {
2    if (hasLoggingEnabled) println(s"tree-code: ${showCode(tree)}")
3    if (hasLoggingEnabled) println(s"tree-raw: ${showRaw(tree)}")
4
5    tree match {
6
7      // For Select AST
8      case select @ Select(qualifier, name) =>
9        val caseClassName      = qualifier.symbol.name
10       val caseClassParameter = (name.toTypeName, select.tpe.finalResultType.typeArgs.last)
11       println(caseClassName, caseClassParameter)
12
13      // For Apply AST
14      case apply @ Apply(fun @ Select(_, _), _) =>
15        println(macroUtil.extractArgs(apply))
16        traverse(fun)
17
18      // For Apply-TypeApply AST
19      case Apply(TypeApply(fun, _), args @ List(_*)) =>
20        traverse(fun)
21        args.foreach(traverse)
22
23      // For Apply-Apply-TypeApply AST
24      case Apply(fun1 @ Apply(TypeApply(fun2, _), _), args @ List(_*)) =>
25        println(macroUtil.extractArgs(fun1))
26        traverse(fun2)
27        args.foreach(traverse)
28
29    }
30  }

```

Listing 57: Pattern matching **Apply-Apply-TypeApply** ASTs

```

1  Right(ListMap(name → repo-name))
2  (User,(repository,Option[A]))
3  (Repository,(id,String))

```

Listing 58: Output for pattern matching **Apply-Apply-TypeApply** ASTs

7.8 Accounting for Selection Combination ASTs

In the previous sections we worked with unit selections. However most selections are a combination. For example, `User.id ~ User.bio`; using the `~` (i.e. `$tilde`) operator for combining compatible origins. Consider this macro usage in listing 59.

```

1 @mapSelection val userFields5: SelectionBuilder[User, _] = User.bio ~
  ↳ User.repository("repo-name")(Repository.id)
2 // tree-code: com.Github.User.bio.~[com.Github.User,
  ↳ Option[String]](com.Github.User.repository[String]("repo-
  ↳ name")(com.Github.Repository.id))
3 // tree-raw: Apply(TypeApply(Select(Select(Select(Select(Ident(com), com.Github),
  ↳ com.Github.User), TermName("bio")), TermName("$tilde")), List(TypeTree(), TypeTree()),
  ↳ List(Apply(Apply(TypeApply(Select(Select(Select(Ident(com), com.Github), com.Github.User),
  ↳ TermName("repository")), List(TypeTree()), List(Literal(Constant("repo-name")))),
  ↳ List(Select(Select(Select(Ident(com), com.Github), com.Github.Repository),
  ↳ TermName("id"))))))))

```

Listing 59: Selection combination AST

Multiple field selections on GraphQL Types:

```
User.bio ~ User.repository("repo-name")(Repository.id)
```

AST:

```

Apply(TypeApply(Select(Select(Select(Select(Ident(com), com.Github),
com.Github.User), TermName("bio")), TermName("$tilde")), List(TypeTree(),
TypeTree()), List(Apply(Apply(TypeApply(Select(Select(Select(Ident(com),
com.Github), com.Github.User), TermName("repository")), List(TypeTree()),
List(Literal(Constant("repo-name")))), List(Select(Select(Select(Ident(com),
com.Github), com.Github.Repository), TermName("id"))))))))

```

Figure 7.5: Components of a Compound Selection

Discussion (listing 59, listing 60):

- Inspecting the AST in listing 59, we can see that it's a concatenation of two Apply trees.
- They are separated by the Select tree for \$tilde.
- We can account for this by modifying the pattern matcher as shown in listing 60, line 8.

```

1  protected def traverse(tree: Tree): Unit = {
2    if (hasLoggingEnabled) println(s"tree-code: ${showCode(tree)}")
3    if (hasLoggingEnabled) println(s"tree-raw: ${showRaw(tree)}")
4
5    tree match {
6
7      // For multiple selections combined using ~ operator
8      case Select(qualifier, name) if name == TermName("$tilde") => this.traverse(qualifier)
9
10     // For Select AST
11     case select @ Select(qualifier, name) if qualifier.symbol.isModule =>
12       val caseClassName      = qualifier.symbol.name
13       val caseClassParameter = (name.toTypeName, select.tpe.finalResultType.typeArgs.last)
14       println(caseClassName, caseClassParameter)
15
16     // For Apply AST
17     case apply @ Apply(fun @ Select(_, _), _) =>
18       println(macroUtil.extractArgs(apply))
19       traverse(fun)
20
21     // For Apply-TypeApply AST
22     case Apply(TypeApply(fun, _), args @ List(_*)) =>
23       traverse(fun)
24       args.foreach(traverse)
25
26     // For Apply-Apply-TypeApply AST
27     case Apply(fun1 @ Apply(TypeApply(fun2, _), _), args @ List(_*)) =>
28       println(macroUtil.extractArgs(fun1))
29       traverse(fun2)
30       args.foreach(traverse)
31
32   }
33 }

```

Listing 60: Selection combination AST

7.9 Testing the `mapSelection` Annotation Macro

The `ScalaTest` package provides DSL to check a code snippet for compile time error via: `should compile`, `should typeCheck`. We can use them to see if our macro expansion produces any errors as shown in listing 61.

Discussion (listings 61 and 62):

- Test cases in listing 61 check for compilation error during macro expansion.
- However, one drawback of this method is that it does not check the underlying reason for failure.
- There are more than one way in which our macros could fail to compile and it would be nice to have a way to check for a specific case.

```

1 class MapSelectionTest extends AnyFlatSpec with Matchers {
2   "MapSelection Macro Annotation" should "compile when applied to a value definition of
   ↳ correct form" in {
3     ""
4     @mapSelection val userFields: caliban.client.SelectionBuilder[com.Github.User, _] =
5       com.Github.User.login
6     "" should compile
7     ""
8     @mapSelection val userFields: caliban.client.SelectionBuilder[com.Github.User, _] =
9       com.Github.User.login ~ com.Github.User.bio ~
10      com.Github.User.email
11     "" should compile
12     ""
13     @mapSelection val userFields =
14       com.Github.User.login ~ com.Github.User.bio ~
15       com.Github.User.email
16     "" should compile
17     ""
18     @mapSelection val userFields = com.Github.User.login
19     "" should compile
20   }
21 }

```

Listing 61: Macro testing using ScalaTest package

- We came across a Def Macro implementation in the Shapeless library, which overcomes this issue by validating underlying compile-time errors (i.e., those emitted by `c.error`) and hence can be used to define the behavior of macros more precisely in tests.
- A modified version of this Def Macro is shown in listing 62. We made changes to when error messages are shown and to the regex functionality.
- This `illTyped` Def Macro accepts two parameters — a code snippet (possibly, with some error) and an expected error message.
- If the compilation fails, it compares the compiler's error message with the expected error message to check if they are identical.

```

1  /** A utility which checks if a code fragment typecheck.
2   *
3   * Copyright (c) 2013-16 Miles Sabin, Apache License, Version 2.0
4   * Credit: Stefan Zeiger (@StefanZeiger)
5   */
6  object illTyped {
7    def apply(code: String): Unit = macro IllTypedMacros.implNoExp
8    def apply(code: String, expectedError: String): Unit = macro IllTypedMacros.impl
9  }
10
11  class IllTypedMacros(val c: blackbox.Context) {
12    import c.universe._
13
14    def patternPredicate(regex: String): Predicate[String] =
15      Pattern.compile(regex, Pattern.CASE_INSENSITIVE | Pattern.DOTALL).asMatchPredicate()
16
17    def typecheck(code: String): scala.util.Try[Tree] = {
18      val tname1: TermName = TermName(c.freshName())
19      val tname2: TermName = TermName(c.freshName())
20      scala.util.Try(c.typecheck(c.parse(s"object $tname1 { val $tname2 = { $code } }")))
21    }
22
23    def implNoExp(code: Tree): Tree = impl(code, null)
24
25    def impl(code: Tree, expectedError: Tree): Tree = {
26      val Literal(Constant(kode: String)) = code
27      val (isUnexpectedError, expected) = expectedError match {
28        case null => (patternPredicate(".*").negate(), "Expected some type-checking error.")
29        case Literal(Constant(s: String)) =>
30          (patternPredicate(s).negate(), s"Expected error matching: ${showRaw(s)}")
31      }
32      typecheck(kode) match {
33        case Failure(e: TypecheckException) if !isUnexpectedError.test(e.getMessage) =>
34        case Failure(e: TypecheckException) if isUnexpectedError.test(e.getMessage) =>
35          c.error(
36            c.enclosingPosition,
37            s"""Type-checking failed, but in an unexpected way.
38              |${e.getMessage}
39              |$expected""".stripMargin
40          )
41        case Failure(e: ParseException) =>
42          c.error(
43            c.enclosingPosition,
44            s"""Unable to type-check due to parsing error.
45              |${e.getMessage}
46              |$expected""".stripMargin
47          )
48        case Success(_) =>
49          c.error(c.enclosingPosition, s"Type-checking succeeded unexpectedly. $expected")
50      }
51      q"()"
52    }
53  }

```

Listing 62: Implementation for the `illTyped` Def Macro

8 Conclusion

This work presented a compile-time metaprogramming-based solution to address the impedance mismatch problem when working with a GraphQL API. The solution aims to ease the development of GraphQL client applications and increase developer productivity while maintaining high program safety standards through compile-time type-checking. Thus in essence, creating adaptive GraphQL client applications. In this chapter, we compare our implementation with another macro-based library for GraphQL. We also recognize some current limitations of our implementation.

8.1 Comparison with Existing Solutions

There are many GraphQL client libraries that implement the GraphQL specification [43]. However, these are limited to addressing the problem of type-checking GraphQL queries. They do not offer any safe mechanism to deserialize GraphQL response. The only library that is comparable to our implementation is `graphql-client`¹ written in Rust programming language. This library also takes a macro-based approach to both — type-check GraphQL query and deserialize GraphQL response. However, it has some constraints:

- At this point, Rust is a new programming language and does not have wide-scale adoption comparable to other traditional languages. Our implementation is more suitable for Java/Scala projects with broader adoption.
- GraphQL queries are type-checked by the library but are external to the Rust programming language and written in plain text files. As a result, the developer can not take any advantage of IDE hints and warnings when creating the queries. Our implementation on top of Caliban-client generated boilerplate better integrates with the overall source code (in Scala).
- The library offers its functionality for use in the web browser directly. However, running this library in the web browser requires WebAssembly, which has limited browser support [44]. Our implementation may be easily compiled to JavaScript to enable similar functionality with less complexity and better browser support.

¹`graphql-client` — <https://github.com/graphql-rust/graphql-client>

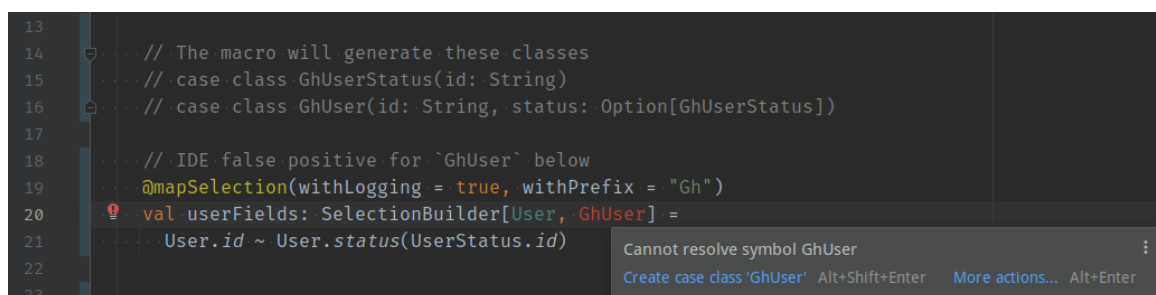


Figure 8.1: False positive “missing type or symbol” warning in IntelliJ IDE

8.2 Limitations

This section notes four limitations of our current implementation and lists any possible mitigation techniques to these limitations.

- **Scala** macros must compile before use in a separate compilation unit. This requirement is generally not an issue when multi-project builds or external dependencies are allowed. However, if that is not the case, our macro-based implementation can not be used.
- It is essential to recognize that, in our implementation, class definitions for response deserialization are available only after source code compilation. As a result of this, they are not entirely transparent to the developer (or the IDE) during the development phase, i.e., when writing the code. IDEs will produce false-positive warnings for missing class definitions referred-to in the source code during development (fig. 8.1), although the code would compile without any issues. Developers can mitigate this to some extent by using a base interface (or **Scala** trait) as stand-ins for those representations. Another solution is to disable IDE inspection on a per-file or per-line basis for such false positives. However, it may not be ideal for all situations.
- Extending a **Scala** macro implementation is not straight-forward since it requires a significant amount of time to learn. Our implementation is no exception. We can aim to provide the most sought functionalities that developers may seek so that they do not feel the need to extend the macro implementation (themselves).
- **Scala** macros have remained an experimental feature ever since their release in **Scala** 2.10. This experimental tag implies that they can change significantly in the future. There is also a risk of partial or complete deprecation of Annotation Macros that our implementation uses, with an upcoming **Scala** release such as **Scala** 3, code-named Dotty. In this situation, the implementation would require porting to a separate entity such as an IDE extension or **sbt** plugin, possibly based on the **ScalaMeta** library, which has an almost equivalent API for working with **Scala**

ASTs. However, this is not the most convenient solution (section 2.3.1) and requires additional effort to implement.

CITED LITRATURE

- [1] The GraphQL Foundation — graphql.org. **The GraphQL Foundation | An open and neutral home for the GraphQL community**. URL: <https://foundation.graphql.org/> (visited on 02/25/2021) (cit. on p. 1).
- [2] The Linux Foundation — linuxfoundation.org. **The Linux Foundation Announces Intent to Form New Foundation to Support GraphQL - Linux Foundation**. URL: https://www.linuxfoundation.org/press-release/2018/11/intent_to_form_graphql/ (visited on 02/25/2021) (cit. on p. 1).
- [3] The GraphQL Foundation — graphql.org. **GraphQL Landscape**. URL: <https://landscape.graphql.org/card-mode?category=graph-ql-adopter&grouping=category> (visited on 02/25/2021) (cit. on p. 1).
- [4] The GraphQL Foundation — graphql.org. **Thinking in Graphs | GraphQL**. URL: <https://graphql.org/learn/thinking-in-graphs/#it-s-graphs-all-the-way-down-https-en-wikipedia-org-wiki-turtles-all-the-way-down> (visited on 02/25/2021) (cit. on p. 1).
- [5] C Ireland et al. “A Classification of Object-Relational Impedance Mismatch”. eng. In: **2009 First International Conference on Advances in Databases, Knowledge, and Data Applications**. IEEE, 2009, pp. 36–43. ISBN: 142443467X (cit. on p. 2).
- [6] The GraphQL Foundation — graphql.org. **GraphQL**. URL: <https://spec.graphql.org/June2018/> (visited on 02/25/2021) (cit. on pp. 5, 13).
- [7] Erik Wittern et al. “An empirical study of GraphQL schemas”. In: **International Conference on Service-Oriented Computing**. Springer, 2019, pp. 3–19 (cit. on p. 7).
- [8] OWASP Foundation Inc. — owasp.org. **A8:2017-Insecure Deserialization | OWASP**. URL: https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization (visited on 02/25/2021) (cit. on p. 8).
- [9] Sondre Forland Fingann. “Java Deserialization Vulnerabilities”. MA thesis, 2020 (cit. on p. 8).
- [10] Stefano Cristalli et al. “Trusted Execution Path for Protecting Java Applications Against Deserialization of Untrusted Data”. eng. In: **Research in Attacks, Intrusions, and Defenses**. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 445–464. ISBN: 3030004694 (cit. on p. 8).
- [11] Pierre Ernst — ibm.com. **Look-ahead Java deserialization**. URL: <https://www.ibm.com/developerworks/library/se-lookahead/index.html> (visited on 02/25/2021) (cit. on p. 8).
- [12] Yannis Lilis and Anthony Savidis. “A Survey of Metaprogramming Languages”. eng. In: *ACM computing surveys* 52.6 (2020), pp. 1–39. ISSN: 0360-0300 (cit. on pp. 9, 10).
- [13] GitHub — github.blog. **The GitHub GraphQL API - The GitHub Blog**. URL: <https://github.blog/2016-09-14-the-github-graphql-api/> (visited on 12/25/2020) (cit. on p. 12).
- [14] Pierre Ricadat — caliban. **GraphQL Client · Issue #213 · ghostdogpr/caliban**. URL: <https://github.com/ghostdogpr/caliban/issues/213> (visited on 12/25/2020) (cit. on p. 16).
- [15] Caliban — caliban. **GraphQL Client | Caliban**. URL: <https://ghostdogpr.github.io/caliban/docs/client.html#code-generation> (visited on 12/25/2020) (cit. on p. 16).
- [16] Scala — scala-lang.org. **Overview | Reflection | Scala Documentation**. URL: <https://docs.scala-lang.org/overviews/reflection/overview.html#universes> (visited on 12/25/2020) (cit. on p. 24).

- [17] Eugene Burmako. “Scala macros: let our powers combine: on how rich syntax and static types work with metaprogramming”. eng. In: **Proceedings of the 4th Workshop on scala**. SCALA '13. ACM, 2013, pp. 1–10. ISBN: 9781450320641 (cit. on p. 24).
- [18] Scala — scala-lang.org. **Scala Reflection Library 2.13.5 - scala.reflect.api.Trees**. URL: <https://www.scala-lang.org/api/current/scala-reflect/scala/reflect/api/Trees.html> (visited on 12/25/2020) (cit. on p. 25).
- [19] Scala — scala-lang.org. **Symbols, Trees, and Types | Reflection | Scala Documentation**. URL: <https://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html> (visited on 12/25/2020) (cit. on p. 26).
- [20] Scala — scala-lang.org. **Pattern Matching | Scala 2.13**. URL: <https://www.scala-lang.org/files/archive/spec/2.13/08-pattern-matching.html> (visited on 12/25/2020) (cit. on p. 30).
- [21] Scala — scala-lang.org. **Pattern Matching | Tour of Scala | Scala Documentation**. URL: <https://docs.scala-lang.org/tour/pattern-matching.html> (visited on 12/25/2020) (cit. on p. 30).
- [22] Scala — scala-lang.org. **Introduction | Quasiquotes | Scala Documentation**. URL: <https://docs.scala-lang.org/overviews/quasiquotes/intro.html#interpolators> (visited on 12/25/2020) (cit. on pp. 31, 32).
- [23] Denys Shabalin, Eugene Burmako, and Martin Odersky. **Quasiquotes for scala**. Tech. rep. 2013 (cit. on pp. 31, 32).
- [24] Scala — scala-lang.org. **Def Macros | Macros | Scala Documentation**. URL: <https://docs.scala-lang.org/overviews/macros/overview.html> (visited on 12/25/2020) (cit. on p. 32).
- [25] sbt — scala-sbt.org. **Macro Annotations | Macros | Scala Documentation**. URL: <https://docs.scala-lang.org/overviews/macros/annotations.html> (visited on 12/25/2020) (cit. on pp. 36, 37).
- [26] Bartosz Bąbol — bbartosz.com. **Scala Macros: Part II- Macro Annotations - Bartosz Bąbol**. URL: <http://www.bbartosz.com/blog/2016/01/24/scala-macros-part-2-macro-annotations/> (visited on 12/25/2020) (cit. on p. 38).
- [27] NIST — nist.gov. **Application Programming Interface - Glossary | CSRC**. URL: https://csrc.nist.gov/glossary/term/Application_Programming_Interface (visited on 02/25/2021) (cit. on p. 39).
- [28] Arnaud Lauret. **The design of web APIs**. eng. 1st edition. Shelter Island, NY: Manning Publications Co., 2019. ISBN: 1-61729-510-8 (cit. on p. 39).
- [29] Swagger — swagger.io. **What is API Documentation? | Swagger Blog**. URL: <https://swagger.io/blog/api-documentation/what-is-api-documentation-and-why-it-matters/> (visited on 03/08/2021) (cit. on p. 45).
- [30] Roy T Fielding. **Architectural styles and the design of network-based software architectures**. Vol. 7. University of California, Irvine Irvine, 2000 (cit. on pp. 48–50).
- [31] Roy T Fielding and Richard N Taylor. “Principled design of the modern web architecture”. In: *ACM Transactions on Internet Technology (TOIT)* 2.2 (2002), pp. 115–150 (cit. on p. 49).
- [32] Red Hat Inc. — redhat.com. **What is an API?** URL: <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces> (visited on 02/25/2021) (cit. on pp. 49, 50).
- [33] Y Combinator — ycombinator.com. **What RESTful actually means | Hacker News**. URL: <https://news.ycombinator.com/item?id=10765148> (visited on 02/25/2021) (cit. on p. 50).
- [34] Y Combinator — ycombinator.com. **The main problem with REST is Roy Fielding’s communication style. Here’s a quot... | Hacker News**. URL: <https://news.ycombinator.com/item?id=17565614> (visited on 02/25/2021) (cit. on p. 50).

- [35] Roy T Fielding. “REST APIs must be hypertext-driven”. In: *Untangled musings of Roy T. Fielding* (2008), p. 24 (cit. on p. 50).
- [36] Leonard Richardson — crummy.com. **JWTUMOIM: Act 3**. URL: <https://www.crummy.com/writing/speaking/2008-QCon/act3.html> (visited on 02/25/2021) (cit. on p. 50).
- [37] SmartBear — swagger.io. **About Swagger Specification | Documentation | Swagger**. URL: <https://swagger.io/docs/specification/about/> (visited on 12/25/2020) (cit. on p. 51).
- [38] Thomas Eizinger. “API design in distributed systems: a comparison between GraphQL and REST”. MA thesis. 2017 (cit. on p. 51).
- [39] Maximilian Vogel, Sebastian Weber, and Christian Zirpins. “Experiences on Migrating RESTful Web Services to GraphQL”. eng. In: **Service-Oriented Computing – ICSOC 2017 Workshops**. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 283–295. ISBN: 9783319917634 (cit. on p. 51).
- [40] The GraphQL Foundation — graphql.org. **Serving over HTTP | GraphQL**. URL: <https://graphql.org/learn/serving-over-http/#http-methods-headers-and-body> (visited on 12/25/2020) (cit. on p. 53).
- [41] JSON:API — jsonapi.org. **JSON:API — Latest Specification (v1.0)**. URL: <https://jsonapi.org/format/#fetching-sparse-fieldsets> (visited on 12/25/2020) (cit. on p. 53).
- [42] Scala — scala-lang.org. **SID-1 Named and Default Arguments | Scala Documentation**. URL: <https://docs.scala-lang.org/sips/named-and-default-arguments.html> (visited on 12/25/2020) (cit. on pp. 59, 60).
- [43] The GraphQL Foundation — graphql.org. **GraphQL Code Libraries, Tools and Services**. URL: <https://graphql.org/code/> (visited on 12/25/2020) (cit. on p. 74).
- [44] Can I Use — caniuse.com. **Can I use... Support tables for HTML5, CSS3, etc.** URL: <https://caniuse.com/wasm> (visited on 12/25/2020) (cit. on p. 74).

VITA

Name Ashesh Kumar Singh

Links

Website <https://ashessin.com>

LinkedIn <https://linkedin.com/in/ashessin>

Education

Bachelor of Engineering — Information Technology, July 2012 — May 2016
University of Mumbai

Work Expirence

Systems Engineer, August 2016 — July 2019
Tata Consultancy Services Ltd.
Worked on CISCO's eCommerce project (CCW) in their agile technical operations team.

Intern, June 2015 — August 2015
Indian Institute of Technology, Bombay
Worked on e-Yantra project at Embedded Real-Time Systems (ERTS) Lab.