UNIVERSIDAD POLITÉCNICA
DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE -
EUROPEAN MASTER IN SOFTWARE ENGINEERING

Moogle: A Type-based API Search Engine for Elm

Master Thesis

Junpeng Ouyang

Madrid, June 2019

This thesis is submitted to the ETSI Informáticos at Universidad Politécnica de Madrid in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering.

Master Thesis
Master Universitario en Ingeniería del Software – European Master in Software Engineering
Thesis Title: Moogle: A Type-based API Search Engine for Elm
Thesis no: EMSE-2019-10
June 2019

Author: Junpeng Ouyang
EMSE student
Universidad Politécnica de Madrid

Supervisor:

Julio Mariño Carballo
Associate Professor
Universidad Politécnica de Madrid

Languages and Systems and Software Engineering
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Co-supervisor:

Ángel Herranz Nieva
Assistant Professor
Universidad Politécnica de Madrid

Languages and Systems and Software Engineering
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

ETSI Informáticos
Universidad Politécnica de Madrid
Campus de Montegancedo, s/n
28660 Boadilla del Monte (Madrid)
Spain

Acknowledgment

Firstly, I would like to thank my supervisors, Professor Julio Mariño Carballo and Professor Ángel Herranz Nieva, who had helped me on completing my thesis. Thanks to your selfless guide and careful mentoring, which makes this work goes smoothly and inspired me even I was rather novice to functional programming.

I also would like to thank my domestic supervisor, Professor Liu Yan, who had supported me to seize this opportunity of study abroad for double master degree on Software Engineering. Without her previous education on related field and routine discussions during this year, I would never have an extended view on the work of analyzing programming languages.

I am also grateful to the coordinators of this exchange program, which includes Professor Xavier Ferré Grau, Ms. Li Meihui, Ms. Yang Xiaowen and Ms. Ge Lei, who had been taken care of my study progress and helped me during the entire academic year.

And I would like to give my special acknowledgment to Chinese Scholarship Council, who had sponsored me during the entire program. Their financial support has helped me focused on my studies.

Finally, but not the last, I would like to say thank you to my parents and family. Your love supports me to go through every rough time even we cannot reunion for almost the entire year. I cannot say anything but only appreciation to all your devotion.

Abstract

Searching for code is, in general, a hard problem, as text is not commonly representative of what a program does. In this thesis, we present Moogle, a type-based API search engine for Elm, a functional programming language designed for web development. In Moogle, search queries are based on names or type signatures, whose adequate information can be leveraged for matching APIs from third-party libraries so that developers are able to discover and reuse existing functions without reconstructing a similar one.

The key concept behind Moogle is the unification algorithm, where generic type signatures can be matched to concrete ones and vice versa. In order to optimize the performance for applying unification matches, Moogle stores the information of signatures in Neo4j, a graph DBMS that provides graph-based queries in its own query DSL, Cypher. Moogle also has its implementation on a parser to convert the signature of types into abstract syntax tree data models, and search string from users into Cypher queries. The signature of APIs in Neo4j are first selected by generated queries then finally filtered by unification module, the results of which can be categorized into two sets: the candidate set and the full match set, where all items will be ranked and shown in the user-interface.

After development phase, we built and deployed our software with an online version based on Docker image.[1] According to our test, Moogle outperforms its congeneric work, Elm-search, on many aspects including search range, allowed patterns and ranking metrics with an acceptable trade-off on performance.

[1]The URL is https://moogle.joshoy.xyz

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this chapter, we introduce the context of the thesis, give depiction of motivation and delimit the problems to be solved by our work. After the general overview the structure of thesis documentation is presented.

1.1 Context and Background

In this section, we will briefly introduce some background knowledge and concepts before summarizing the problems to be solved. Subsection 1.1.1 expounds Elm, a functional programming language with its lingual features, which boosts the maintainability of software. Then Subsection 1.1.2 introduces some basic concepts on Moogle's primary role, API search engine.

1.1.1 Elm: a functional programming language for web

Nowadays, web development has leaped into a new stage since the quick involvement of new standards and runtime environments. Unlike 20 years ago, web pages are no longer statically rendered as before in many websites. Instead, as more interactive elements are embedded into the front-end side, state management starts to become a non-trivial problem since it regularly brings troubles for the developers. TypeScript and Dart indeed effectively solve the untyping problems in JavaScript by bringing strong typing. Nevertheless, they cannot solve two underlying problems which frequently haunted maintainability and robustness of the software - type safety and side effect. Hence, *functional programming* (abbrev. FP) languages like Elm and their ideologies are introduced into web development.

Elm is a functional programming language influenced by Haskell, Standard ML, OCaml and F#. It was designed for GUI programming and deeply affected by FRP. Elm is fully focused on building webapps with both robustness and good performance. Its strongly-typed compiling system guarantees the maintainability of software, and no side effect ensures that there will be no runtime exception during the usage. To achieve this, Elm adopts a strong, static type system like Haskell but removing advanced features like Type-class. In Elm, there is a centralized state which can be only be updated by `update` function. Thus, in most cases, developers can focus on how to create functions that do the mappings.

In the world of imperative programming, two variables a and b may be assigned with the same expression $a \leftarrow f(x)$ and $b \leftarrow f(x)$ but there is no guarantee that the equation $a = b$ would always be satisfied. The reason is that we cannot know whether the procedure of function f refers to any global variable that may be subject to change [App04].

In the world of FP, however, the core idea is that functions, or more precisely, pure functions, are "first-class" citizens. In the case above, the expression $a \leftarrow f(x)$ or $b \leftarrow f(x)$ would never occur, for any expressions in FP are always *equations*. A equation $f(x) = y$ will always be satisfied once it was fulfilled regardless of any time factor. And the output of a function only depends on its input. This is also known as *stateless*.

In FP, Everything is built *bottom-up* through limited primitive types and the combination of built-in functions. This ideology is on the opposite side of Object-Oriented Programming(abbrev. OOP), which takes everything as objects and modules, and adopts a top-to-bottom approach by decomposition. Both paradigms provide the capacity of abstraction. OOP focuses on the data structure, while FP concentrate on algorithms.

Nevertheless, building a stateless software project is akin to constructing a Utopia. In practical cases, especially in the web, it is impossible to have a stateless runtime for a program. Although it is impossible to exterminate side effect, controlling the influence of side effect at its minimum level and write more pure functions is still a feasible solution. In order to achieve this goal, many approaches are proposed and brought into actions.

Functional reactive programming (abbrev. FRP) is a concept that incorporates FP with reactive programming (abbrev. RP). In RP, time-varying variables are abstracted as *observables*. The advantage of RP is that events and values are regarded as a ordered set, or a stream. The most benefiting part of FRP are that: (1) Shared mutable states are controlled at a minimum complexity. (2) Event handlers are easy to be written in chained, combined, pure functions. These traits enhance the testability and scalability of our software project.

Influenced by Elm, a widely-known solution to state management in JavaScript is to keep a centralized state. Redux and Vuex integrate the concepts of dispatcher, view, single store and unidirectional data flow from Flux [flu19] (an application architecture for building user interfaces) and become one of the most popular state management libraries so far.

Yet, a most troubling problem is that when the programmer tries to build up a function in Elm, he or she has to find a existing code and reuse those code to build up the source code.

1.1.2 API search engine: a useful tool for code reuse

To increase the maintainability of a software project, Developers regularly invoke code segments or ready-made APIs from libraries and packages when coding on specific functions. This behavior is also known as *code reuse*. However, in practical cases, the major obstacle for developers is how to retrieve the names of APIs or whether there exists a ready-made API with the functionality that they demanded. In order to solve this problem, the developers may look to API search engines to retrieve specific APIs that might be useful.

An API search engine, literally speaking, is a software that allows the users to search for ready-made APIs from libraries and packages. Different from *code* search engines,

API search engines do not concern about collecting source code implementation of libraries. Instead, they store and index the functionality, documentation, and signature of each API, which are more readily comprehensible for the developers to understand whether the API is useful or not.

It also needs to be noticed that there are multiple categories of API search engines because of various aims, and each of them is built then optimized based on specific principles and usage scenarios. In this thesis, we will focus on searching APIs by type constraints provided by users.

Now that we have introduced some background knowledge on Elm and API search engine, in the next section, we will continue elaborating on the problems to be solved. On the other hand, in Chapter 2, we will study on several type-based API search engines that heavily influenced the design and functionality of Moogle.

1.2 Problem Description

As is stated in the previous section, reusing the existing modules have empirically identified to be effective. However, building a complex module from scratch is, in general, a hard problem. One typical case is that when given concrete input and output types, the developers are in demand of combining a series of existing APIs from libraries into a convoluted function to implement the required module.

FP requires complex combination of functions which may involve a mass of type transformation. For instance, a chained currying function `f :: A -> B -> C -> D` in Haskell binds a single parameter `x` with type `A` and returns function `(f x) :: B -> (C -> D)`, where `A`, `B`, `C` and `D` are concrete types in this case. A potential dilemma for the developer in the case above is that he or she may only know that the required input type is `A` and the final output type is `D` but has no idea of how to do the combinations with existing APIs.

Intuitively, users may try to describe their requirements (functionalities of API) in a semantic approach. Consider another case as an example: a developer is looking for a function that returns the element number of a `List` value in Elm. However, he or she is ignorant about the name of that API. Hence, the user may look to an engine and search by some semantic clues (for instance, keywords) based on their knowledge in natural language. However, the inputs can be variant since a keyword may have synonyms, which makes it hard to guarantee that the engine would do a hit. In the case above, the synonymous terms could be (but not limited to):

- length
- size
- len
- sizeof
- lengthOf

It is obvious to observe that in this case, semantic information based on natural language could be inaccurate.

On the other hand, if the user searches with type information instead of natural language ones, it is easy to conclude that the API he or she wants should accept a `List` of any type as its input, and returns an integer (`Int`) as its output. Therefore, the signature must be compatible with type: `List a -> Int`. Then instead of using any name keyword, the user searches for an API which conforms to that type, and the engine quickly retrieves the full signature of demanded API, which should be `length : List a -> Int`. Note that in this case, instead of trying different keywords for multiple times, the user obtains required API in a single search operation, which is more convenient and efficient than the former approach.

Moreover, if both name and type information can be provided, it shall be more comfortable for the search engine to retrieve desired APIs.

Based on the usage scenarios above, we define the problems by providing several examples and use cases:

1. **Search by function name**: The developer knows the name of the API and tries to search the function by name directly. However we cannot guarantee that the name is precisely inputted thus fuzzy search is required in our product.

   USE CASE 1: A developer wants to know the detailed usage of function `Array.initialize`. Then he or she enters `Array.initialize` or `initialize` into the search input. The search engine should retrieve a list of results including the function named "initialize" with the package name "Array" including its type (`Int -> (Int -> a) -> Array a`) and the outline documentation of the function.

   USE CASE 2: A developer wants to search for the usage information of function `foldr`. But he or she fails to remember the very precise name of the API and then enters "fold" inside the input. The search engine should be able to retrieve any results that similar to the function name, including any existing functions or values that named `foldr`.

2. **Search by type**: In a strongly typed FP language, type expressions are syntactic terms annotating the value of types, and a type represents a set of values [HFP92]. Searching by type allows developers filter and retrieve the functions they need quickly. The search engine should provide a retrieval function that allows the developers to search APIs by any legal type expression.

   USE CASE 3: A developer wants to find a function that parses a `Int` value to a `Float` number. Then he or she enters the type signature `Int -> Float` inside the search input. And the engine retrieves all the possible packages and returns a list of functions that matches the result with their documentations.

The items above will be the initiate problems that Moogle has to solve. In Section 2.3 the use cases will be extended according to our observation and study from related works, and further requirements will be added.

1.3 Objectives

According to the context and problems we faced when using Elm, we define the aims of our work as the followings:

1. Look into related works and find referable features that may required by Elm developers. Study on and implement algorithms and metrics which support name-based and type-based searching.

2. Design the architecture as the solution of a type-based search engine, Moogle, for searching Elm APIs.

3. Design and implement a search-oriented DSL called MoogleQL, which can be used for parsing type signatures and search queries.

4. Crawl and extract package informations from Elm packages and set up data models to store them.

5. And finally, aggregate all the modules and implement the final product, Moogle.

1.4 Structure of Documentation

In this section we list the structure of entire thesis documentation, which is composed of another 5 chapters.

In Chapter 2 we study on related works, which are principally open-source type-based code search engines. We will introduce Hoogle, a referable type-based code search engine for Haskell, and Elm-search, a existing yet flawed Elm code search.

Chapter 3 focuses on algorithm and metrics to several principal problems that Moogle has to solve, where some theories, concepts and algorithms for searching APIs and ranking the results will be introduced in detail.

Chapter 4 depicts the implementation details, which involves the query DSL design, the architecture and database structure that we apply.

In Chapter 5 test and benchmark results will be presented in tables and diagrams. We will also display the functionality of the Moogle prototype with some use cases.

Then finally in Chapter 6 we draw the conclusion of our work and summarize the experience and future work.

Chapter 2

Related works

Before we commence our work, we look into some inspiring or referable work, all of which are open-source projects that be widely used by developers. In Section 2.1 we introduce Hoogle, a code search engine based on type signatures, which provides several innovative features that has been absorbed by work Section 2.2 introduces Elm-search, an existing API search engine which is similar to Hoogle but defective. And finally in Section 2.3 we extend the problems that we described in the previous chapter, which delimits our study more precisely.

2.1 Hoogle

In Haskell, one of the widely used tools for searching APIs is Hoogle [Mit19], which is an open-source API search engine that allows searching Haskell libraries by either function name or by similar type signature. The greatest advantage of using Hoogle is that the developers do not have to know the name or exact signature of the function at same time. An example is given that the programmer knows that the part of/the full signature of the function he or she need is `(a -> b) -> [a] -> [b]`. Then he or she can type the expression above into the search input of Hoogle, and all the matching results will be ranked and shown in the browser or in the terminal. Figure 2.1.1 demonstrates usage of Hoogle in the browser.

2.1.1 Elm vs Haskell on function signatures

Hoogle provides a referable solution for API searching in FP languages. But the grammar and syntax of Elm is quiet different from Haskell. We collect these differences, especially on the typings aspect, and summarize them in a list of items. In Elm:

- Types are declared with `:` instead of `::`
- There is no type-classes and higher-kinded types
- The keyword `type` is corresponding to keyword `data` in Haskell
- It has a unique primitive type called `Record` which is also extensible

The differences above should be noted while designing our DSL for Elm API searching.

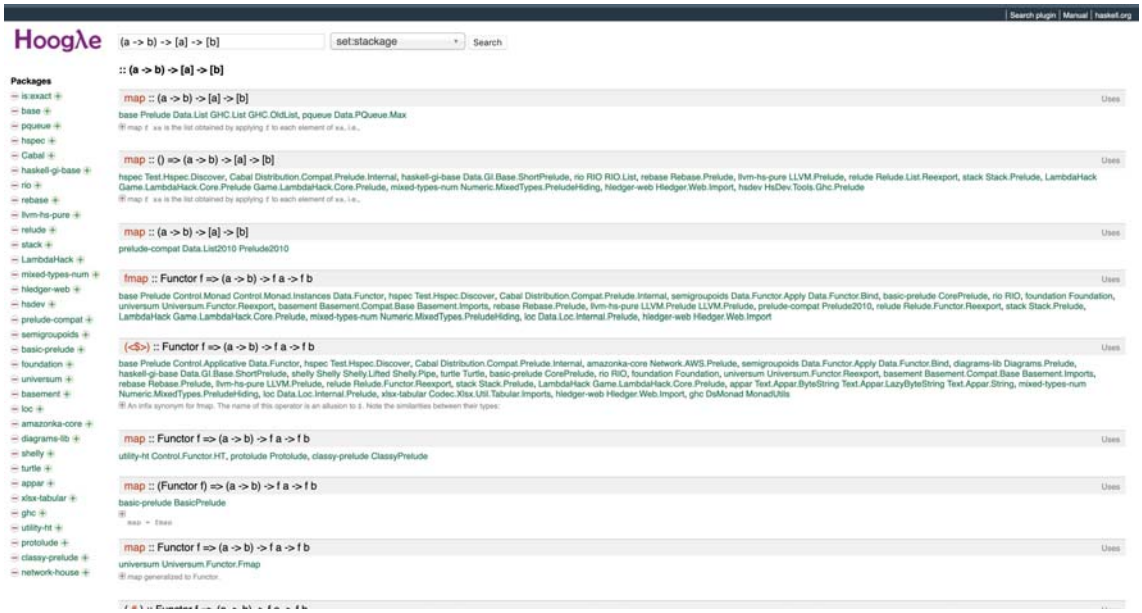

Figure 2.1.1: Hoogle API search engine usage example

2.1.2 Remarkable features in Hoogle

Hoogle provides a powerful search engine for Haskell functions and APIs since it supports type-based code search. We list some of features which are worth considering to be implemented in our work as below:

1. **Matching with type variables**: In Haskell and Elm, a type variable (similar to the concept of generic type in C++/Java) may exist in API functions. A type variable in type signature represents a type placeholder where you can put any concrete type in the corresponding place as long as all of them are matching the same type variable. We will go into further details on this topic in Chapter 3, where the core mechanism behind will be explained.

2. **Reordered parameters**: In some cases, developers may not know precisely the order of arguments for a high-order function. For instance, a developer wants to take the first `x` (where `x` is of `Int`) elements from a list. In Haskell, the build-in function that fulfills his or her need is `take :: Int -> [a] -> [a]`. However, the developer mistakenly enters the type expression as `[a] -> Int -> [a]`, where the order of first two arguments is reversed. In this case, Hoogle returns a list of APIs that unify not only the type of `[a] -> Int -> [a]` but also the pattern `Int -> [a] -> [a]`, as Figure 2.1.2 shows.

3. **Search by type constructors**: In Haskell, custom types we created with `data` keyword (In Elm, the corresponding one is `type`) are called union types, which is analogous to a disjoint union set of tagged elements. We usually name these tags as term "type constructors". A common problem developers may face is that they may not understand which type does a constructor build. For instance, `Just 5` represents `Maybe Int` type in Elm and Haskell. But if there is no explicit expression in the code, developer may be confused at which type it belongs to. In

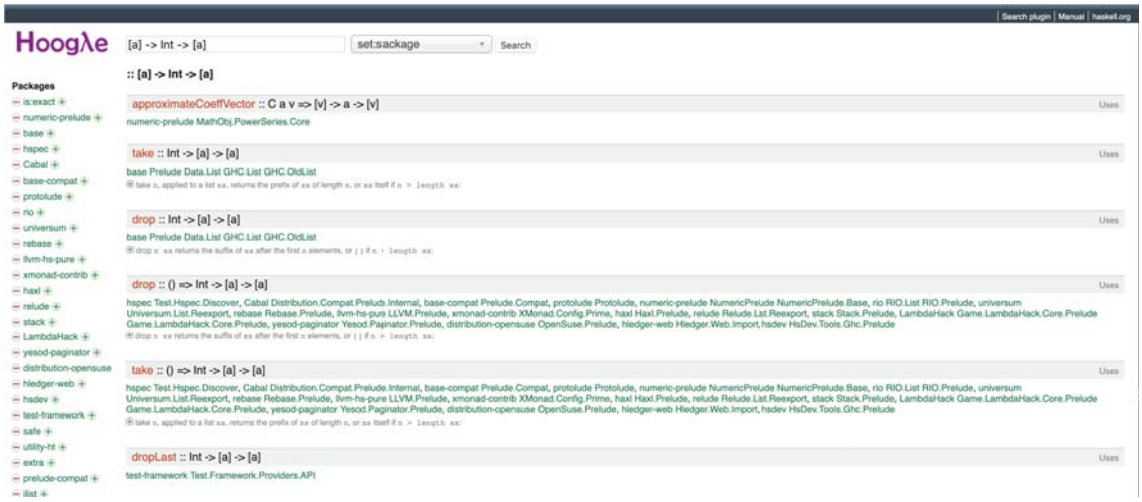

Figure 2.1.2: Reordered parameters in Hoogle

Hoogle, the developer may search the type constructor `Just` directly. The engine will provide a list of results where `Just` is given as a function with signature `Just :: a -> Maybe a`. Then the developer will realize that `Maybe Int` is the type of value `Just 5`.

4. **Search by package name**: Searching by package name allows developers to browse all the API members of a specific package, which significantly increases the usability since unwanted results can be filtered out quickly.

2.1.3 Limitation in Hoogle

Hoogle is powerful on both name-based and type-based search scenarios. However, we still noticed a limitation case. In Hoogle, it is always possible to map a type variable in query to a concrete type in results. However, it is not working all the way when we tries to map a concrete type in query to a generalized one.

For instance, a developer wants to transform a list of integers to strings, but he or she does not remember which function should be applied, thus the user types "(Int -> String) -> [String] -> [String]" as the query. Hoogle, however, returns only 3 results as Figure 2.1.3 shows. Nevertheless, the real API that our user need is function "map" with signature `map :: (a -> b) -> [a] -> [b]`.

2.2 Elm-search: An open-source project for Elm API search

Elm-search [Coc19] is an open-source search engine created by Jonas Coch in 2016, which is used for searching all exposed APIs in Elm packages.The engine is written in Elm and served as a SPA (Single Page Application) on GitHub static page. In Elm-search developers can search for APIs in the similar way as they do in Hoogle, since the typing system in Elm is quiet similar to Haskell. A few examples can be found in the homepage of Elm-search, which provides its essential usage. Figure 2.2.1 gives an example of searching functions with type `a -> List a`.

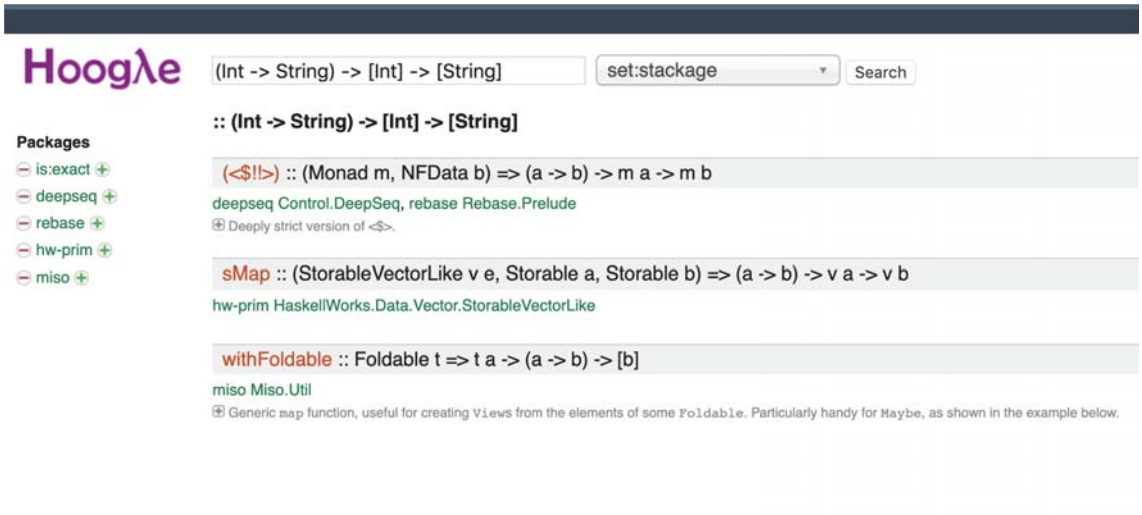

Figure 2.1.3: Limitation case of Hoogle

Elm-search provides a useful approach for Elm developers to search APIs by type easily, as Elm-search is trying to build up a Hoogle for Elm. From the usages it provides, we can easily see that Elm-search has implemented some features that we had described in. Nonetheless, there still exists many problems that worth discussions. In the next section and its subsections we will mention some issues around it and find out why it does not conform to our requirements.

2.2.1 Limitations in Elm-search

Elm-search provides a type-based solution for Elm API searching but sometimes the results that it found are confusing. In the next subsections we summarize and categorize all these exceptional, or unexpected behaviors into varies of categories, which are indicated in the following subsections.

1. **Incomplete list of results**: A common issue in Elm-search is that when involving a type variable it performs unexpected behavior and returns a list of incomplete results, which may cause the absence of the needed API.

   One example is when searching for an extensive type `a -> b`, Elm-search returns only one single result, function `empty : k -> Maybe v`, as Figure 2.2.2 shows. While in Hoogle, `a -> b` matches all the possible APIs that refers to a function. This result cannot be accepted as an expected behavior since all the the pattern above should match all the available functions in Elm-Package.

2. **Unwanted types**: Also, Elm-search performs unexpected behavior when doing the search that involves a type constructor. For instance, when doing the search `(a -> b -> b) -> b -> List a -> b`, the top 6 results that Elm-search returns is shown in Table 2.1, from which we see that only the first and the second result can fully match the given query, while the 3rd, the 4th and the 5th ones are changing the given type constructor `List` into other constructors. Meanwhile,

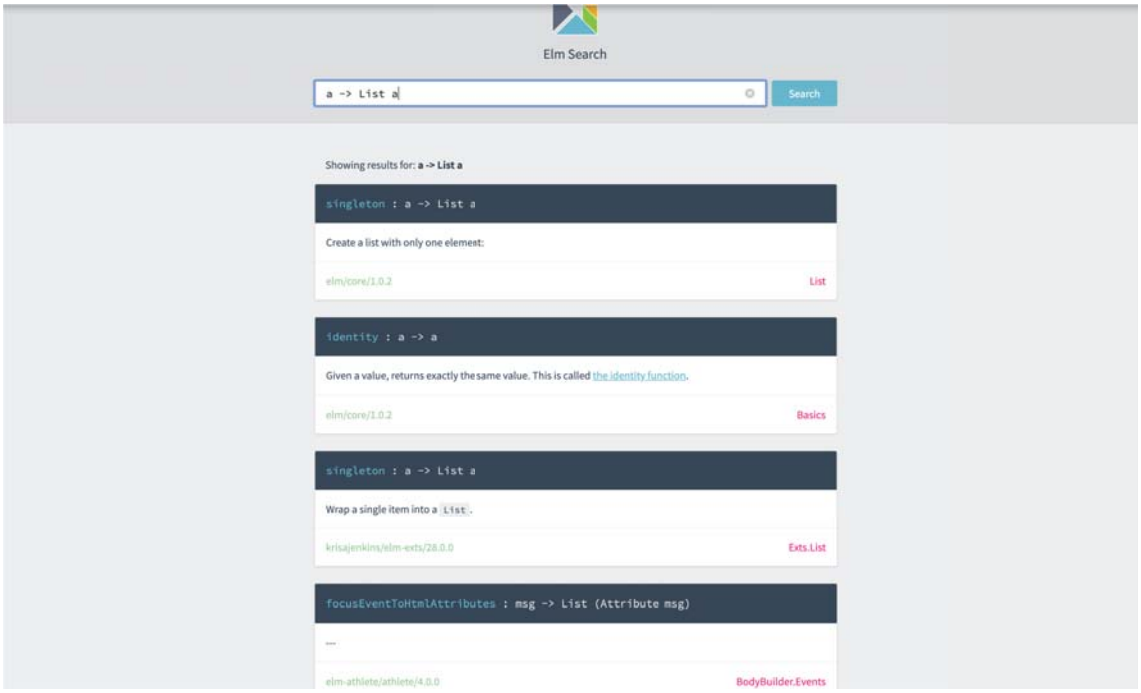

Figure 2.2.1: Elm-search usage example

the 6th result changes the type of the first parameter from `(a -> b -> b)` to `(Int -> a -> b -> b)`.

Table 2.1: Top 6 results of searching `(a -> b -> b) -> b -> List a -> b`

No	Name	Type signature
1	foldl	`(a -> b -> b) -> b -> List a -> b`
2	foldr	`(a -> b -> b) -> b -> List a -> b`
3	foldl	`(a -> b -> b) -> b -> Array a -> b`
4	foldl	`(a -> b -> b) -> b -> Set a -> b`
5	foldl	`(a -> b -> b) -> b -> UndoList a -> b`
6	indexedFoldl	`(Int -> a -> b -> b) -> b -> List a -> b`

Although allowing fuzzy search is not a bad idea since types may not be specific, but providing unwanted types is never efficient and sometimes misguiding.

3. **Lacking support for Elm records**: Record is a unique type in Elm and a feature differentiating Elm from Haskell. A record corresponds to a hash map data structure similar to objects in JavaScript but requires typings for its members. A common convention is to use type alias to shorten the notation of a record type. For instance, we use `type alias Person = {name : String, age : Int}` to notate a Person record data structure instead of using `{name : String, age : Int}` every time.

   Another Record-related feature that deserves to be mentioned is that Records are *extensible*. Although there is no typeclass, Elm does provide another approach to notate a desiring interface for complex data structures by using extensible records.

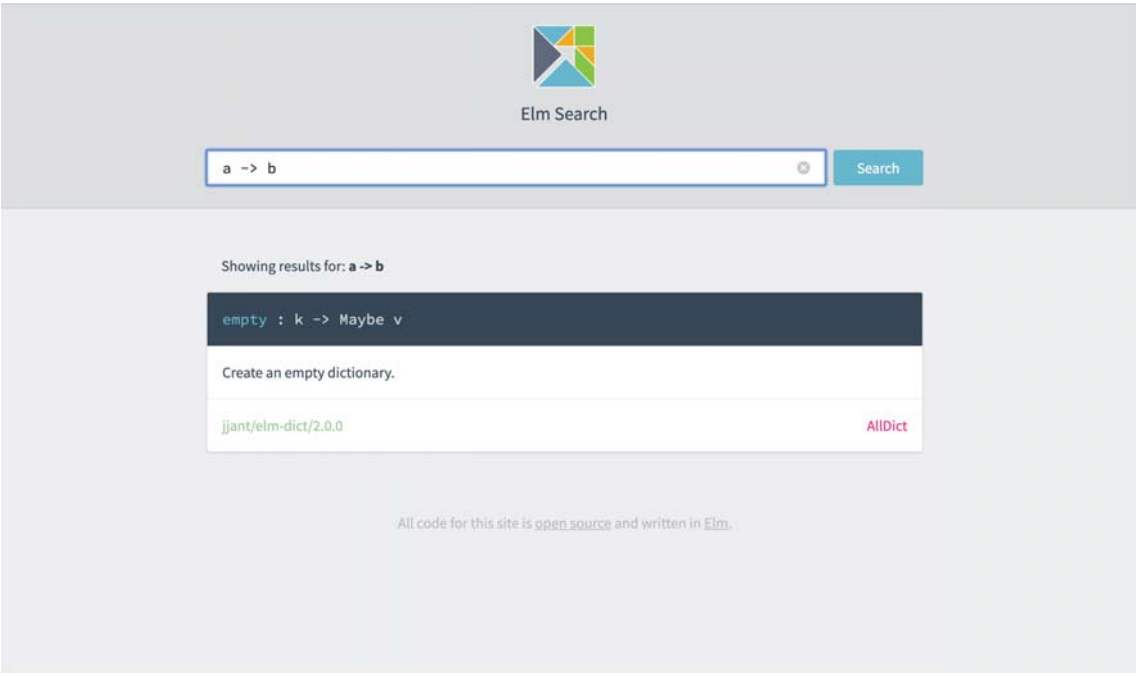

Figure 2.2.2: Elm-search limitation case: searching for type `a -> b`

Extensible records are defined to be having at least one or more assured field with specific types, which grants usages similar to interfaces for developers. An example is that we can define `type alias Named a = {a | name : String}` and use the type in function `getName : Named a -> String`. Then we can fill in any records, for instance, a `Person` record above, to the parameter field.

Based on above reasons, record is a powerful feature in Elm. Yet, there is not any search pattern available for supporting Elm record types in Elm-search so far.

4. **Incomplete search patterns**: In Elm and Haskell, name terms and type variables are always written in a lower case word, while concrete types are written in capitalized ones. Thus, searching by names and types is feasible by defining the pattern of `name : Type`.

   Elm-search does not support searching an API by providing both name and type at the same time, while it is a common case in Hoogle to search by typing `name :: Type`. On the other hand, Elm search does not support any queries in substring pattern or searches by module or package name.

2.2.2 Why Elm-search flaws

As we mentioned several problems above, Elm-search may not perform expected behaviors and returns precise results according to our given use cases. In order to find out the root causes of the problem, we look into the source code of the project itself then inspect the code and its mechanisms. In the next subsections we will illustrate the background mechanism of Elm-search and illustrate why the issues happen.

Elm search uses *distance*, a metric with penalty factor, to determine whether a function should be added into the list of results and what its rank should be. The smaller a distance value is, the higher rank of API will be. A distance greater than a threshold will be filtered out and not listed in the results. This metric defines several penalty factors as constants at different weights as Figure 2.2.3 shows.

```
1     noPenalty : Float
2     noPenalty = 0
3
4     lowPenalty : Float
5     lowPenalty = 0.25
6
7     mediumPenalty : Float
8     mediumPenalty = 0.5
9
10    highPenalty : Float
11    highPenalty = 0.75
12
13    maxPenalty : Float
14    maxPenalty = 1
```

Figure 2.2.3: Penalty factors definition

After we checked all the functions that are externally exposed in this module, only two functions `simple`, `tipe` and one constants `lowPenalty` are used externally.

The procedure of function `simple` is defined as pseudo code (Algorithm 1) shows.

Algorithm 1: function `simple` in Distance.elm

Input: $ExtractFunc\ (type: Chunk \rightarrow String)$, $queryStr\ (type: String)$, $chunk\ (type: Chunk)$
Constants: $NoPenalty = 0, MediumPenalty = 0.5, MaxPenalty = 1$
Output: $distanceMeasure\ (type: Float)$

$chunkStr \leftarrow ExtractFunc(chunk)$;
if $chunkStr = queryStr$ **then**
  $distanceMeasure \leftarrow NoPenalty$;
else
  if *queryStr is a substring of chunkStr* **then**
    $distanceMeasure \leftarrow MediumPenalty \cdot (1 - \frac{length\ of\ queryStr}{length\ of\ chunkStr})$;
  else
    $distanceMeasure \leftarrow MaxPenalty$;
  end
end

Here we see that the basic functionality of function `simple` is to compare *chunkStr*

and *queryStr*, calculating a distance measurement and returns it. However, function `simple` is only used when comparing names of constants, users, packages and modules, as Figure 2.2.4 indicates. In Elm-search, a query is compared to each "chunk" one-by-one, and the types are calculated according to an accumulation of the name and type distances.

```
distanceByQuery : Query -> List ( Float, Chunk ) -> List ( Float, Chunk )
distanceByQuery query chunks =
  let distance =
      case query of
        Name name ->
          Distance.simple (.context >> .name) name
        Type tipe ->
          Distance.tipe tipe
        User name ->
          Distance.simple (.context >> .userName) name
        Package name ->
          Distance.simple (.context >> .packageName) name
        Module name ->
          Distance.simple (.context >> .moduleName) name
  in
  List.map (\( d, c ) -> ( d + distance c, c )) chunks
```

Figure 2.2.4: Function distanceByQuery

When comparing two types, Elm-search applies a function `normalize` on both signatures, which transforms all the type variables into normalized ones. For instance, `Int -> x -> y` will be rewrite as `Int -> a -> b`. Then Elm search compares each argument and return type by name strings, which is similar to the approach for names that we mentioned above.

Although the mechanism above seems reasonable, but in actual implementation, there are several fatal flaws that made the results of Elm-search less than satisfactory.

- When comparing two types, Elm-search compares the name strings of two types as the distance measurement. A problem is that the returning distances are always *fixed* values, where the rankings between two APIs may not accurate.
- Another problem is that, when applying the comparison, Elm-search does not take any type variable as a substitutable placeholder. It is only effective when comparing two functions whose function signatures have one-to-one correspondence. For instance, `x -> x` can match `comparable -> comparable` since both functions have normalized format as `a -> a`. However, when comparing `x -> y` with `a -> a`, the matcher broke down.
- On the other hand, Elm-search does not support tuples and records, which is caused by an incomplete work on implementation.

2.3 Supplemental features from related works

In this chapter, we looked into Hoogle and Elm-search, both of which are type-based search engines.

Hoogle and Elm-search provides a series of referable functionalities and features, which includes substitutable variables, reordered matchings and type constructor variables. These features could be effective in more use cases which supplement the problems that we have mentioned in Section 1.2, which includes:

1. **Search by type constructor**: The developer encounters a type constructor but he or she does not know which type it belongs to. Then he or she may inputs the name of that type constructor in the search input, and the engine should returns the target type in the format of a function.

   USE CASE 4: The developer encounters a type constructor `Internal Url` in the source code and he or she wonders which type does this constructor belongs to. Then he or she types `Internal` into Moogle, and the engine returns a list of results, where the target type `UrlRequest` should be included in the format of a function `Internal : Url -> UrlRequest`.

   USE CASE 5: The type `Either a b` includes 2 type variables and `Left a`, `Right b` are its type constructors. Then the developer can search by the type constructor `Right`, the results should include a type constructor in the format of a function `Right: b -> Either a b`.

2. **Reordered matchings**: In some cases, developers may not know precisely the order of arguments for a high-order function. Thus, input parameter types may be inputed in a random sequence. The engine, in some extent, is required to provide error-tolerant capability on this kind of problem.

   USE CASE 6: The user wants to search for APIs that return the first/last n elements of a list then he or she inputs `List a -> Int -> List a`. Note that in this case, the user mistakenly puts `Int` into the second argument. Moogle is expected to return a list of results which includes the function "take" which actually fulfills the user's need.

3. **Abstraction for concrete types**: The user, in some cases, may not realized that the needed APIs are designed for "general cases". Moogle should be able to recognize the potential generic APIs that matches the pattern.

   USE CASE 7: The user wants to retrieve the API that gets the size of a list of Strings. So he or she inputs `List Int -> Int`. Moogle should be able to return the API `length` with signature `List a -> Int`

4. **Type constructor as variables**: Elm-search provides fuzzy results on type constructors by substituting `List` with `Array`, `Set`, etc. This approach, however, may confuse the user since these types are not matching(or precisely, unifiable). A better solution is, borrowing the concept of generic type constructor of Haskell.

   USE CASE 8: The user writes search query as `(a -> b -> b) -> b -> f a -> b`, where `f` refers to an identical type constructor. Moogle should be able to substitute `f` with concrete type constructors like `List`, `Array`, `Set`, etc.

5. **Package constraints**: the query can include package constraints, which limits the search range for packages.

   USE CASE 9: The user writes search query for "concat" but he or she only wants to search in a specific package with name "core". Thus the he or she enters "+core" as the prefix before the query.

6. **Module constraints**: The module name of a specific type in a signature can be specified.

   USE CASE 10: The user wants to know if there is any API that has type `Generator a`, while the module of `Generator` is "Random.Pcg.Extended". Hence the user writes `Random.Pcg.Extended.Generator a` to filter out unnecessary cases from other modules.

7. **Support for tuple and records**: Moogle should be able to support syntax for matching tuples and records, including the extensible ones.

   USE CASE 11: The user wants to know if there is an API which combines two items into a tuple with two elements (precisely speaking, a pair). Thus he or she search for type `a -> b -> (a, b)` and Moogle should returns the API: `pair : a -> b -> (a, b)`.

   USE CASE 12: The user searches for a record that contains two attributes: "text" and "font". Both of keys map to `String` type. But the user cannot recall the full signature of that API. Hence, he or she searches for `{ x | text: String, font: String }` and Moogle returns the aliased type `type alias PathSpec = { text : String, font : String, fontSize : Float }`.

Chapter 3

Theories and metrics for designing Moogle

In this chapter, we introduce some concepts and theories that supports the design of our search engine before embarking on the implementation in Chapter 4. Section 3.1 introduces our design for name-related searching metrics. Section 3.2 elaborates on type related theories and algorithms for solving type-based search cases that involves type variables and polymorphism. And in Section 3.3 explains some metrics that we designed for type signatures in Moogle.

3.1 Name searching algorithm and metric

Searching by name is a common and efficient approach when developers are looking up a specific API. As we collect all the type and module names from Elm packages, we should define a precise algorithm which filters and ranks all the searching results. In order to make our approach practical, we summarize all the potential cases that may occur during a search.

3.1.1 Search cases and patterns

1. **Search by value name**: In this case, the users search in values, aliases that present in Elm packages. The query string must be fully or partially match to the name string of the value or alias. The pattern of a query string should be a single word, which starts with a lowercase letter and only consist of characters and numbers. For instance, the user enters “map” as the search query, in order to retrieve any value or alias that fully or partially includes “map” in its name.

2. **Search by individual type or constructor**: In this case, the developers search for union types, type constructors and aliases that presents in Elm packages. The query string must be fully or partially match to the name string of the union type, type constructor or alias. The pattern of a query string should be a single word, which starts with a capitalized letter and only consist of letters and numbers.

3. **Search by record attributes**: Record is a unique type in Elm which corresponds to objects in JavaScript. A Elm record is consist of the key-value pairs listed are

surrounded with a left brace ({) and right brace (}) and separated by commas.

A special case is extensible records in Elm, where attributes are the subset of the searching record. For instance, the expression `{ s | a: A, b: B }` denotes the set of records s which meet the condition in Equation 3.1:

$$\begin{aligned} & s \in \{\, r \mid \text{“}a\text{”} \in K_r, \text{“}b\text{”} \in K_r \,\} \\ & T(r.\text{“}a\text{”}) = A \\ & T(r.\text{“}b\text{”}) = B \\ & r \in Records \\ & A, B \in Types \end{aligned} \tag{3.1}$$

where r are records; $r.k$ retrieves the value that k maps to in the record r ; K_r is the attribute key set of s; $T(x)$ represents the type signature of term x .

4. **Module names**: Module names are namespaces for all the types and values which helps avoiding naming conflict. The notation can be changed into pattern *ModulePath.name*, where module paths are separated with dot (.) characters, in order to restrict our search range into a specific module. This pattern can be applied to all the cases above.

3.1.2 Ranking metric: Levenshtein distance

After matching the names of searchable items we have to rank up the results. Although sorting with alphabetical order is always effective, the rank of an expected API could goes down due to this reckless metric. Therefore, a revised system of measurement for ranking results by name is required.

Levenshtein distance [Lev66] is a string metric to measure the difference between two strings. It is also one of the most widely used metrics for “Edit distance”. Levenshtein distance between string s_A to s_B refers to the minimum amount of *operations* to be taken for the transformation from s_A to s_B. Here an *operation* refers to one of the followings:

1. Insert a character: For instance, ac $\rightarrow$ a**b**c
2. Remove a character: For instance, abc**d** $\rightarrow$ abc
3. Substitute a character into another: For instance, a**b** $\rightarrow$ a**c**

Levenshtein distance can be denoted by Equation 3.2:

$$levdist(s_a, s_b) = dp(s_a, s_b, |s_a|, |s_b|) \tag{3.2}$$

Where $|s|$ denotes the length of string s and function dp is defined as Equation 3.3:

$$dp(a,b,i,j) = \begin{cases} max(i,j) & \text{if } min(i,j) = 0 \\ min \begin{cases} dp(a,b,i-1,j) \\ dp(a,b,i,j-1) \\ dp(a,b,i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise} \end{cases} \tag{3.3}$$

In Equation 3.3, $1_{(a_i \neq b_j)}$ denotes the indicator function, which equals to 1 when $a_i \neq b_j$ else 0.

From the equation we see that Levenshtein distance is defined recursively. We commonly use dynamic programming to optimize the time complexity, where optimum solutions are recorded in a 2-dimension matrix, as pseudo code in Algorithm 2 shows. In this algorithm the time complexity is $O(MN)$ and the space complexity is also $O(MN)$, where M and N are the length of two strings.

Algorithm 2: Levenshtein distance with dynamic programming

Input: $stringA$, $stringB$
Output: Levenshein distance between $stringA$ and $stringB$

```
1  m ← length(stringA);
2  n ← length(stringB);
3  S ← a m × n zero matrix;
4  for i ← 1 to m do
5  |  S_{i,1} ← i − 1;
6  end
7  for j ← 1 to n do
8  |  S_{1,n} ← j − 1;
9  end
10 for i ← 1 to m do
11 |  for j ← 1 to n do
12 |  |  if stringA_{i−1} = stringB_{j−1} then
13 |  |  |  S_{i,j} = S_{i−1,j−1};
14 |  |  else
15 |  |  |  S_{i,j} = min(S_{i−1,j−1}, S_{i−1,j}, S_{i,j−1}) + 1;
16 |  |  end
17 |  end
18 end
19 return S_{m,n};
```

3.2 Type searching theories and algorithms

The core feature and advantage of Moogle is that it allows users to carry out type searchings. Type searching refers to search by type signatures.

Type searching is similar to a type checking problem. In a monomorphic language, applying a type-checking is easy. Elm, however, has a polymorphic type system, where types can be generic with some parameters denoted as a symbol, which is named as term "type variable".

The compiler can infer and unify types with different signatures during the compilation or static analysis. For instance, a function with type `a -> a` can be "compatible to" (or to say, unify with) any function that accepts type T and returns the same type T. It signifies that `Int -> Int` can unify with `a -> a` when we apply the unifier { a = Int },

while `Int -> Float` would not be the solution since the unifier { a = Int, a = Float } is in conflict.

In short, searching by simply comparing strings of the type signatures would seldom returns a desired result. The following subsections introduce several theories that relate to type searching. And we will put these theories and algorithms into utilization during the implementation phase.

3.2.1 Type signature and polymorphism in Elm

In Elm, the signature of a type τ can briefly infer to an expression with the recursive syntax format (Equation 3.4):

$$\tau ::= apply \mid \alpha \mid \tau \rightarrow \tau \tag{3.4}$$

Where *apply* denotes an *instantiated type* and α implies a *type variable*. An instantiated type is consist of a type constructor with any number of arguments. Note that in this context, "type constructor" is not the equivalent concept of constructor in union types (For instance, `Just` of the union type `Maybe`). Here a "type constructor" is refers to the **parametric type constructor**, that is, the identity of union types, aliases, tuples, and records. For example, `Either a b` is a instantiated type with type constructor `Either` which takes two arguments `a` and `b`. And `Int` can be regarded as a instantiated type where a type constructor `Int` takes zero arguments. The expression `Just 1`, however, is a value.

In the cases above, both `Either a b` and `Int` are simplified representing format of instantiated types. Internally, the compiler introduces an meta type **Apply**, where instantiated types are actually taken as:

$$\begin{aligned} apply &::= Apply(tycon, tyList) \\ tycon &::= unionTyName \mid aliasTyName \mid tuple \mid record \\ tyList &::= apply :: tyList \mid \alpha :: tyList \mid [] \end{aligned} \tag{3.5}$$

It is easy to deduce from Equation 3.5 that the internal representation of `Either a b` is
Apply(Either, [Var a, Var b]). And the representation of `Int` is **Apply(Int, [])**. In a more generalized formal representation, **Apply** could even represents functions by introducing another meta type $\rightarrow$ and denotes all the functions `a -> b` as **Apply($\rightarrow$, [a, b])**.

Above all, we conclude that type signatures in Elm can be basically divided into three categories: instantiated types, type variables and functions.

The concept of term *polymorphism* in Elm is different from polymorphism in Java. In Java, polymorphism is implemented by *overloading*, which only preserves the identifier and signature of a function or method but allows rewriting the entire algorithm of the function. In Haskell, ML, and Elm, a polymorphic function not only keeps the same signature but also follows the same algorithm even if it accepts different types. This kind of polymorphism is called *parametric polymorphism* [App04]. A type variable α_i stands for a formal parameter in type signatures. And an instantiated type t_i can *substitute* it.

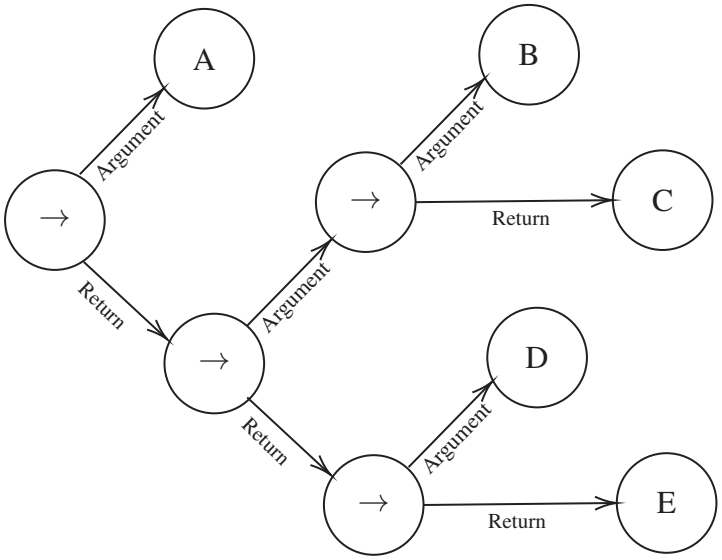

Figure 3.2.1: Abstract syntax tree of type `A -> (B -> C) -> D -> E`

Substituting α_i for t_i means in the scope of a type expression t, all the occurrences of α_i is substituted by t_i on the premise of avoiding *variable capture*. We will elaborate on this concept in Subsection 3.2.3.

3.2.2 Abstract syntax tree

Abstract syntax tree (abbrev. **AST**) is a tree-based data structure that contains information of a program. Tree is a subset of **connected acyclic graphs**. It is a finite set of $n(n > 0)$ nodes and paths. In a tree data structure, nodes are organized in a layered formation, and it ensures that there is always a path from the root node to any other one in the node set. Tree can also be denoted as a **partially ordered set** $(T, <)$ in set theory.

An AST is constructed after the syntax analysis phase during the compilation. Compiler transforms tokens into legal expressions and statements which can be denoted by nodes in the AST, and later be sent to semantic analysis phase where all the statements will finally be used for object code generation. The compiler convert AST into object code by performing a recursive tree traversal.

In Moogle, transforming the type signature of a searchable API into an AST is necessary. The AST creates the model and stores the semantic information of a complex type signature. For instance, a function `A -> (B -> C) -> D -> E` can be denoted in the AST structure as Figure 3.2.1 indicates.

As we mentioned in Chapter 1, the major conceptional difference between FP and Object-Oriented languages is that the function could be a concrete type (which is similar to the concept of function pointer in C/C++) and can be assigned as value to a variable, or be taken as an argument in another function.

Functions in Elm refers to a mapping relation and it maps the values of a type to another. All functions in Elm are **curried** which accept only one argument each time and output another type. A function `A -> B -> C -> D -> E -> ...` could be interpreted as `A -> (B -> (C -> (D -> (E -> ...))))`. For instance, in Figure 3.2.1, the type is equivalent to the signature `A -> ((B -> C) -> (D -> (E)))`.

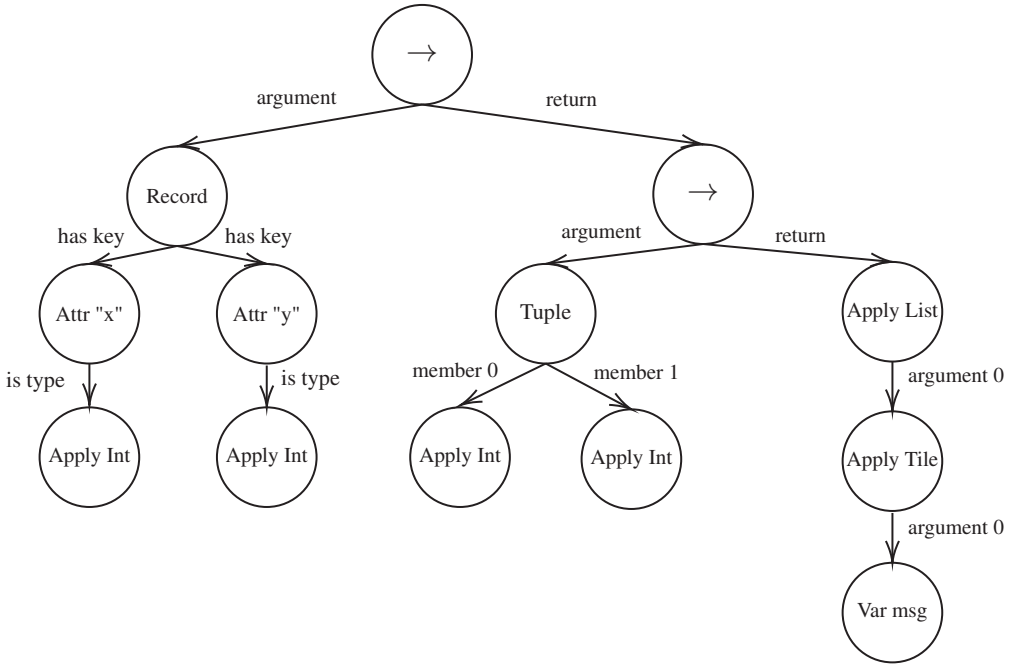

Figure 3.2.2: Abstract syntax tree of `{x: Int, y: Int} -> (Int, Int) -> List (Tile msg)`

Consider another case: `{x: Int, y: Int} -> (Int, Int) -> List (Tile msg)` where composite types appears. Applies, tuples and records can also be denoted as sub-tree structures in AST. And constructor arguments, tuple members and record attributes are presented as the child nodes of that type. Figure 3.2.2 demonstrates one possible representing of the type in AST.

The cases above suggest that it is feasible to extract sufficient type information in a type declaration expression from its AST. One of the solutions is designing a Domain Specific Language compiler and convert the type signatures into AST models, which will be explained in Chapter 4. The next subsection will elaborate on the core algorithm of performing type matchings.

3.2.3 Type unification algorithm

In the previous content, the documentation expounds essential concepts and knowledge on the syntax and grammar of Elm to the readers. A type in Elm may include type variables, which can be inferred to any static type by the compiler during usage. This dynamic language feature grants flexibility to the programmers and projects. Nevertheless, it brings the primary problem on API searching.

As the documentation mentioned in the start of this section, types with different signatures in Elm can be "compatible", which allows assigning the value of one to the other. This mechanism is called *unification*, which is a process of discovering a set of *substitutions* that makes a given term equivalent to the other [MM82].

The problem of finding finite set of solutions is also known as "unification problem", which was first proposed in Robinson's work [R+65] in 1965. In the context of comparing two type signatures, the problem is defined as: Given a pair of terms t, u, find finite sets of equations as solutions S to make t and u identical. A substitution s is a solution of the

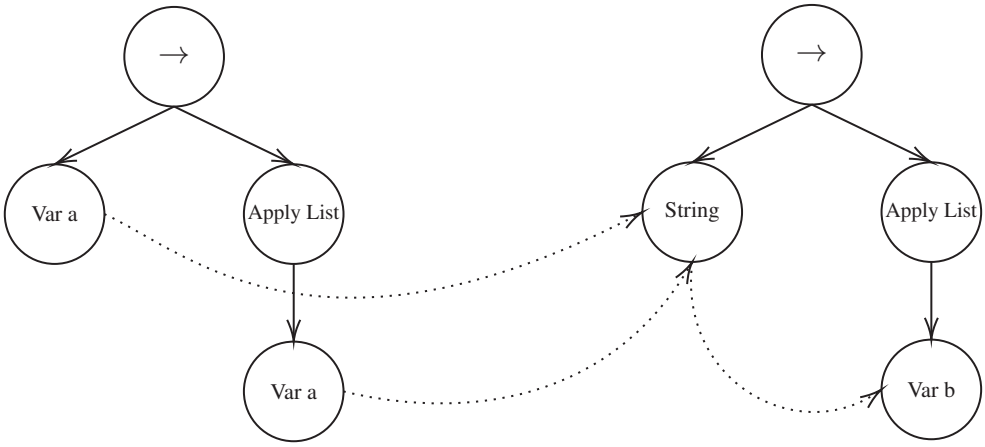

Figure 3.2.4: Type unification with AST visualization

problem when $s(t) = s(u)$. Here a substitution is a set of mappings from term x_i to term y_i (denote as $x_i \mapsto y_i$). If there exists at least one s to make S non-empty ($\exists s \in S,\ S \neq \varnothing$), then we say that t and u are *unifiable* and s is called a *unifier*. s is the *minimal unifier* or *most general unifier* (abbrev. *mgu*) when any other unifier θ can be presented by applying a substitution on s (based on renaming of s), which can be denoted as Equation 3.6 shows.

$$(s, \theta \in S) \wedge (\forall \theta, \exists \eta,\ \eta \circ \theta = s) \Leftrightarrow s \text{ is the minimal unifier of } t \text{ and } u \tag{3.6}$$

Our problem on two types t and u is to find a most general unifier s that makes t and u equivalent, or prove that t and u are not unifiable. A referable algorithm from [Hil19] can be presented as Algorithm 3 shows. We define the procedure as a function **unify** which accepts two terms t and u as inputs, and it outputs whether t and u are unifiable. In additional, we denote the set of type variables as V and the set of instantiated types as C. t_i, u_i represents the i-th argument of the instantiated type, provided that $t_i, u_i \in C$. Notice that the definition of function $binding(T)$ is shown in Equation 3.7.

$$binding(T) = \begin{cases} binding(T') & \text{if } T \text{ has been bound to } T' \\ T & \text{otherwise} \end{cases} \tag{3.7}$$

As we have mentioned in Subsection 3.2.1, the signature of a type in Elm can be divided into three categories: instantiated types, type variables and functions. Specially, a function could be denoted as a subset of instantiated types once given a new meta type $\rightarrow$. In Algorithm 3, we are applying this precondition that functions are a special type of instantiated types so that there are only two cases to be considered.

```
1        x : a -> List a
2        y : String -> List b
```

Figure 3.2.3: Type unification example 1

Consider the case in Figure 3.2.3 as an example. We use the Algorithm 3 to unify signatures `a -> List a` and `String -> List b`. The process of algorithm can be

Algorithm 3: Type unification algorithm

Input: Term t, Term u
Output: t and u are unifiable or not

```
1  Function unify(t, u)
2      t ← binding(t);
3      u ← binding(u);
4      if t = u then
5          return True;
6      end
7      if t ∈ V then
8          bind t to u;
9          return True;
10     end
11     bind u to t;
12     if u ∈ V then
13         return True;
14     end
15     if (t ∈ C) ∧ (u ∈ C) ∧ (argumentSize(t) = argumentSize(u)) then
16         unifiable ← True;
17         for i ← 1 to argumentSize(v) do
18             unifiable ← unifiable ∧ unify(t_i, u_i);
19         end
20         return unifiable;
21     end
22     return False;
23 end
```

visualized by applying AST as Figure 3.2.4 shows. We sketch out the entire process as follows:

1. Let t = `a -> List a` and u = `String -> List b`. Bind u to t since u and t are not type variables.
2. Both u and t are instantiated types with same arguments. We calculate on unify(`a`, `String`) and unify(`List a`, `List b`).
3. In unify(`a`, `String`), bind `a` to `String` since `a` is a type variable and returns true.
4. In unify(`List a`, `List b`), bind `List b` to `List a` then continue on unify(`a`, `b`).
5. Since `a` has already been bound to `String`, we set t = *binding*(`a`) = `String`. Then we bind `b` to `String`. As t = `String`, u = `b` and `b` is a type variable, unify(`a`, `b`) returns true.
6. The unification on all the arguments returns true. Hence unify(`a -> List a`, `String -> List b`) returns true.
7. The mgu is { `a` $\mapsto$ `String`, `b` $\mapsto$ `String` }.

3.2.4 Unification pitfalls and solution

The previous subsection introduces the algorithm of type unification. Nevertheless, It is worth to pay attention to a few pitfalls and corner cases.

Consider solving type t = `a` and u = `List a`. If we apply the Algorithm 3 on t and u, then the mgu for this problem is s = { `a` $\mapsto$ `List a` } and returns true, meaning that t and u are unifiable. It causes a problem when developers tries to instantiate `a` with a primitive type `Int`. The algorithm causes an infinite loop when applying on u with both constraints `a` $\mapsto$ `List a` and `a` $\mapsto$ `Int`, making u = `List List List ... Int` with infinite type constructors `List`. The logic fallacy on the case above is that the algorithm in Algorithm 3 is a naive process for doing unification on symbolic expressions instead of a concrete language. In Algorithm 3 the consequence of *recursive bindings* and *variable capture* is ignored as long as the equations are satisfied.

Avoid recursive bindings. In order to avoid the conflict of recursive bindings, we add another constraint on the algorithm. A referable approach [Ben18], which derived from Norvig's paper [Nor91] in 1991, is to define two new functions **unifyVariables**(v, t) and **occursIn**(v, t) where v is a type variable and t is a term. Function **unifyVariables** is defined as Algorithm 4 and **occursIn** is delimited as Algorithm 5 shows.

The critical concept in these two functions is that when unifying a variable with another term the algorithm always check whether there is a recursive occurrence. Given the case above, **occursIn**(`a`, `List a`) should return True since there is a duplicated occurrence for type variable `a` in `List a`. Hence, `a` can not unify with `List a` when recursive unification never happens.

Avoid variable capture. Consider another case when we tries to unify the signature of two APIs in Figure 3.2.5. Applying unification algorithm on the signatures of `funcX` and `funcY` returns *False*, since a conflict rises for two constraints {`a` $\mapsto$ `Int`} and {`a` $\mapsto$

Algorithm 4: Function unifyVariables

Input: Type variable v, Term t
Output: v and t are unifiable or not

```
1  Function unifyVariables(v, t)
2      if v ≠ binding(v) then
3          return unify(binding(v), t);
4      end
5      if (t is a type variable) ∧ (t ≠ binding(t)) then
6          return unify(v, binding(t));
7      end
8      if occursIn(v, t) then
9          return False;
10     end
       /* v and t have no bindings, hence we bind v with t directly
          */
11     bind v to t;
12     return True;
13 end
```

Algorithm 5: Function occursIn

Input: Type variable v, Term t
Output: Whether type variable v occurs inside term t

```
1  Function occursIn(v, t)
2      if v = t then
3          return True;
4      end
5      if (t is a type variable) ∧ (t ≠ binding(t)) then
6          return occursIn(v, binding(t));
7      end
8      if t is an instantiated type then
9          occurence ← False;
10         for i ← 1 to argumentSize(t) do
11             occurence ← occurence ∨ occursIn(v, t_i);
12         end
13         return occurence;
14     end
15 end
```

`Float`}. The search engine, however, shall not accept it as an expected behavior. In Elm, the name of a type variable in one signature should not be confused with the same name of another type variable. Changing the name of a type variable does not affect the definition of a type. For instance, the signature `identity : a -> a` of an API is absolutely equivalent to type signature `identity : x -> x`. Hence, it is necessary to take approach against this issue, which is known as *variable capture*.

```
1  funcX : Int -> List a -> Float
2  funcY : a -> List Float -> Float
```

Figure 3.2.5: Type unification example 2

The root cause of variable capture is that a type variable is free in a type signature until we bind it with a substitution constraint. Literature [Muk04] provides an solution to the problem by defining two additional sets $FV(M)$ and $BV(M)$. The former represents the set of free type variables, while the latter stands for bound ones. In the example of Figure 3.2.5 when the program tries to unify `Int` in `funcX` with `a` in `funcY`, the type variable `a` of `funcY` becomes bound and shall be moved to $BV(M)$, while `a` of `funcX` remains free to be unified. For simplicity, we rename this free type variable to another name (For instance, `a0`) for averting confusion. Therefore, `funcX` and `funcY` are unifiable with the unifier $\{$ `a` $\mapsto$ `Int` , `a0` $\mapsto$ `Float` $\}$.

3.3 Ranking metrics on type signatures

The previous section dilates on concepts and algorithms for types. And in this section, we define several metrics for type signatures regarding the rankings for results.

1. **Size of type**: In this thesis, the *size* of a type refers to the structure complexity of a type. The size of type t is denoted as $size(t)$ and defined as Equation 3.8, where $t_{argument}$ refers to the type of the argument, t_{return} refers to the type of return when t is a function. In the second expression, n stands for the size of constructor arguments and t_i represents the type of i-th argument.

$$size(t) = \begin{cases} 1 + size(t_{argument}) + size(t_{return}) & \text{if } t \text{ is a function} \\ 1 + \sum_{i=1}^{n} size(t_i) & \text{if } t \text{ is an instantiated type} \\ 1 & \text{otherwise} \end{cases} \tag{3.8}$$

2. **Variable diversity**: The *variable diversity* of type t refers to the occurrence of different type variables in type t. For instance, the variable diversity of type `a -> a` is 1 as there is merely one type variable `a` appears in the signature. Similarly, the variable diversity of `(a -> b) -> List a -> List b` is 2.

In summary, several terms of metrics are defined in this section, which will be used for practical usage when implementing the ranking process.

Chapter 4

Designs and implementation

This chapter introduces the details of implementing the search engine. Section 4.1 introduces the architecture of our work and the trade-off decisions that we made. In Section 4.2, we look into the structure of elm packages and analyze the categories of APIs. In Section 4.3, the definition of search DSL, MoogleQL, would be given. Section 4.4 introduces Neo4j database and how we construct the storage data structure. Section 4.5 presents the internal logic and procedure of our front-end and back-end system where search queries are processed. And finally, in Section 4.6 we discuss the ranking metrics for the results.

4.1 Architectural design

In this section, we introduce the architectural design of Moogle. According to Bass etc. [BCK03], the software architecture of a system is "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and relationships among them". Architectural design is a global, high-level design that reflects the overall structure of a system. In the following content, we define the major components, the relationships among them and rationales behind the components and structures.

4.1.1 Architectural style overview: REST

Moogle is a *RESTful Web Service* which conforms to *REST (Representation State Transfer) architectural style*. First introduced by the Roy Fielding's dissertation [Fie00] in 2000, REST has become one of the most widely-used styles in recent years. Unlike traditional styles, REST, relatively speaking, is a composite style which aggregates constraints and properties from multiple architectural styles and patterns as Figure 4.1.1 shows.

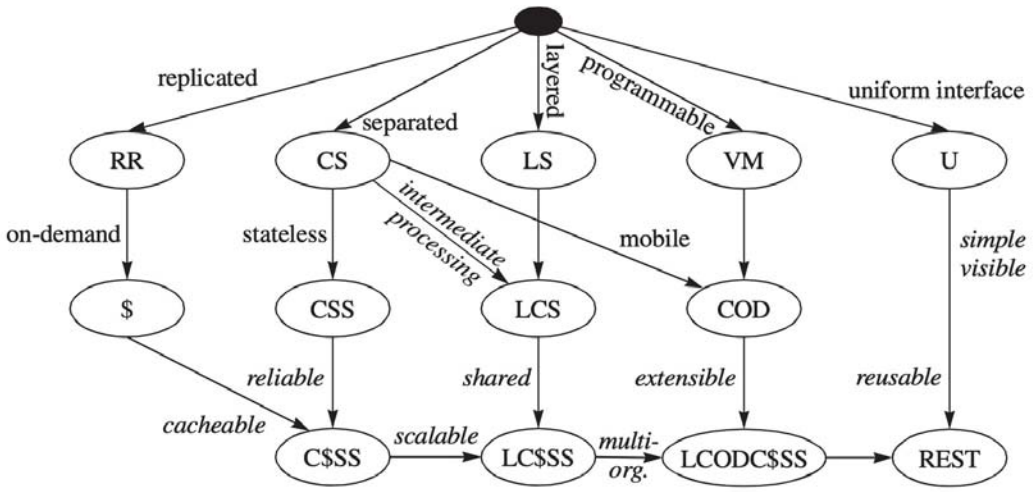

Figure 4.1.1: Derivation of REST given by Fielding's work [Fie00]

Although REST defines constraints and properties that regulates the design, it is a style rather than an implementation. In the following subsection, we will introduce the architectural implementation of Moogle which derives from REST.

4.1.2 Architecture of Moogle

Providing a RESTful Web Service back-end and a Single-page application (abbrev. SPA) as its front-end, the architecture of Moogle is divided into the following components:

1. **Database component**: In Moogle, the database component is Neo4j database. All the information that crawler scratched from Elm packages will be stored and persisted in the database. We will elaborate on the decisions and designs that we made in Section 4.4.

2. **REST API component**: This component comprises RESTful APIs that provides external interfaces via HTTP protocol. As Moogle adopts the uniform interface constraint from REST, several Web APIs with semantic URLs that can be invoked by HTTP requests are made.

3. **Parser component**: This component is consist of abstract data models (for instance, term types classes) and the parser of MoogleQL that transform signature strings into predefined data structures, which can later be leveraged for type unification matchings, queries or database import.

4. **Data access component**: This component plays the role as an intermediate layer of abstraction between the database and Web services. All search requests will be parsed into Cypher query language and then send to the data persistence module.

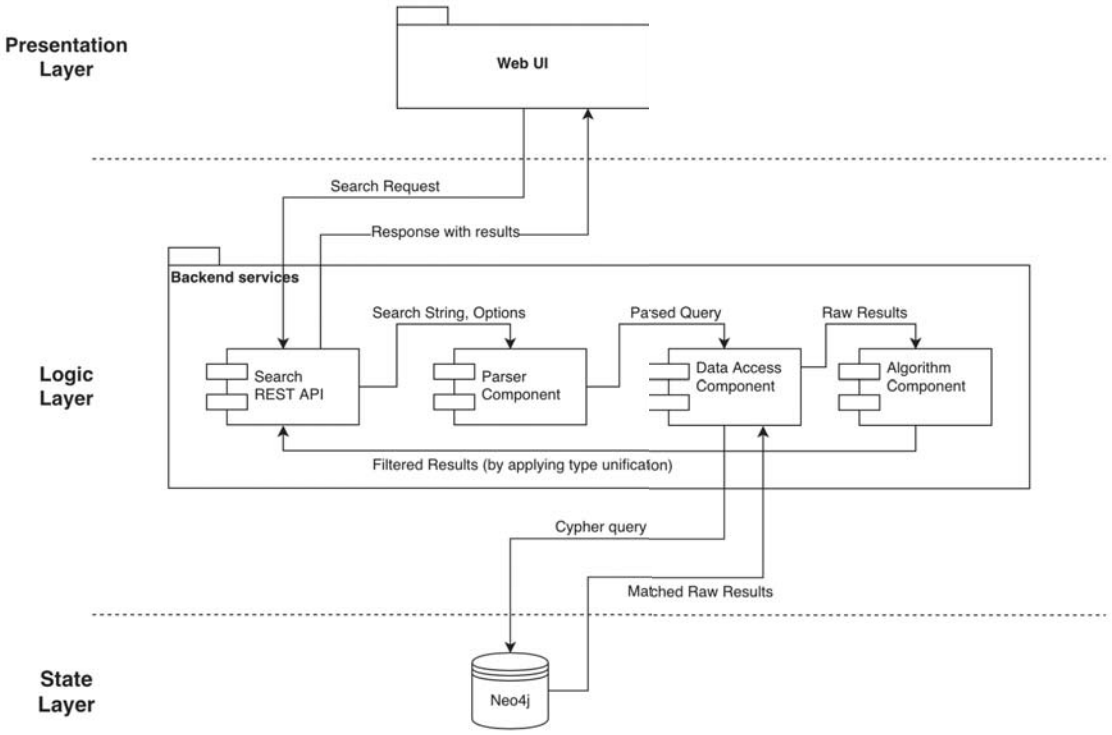

Figure 4.1.2: Modular decomposition view

5. **Algorithm component**: This component is composed of core algorithms (including type unification) and metrics that will be finally used for filtering and ranking raw results from database.

6. **User interface component**: The user interface component represents the front-end UI that will directly interact with the users.

In the design of Moogle, REST API component, Parser component, Data access component and Algorithm component are integrated inside **back-end services module**, while the **data persistence module** is mainly consist of the database component itself, and **user interface module** is constituted by the user interface component. Figure 4.1.2 shows the modular decomposition view of Moogle, where the direction of data exchanges are marked as arrows.

Figure 4.1.3 demonstrates the allocation view of Moogle, where some tactics are used to conform the constraints given by REST style. For instance, static files are required to be cached in local environment of the end-user. Different from traditional Model-View-Controller pattern, the front-end module of Moogle only serves SPA(Single-page application) static files including the rendering scripts. This pattern promotes the performance of the server by reducing computational resource cost on server-side rendering.

Figure 4.1.4 shows the deployment layered view of Moogle. The deployment of the entire software system is based on virtualization technique provided by Docker and services are wrapped as containers which effectively boosts the portability of the search engine. Persistence module (Neo4j), Web services module and user interface module are deployed isolated. Communication between modules is transmitted through the bridge

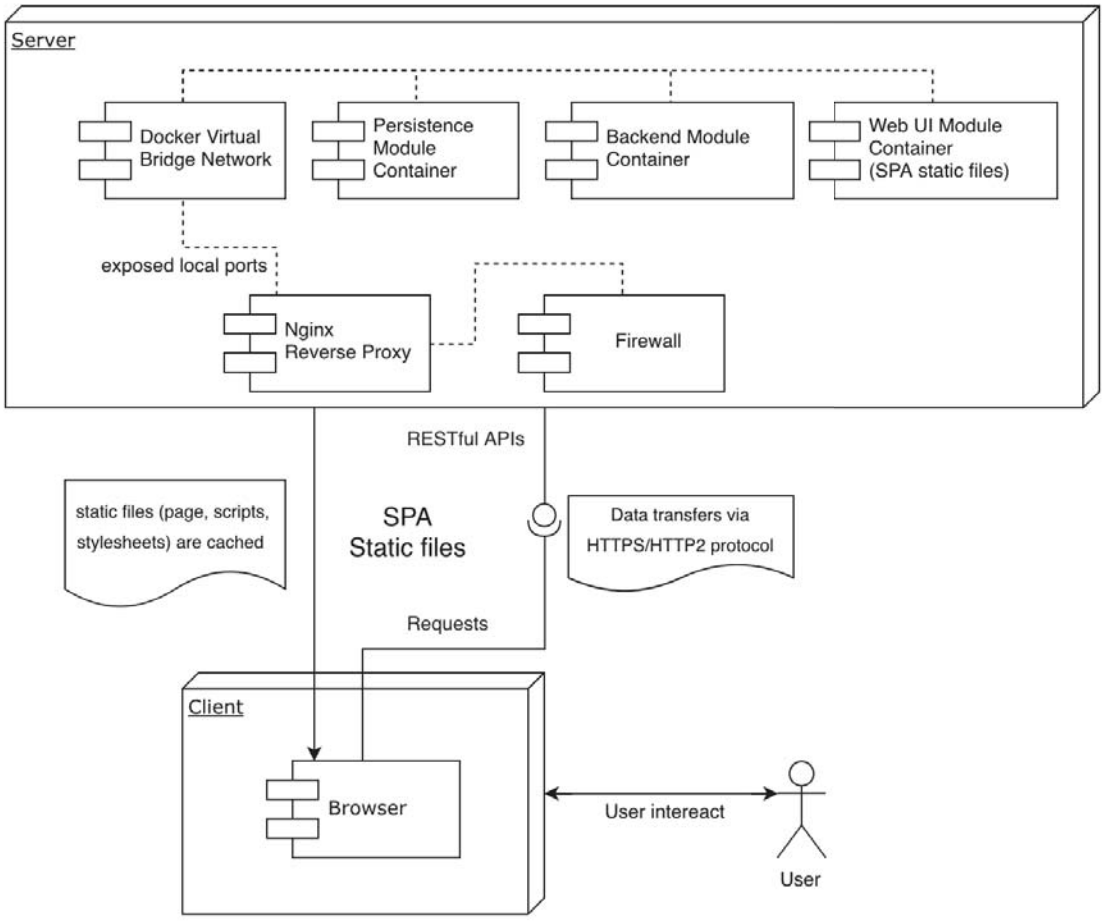

Figure 4.1.3: Allocation view

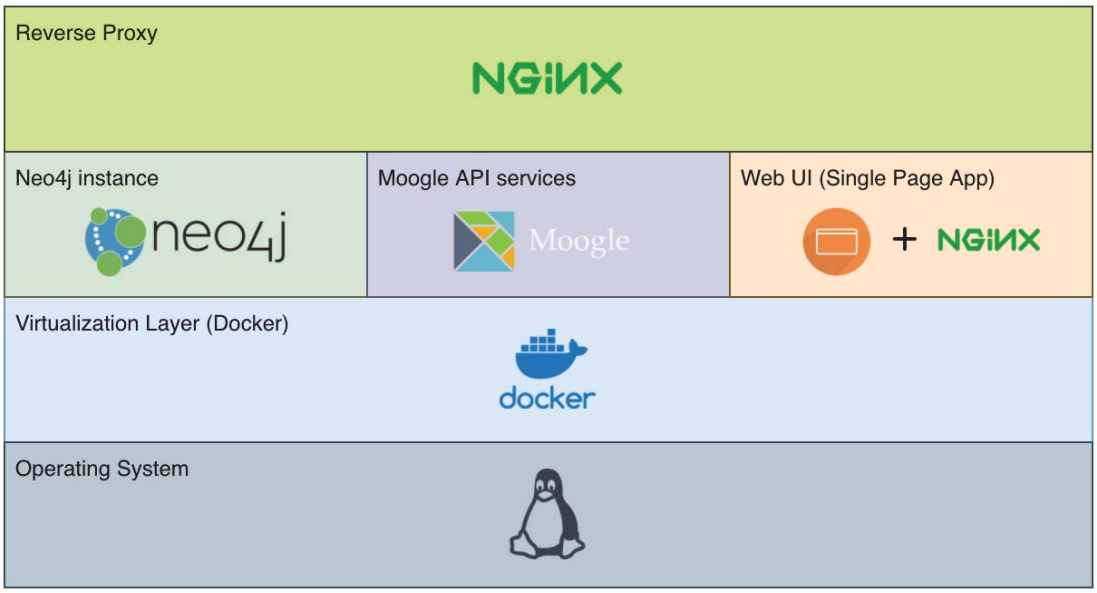

Figure 4.1.4: Layered view of Moogle's deployment

network inside Docker.

4.2 Resolving Elm packages

To build up our search engine, looking into the structure of Elm packages is necessary. Elm-Package is an on-line package repository that stores all the uploaded third-party libraries, and developers can retrieve documentations and meta information of packages via specific APIs. Figure 4.2.1 shows an example documentation web page of an Elm package.

As the goal of this work demands searching APIs from all available repositories, it is essential to analyze the structure and the exposing APIs of each Elm package, which is consist of one or multiple modules. By inspecting the meta data, we conclude three primary API patterns that an Elm package could provide, which are listed as below.

1. **Union types**: In Elm, custom types we created with keyword `type` are called union types, which is analogous to a disjoint union set of tagged elements. For instance, the source code in Figure 4.2.2 defines a union type named `User`. All the cases are listed each line with a unique tag. We usually name these tags as term *type constructors*. Union types are similar to a combination of enumerations and structs in OO languages which aggregate the functionality of constructing complex data structures and giving enumerations.

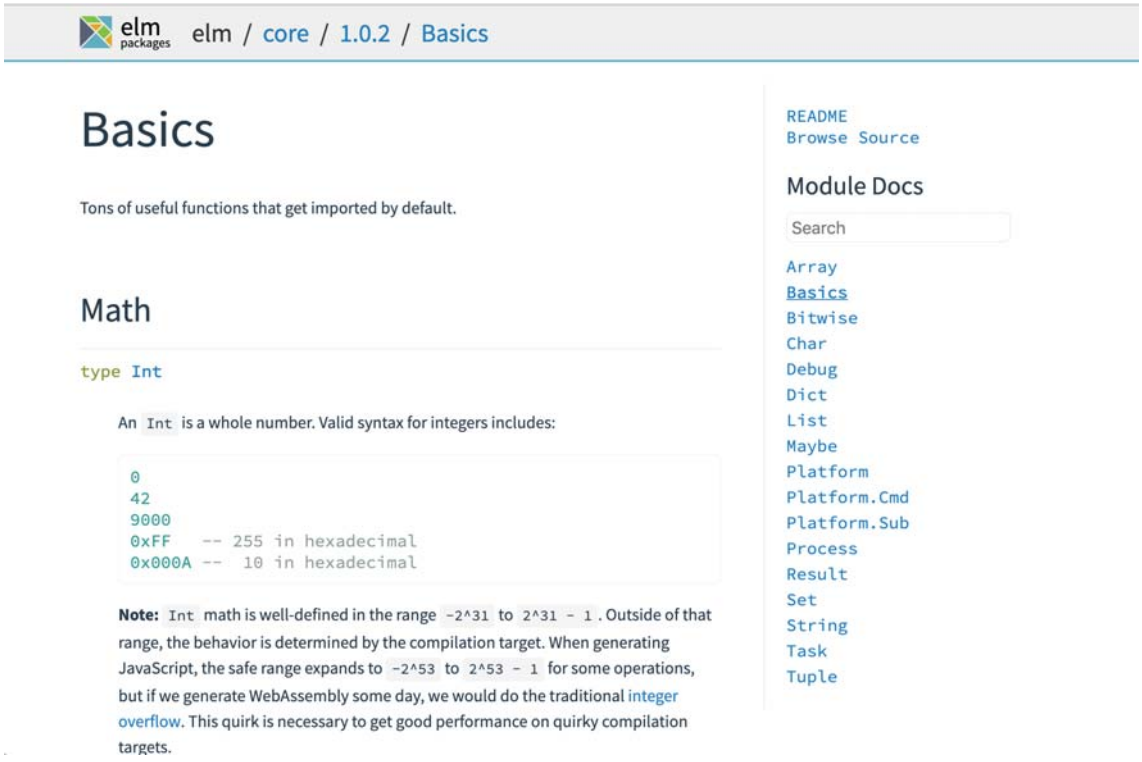

Figure 4.2.1: Elm package example

```
type User
= Anonymous
| Visitor String
| FullMember Int String
```

Figure 4.2.2: Union type definition example

In Elm packages, the definition of each union type are represented in the documentation of each module. By resolving these union types we could extract the name of each union type and their type constructors. Constructor parameters are also useful since they build up the signature of the type constructors.

2. **Aliases**: A type alias refers to a different "name" to the same type. In modules, record types (including extensive records) are usually assigned to an aliased type name in order to make the code readable. In addition, functions and union types can also be aliased. Aliases are declared with keyword `type alias` and type variables should also be included in the signature once they are involved in the original type. Figure 4.2.3 gives an example of defining a type alias in Elm.

```
type alias TextRendering compatible = { compatible |
  ↪ value : String.String, textRendering : Css.Compatible
  ↪ }
```

Figure 4.2.3: Alias type definition example

3. **Values**: They are the major component of a package and they are the primary target to be retrieved by Moogle. In Elm packages, the type of a value could be automatically inferred without any explicit declaration to a specific signature once it is defined. Considering avoid misunderstanding, here "value" could be assigned with an Apply, a record, or a function.

   Figure 4.2.4 presents some examples of the signature of values.

```
x : Int
add : number -> number -> number
acos : Float -> Float
reverseMap : (a -> b) -> List a -> List b
toHtml : List (Attribute msg) -> String -> Html msg
nodeNS : String -> String -> List (Attribute msg) -> List
  ↪ (Node msg) -> Node msg
```

Figure 4.2.4: Value type signature example

4.3 Building MoogleQL parser

Since Moogle is a type-based search engine, we have to deal with type signatures and convert them into data structures that can be ultimately leveraged by our algorithms. Intuitively, the easiest solution to parse an Elm type signature expression is to use the compiler itself. Elm, however, is a Web-oriented language that can only be compiled to browser-oriented JavaScript, and a browser environment has great difficulties on invoking system IO. Although Elm provides a new feature called *port* in recent updates, this new language mechanism still heavily depends on external JavaScript snippets that may not benefit maintainability.

On the other hand, the search query that a user made may distinct to the syntax of a real Elm signature. Take Hoogle as an example, it accepts search queries in the format of "+package_name keyword1 keyword2 keyword3 :: _ -> Int" which is actually an invalid type expression according to the syntax of Haskell. Hence, it is obliged to find another solution for building a generic Domain-Specific-Language (abbrev. DSL) parser that can parse not only type signatures, but also search queries. We name our query DSL as **MoogleQL**.

Building a parser for MoogleQL is no picnic, since parser generator tools like Lex and Yacc are based on C language and they are difficult to master. But fortunately, Python has a parser generator implementation similar to Lex and Yacc called *Ply* [Bea19]. As a

compiler generator, Ply does not execute the program, instead, it analysis the expressions from source code. Ply is consist of two parts: the lexical, and syntax analyzer which supports LALR(1) parsing. Comparing to the similar tools like Antlr, Ply provides adequate functionalities for fast prototyping without losing significant performance.

Introducing the syntax of Lex and Yacc thoroughly would be a great work. Instead, we directly dive into the core parts and explain the key concepts that do the work.

4.3.1 A glimpse of MoogleQL

Before giving a concrete defining on our syntactic rules, we elaborate on the overall design of MoogleQL and presents a general description of the syntax.

The format of MoogleQL is closely similar to a *type annotation* expression in Elm, but MoogleQL is a superset of Elm's type annotation expression, where additional features are added with respect to use case scenarios that we mentioned in Chapter 1. An example query string is given in Figure 4.3.1 where some key differences are labeled.

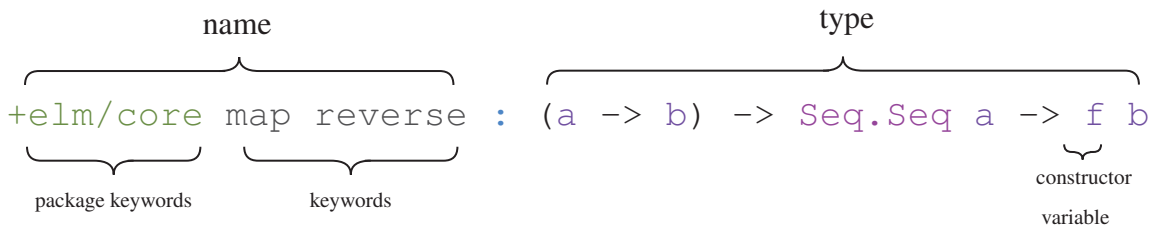

Figure 4.3.1: Sample query string of MoogleQL

MoogleQL is always matching one of the patterns below. If a search query fails to match any one the these patterns will be regarded as invalid syntax, and Moogle will hint user to check his or her inputs without executing any search.

1. **Name**, which refers to a group of name keywords, or package constraints, or both. A package constraint, which starts with a "+" character, stands for searching within specific packages whose names contain the substrings. When there is no package constraint, Moogle searches all the packages.

2. **Type**, which refers to the constraint on type expression of an API. The extension part of the syntax is that MoogleQL borrows the concept of *constructor variable* from Haskell, whereby can uncertain union type tags be substitute as a type variable.

3. **Name : Type**, where two sub-expressions are joined together with a colon character as the case above shows.

4.3.2 Divide and conquer

Based on the descriptions above, it is interesting to note that the grammar of a **Name** expression is relatively loose while the syntax of **Type** is rigorous. Hence, the parser of

MoogleQL can be divided into two parts: The name, and type parser. The former is based on transforming name constraints via simple regular expression pattern matchings, while the latter is based on a complete parser which contains lexer and syntactical analyzer. In order to provide a uniform interface, we set up a search query builder class as its entry, which split up the search query then divide-and-conquer the remaining work by utilizing the name and type parser separately.

The workflow of this mechanism can be represented in Figure 4.3.2. In Moogle, the name parser is built by a series of combination on regular expressions, which is easy for implementation. Another consideration on adopting this solution is that the process of parser would consume a major computational and memory resources, which is not benefiting on the performance.

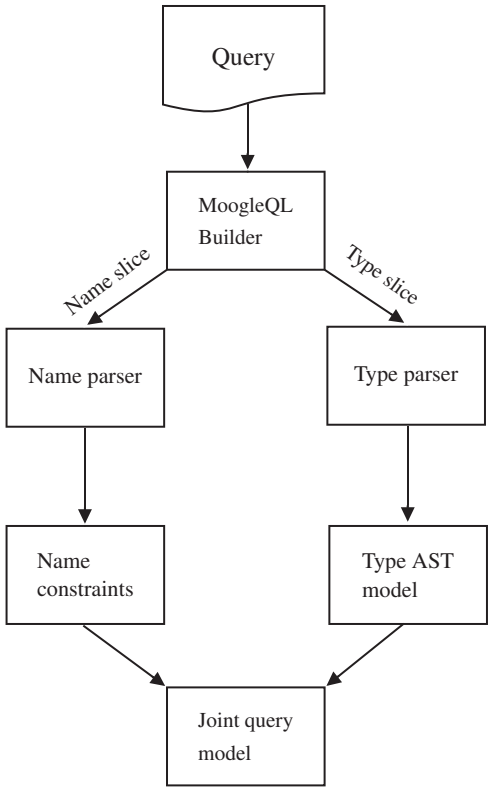

Figure 4.3.2: MoogleQL parser workflow

In the next subsections, we will focus on the construction of type parser that is consist of a lexer and syntactical analyzer.

4.3.3 Constructing type lexer

Lexical analysis is generally the first step for compilers, and the task is assigned to a program called *lexer*. Before analyzing the syntax and expression patterns, the compiler must define a set of lexical tokens so that the tokenizer can identify the type of words and their additional information. The definition syntax of lexical tokens is similar to the rules of regular expressions. In Ply, the tokens are defined in a list with concrete patterns. In MoogleQL, we define the tokens as Figure 4.3.3 shows.

```
1         tokens = [
2             "CAPITALIZED_WORD",  # '[A-Z](?:[a-zA-Z0-9_]*)'
3             "LOWERCASE_WORD",    # '[a-z_](?:[a-zA-Z0-9_]*)'
4             "RETURN",            # '->'
5             "LPAREN",            # '('w
6             "RPAREN",            # ')'
7             "COLON",             # ':'
8             "COMMA",             # ','
9             "DOT",               # '.'
10            "LBRACE",            # '{'
11            "RBRACE",            # '}'
12            "VERTICAL_BAR",      # '|'
13        ]
```

Figure 4.3.3: Tokens of MoogleQL

The list of tokens defines which patterns of strings are "valid spellings" and any exceptional pattern will be regarded as illegal token since the tokenizer cannot recognize it without explicit definition. Note that in MoogleQL all the spaces, tabs and line break characters will be taken as spacers for tokens.

4.3.4 Constructing type syntactical analyzer

A parser, or syntactical analyzer collects the sequence of tokens that parsed from the previous step and continue on parsing the syntax of source code. The definition of syntax is similar to defining types in FP languages. In Yacc, a *context-free grammar* is defined by a four tuple (N, T, P, S), where N represents non-terminals, T stands for terminals (in this context, it is equivalent to the tokens that tokenizer produced in the previous steps), P represents productions and S denotes the start symbol that can regarded as the root of entire program [V.18].

The definition of an production is divided into two parts by a colon. The left part of each production is a non-terminal and the right parts is consist of a terminals and non-terminals sequence that separated by "|". Note that in Yacc, the syntax cannot be ambiguous. Conflicts of ambiguity (*shift/reduce conflicts*) should be solved by either changing the syntax or defining precedence explicitly.

The root syntactic expression of MoogleQL is defined as Figure 4.3.4 shows. The expression `typeExpr` is the start symbol of entire type, which is potentially empty when the user only search by names. A `nonVariableType` refers to instantiated types, which includes: `apply`, `tuple`, `record` and `extensibleRecord`.

The non-terminal `apply` can be expanded as Figure 4.3.5 shows. Note that we defines an extra non-terminal `singleApply` and `typeList` to be non-empty in order to solve potential shift/reduce conflicts. Line 10 suggests that an `apply` can be inferred as a type variable with a list of type arguments, which conforms to the expanded syntax rule in Subsection 4.3.1.

```
typeExpr : /* empty */
| type
;

type : nonVariableType
| typeVariable
;

nonVariableType : type RETURN type      /* Functions */
| LPAREN type RPAREN
| tuple
| apply
| record
| extensibleRecord
;

typeVariable : LOWERCASE_WORD
;
```

Figure 4.3.4: Root syntactic expression of MoogleQL

On the other hand, Figure 4.3.6 shows the syntax of tuples and records, which are relatively easier to be defined.

Based on the syntax definitions above, semantic information of a type expression can be extracted during the parse phase. Hence, we have also define several data model classes such as Apply, Var, Function,Tuple and Record in order to carry these data and later be put into use on creating database and applying unification algorithm, but these implementations are trivial in this thesis because they are completely referring to the models we have mentioned in Chapter 3. Nevertheless, there is necessity to discuss the storage of these data in the next section.

4.4 Neo4j database and storage model

Now that we have built the parser of Moogle which provides the functionality of converting signature strings to data models in our system, we can continue on looking into Neo4j, the DBMS selected for Moogle and find the solution of how should the data models be constructed and stored.

Developed by Neo4j Inc., Neo4j is an ACID-compliant graph database management system that handles the creation, reading, updating, and deletion (CRUD) operations of graph data models [VB14]. Different from traditional relational databases, it is a NoSQL(Not Only SQL) database which stores graph as its first-class data model. A graph G, in graph theory, is a collection of vertices and edges and it can be denoted as a pair $G = (V, E)$. In graph databases, correspondingly, the terms become *nodes* and *relationships*. The application of graph models involves in social network (For instance,

```
apply : typeName typeList
| typeVariable typeList
| moduleName DOT typeName typeList
| singleApply
;

typeList : singleApply
| singleApply typeList
| typeVariable
| typeVariable typeList
| tuple
| tuple typeList
| record
| record typeList
| extensibleRecord
| extensibleRecord typeList
| LPAREN type RPAREN
| LPAREN type RPAREN typeList
;

singleApply : typeName
| unit
| moduleName DOT typeName
;

typeName : CAPITALIZED_WORD
;

moduleName : CAPITALIZED_WORD
| moduleName DOT CAPITALIZED_WORD
;
```

Figure 4.3.5: Syntax of `apply`

```
        tuple : LPAREN typeTuple RPAREN
        ;

        typeTuple : type COMMA type
        | type COMMA typeTuple
        ;

        record : LBRACE attributePairs RBRACE
        | LBRACE RBRACE
        ;

        extensibleRecord : LBRACE typeVariable VERTICAL_BAR
        ↪ attributePairs RBRACE
        ;

        attributePairs : LOWERCASE_WORD COLON type
        | LOWERCASE_WORD COLON type COMMA attributePairs
        ;
```

Figure 4.3.6: Syntax of tuples and records

Twitter and Facebook), information networks, semantic webs, knowledge representation, artificial intelligence and many other modern business scenarios.

4.4.1 Why graph database?

Comparing to relational DBMS and other NoSQL competitors, graph databases has several significant differences, which are listed in Table 4.1.

In the context of our thesis, the information of extracted type signatures should be stored in a database model. Unlike Elm-search, Moogle does not store type signatures in memory by adopting Mobile code style. The reason of this is that:

- The script size of signature and indexes grows larger and memory occupation may become a problem when the scale of packages grows.
- Browsers like Google Chrome set memory usage limit for each page tab. As the size of Elm packages grows, it cannot guarantee this limitation would not be exceeded.
- Furthermore, a large script may increase the load time of a web page, which is not recommended by guidelines.
- JavaScript runtime in legacy browsers may only support single thread execution, thus they are not suitable for parallel computation, which sets a limitation on performance.

On the other hand, adopting Neo4j as our persistence layer provides several advantages as below:

Table 4.1: Comparison on different categories of DBMS

Category	**Storage model**	**Advantages**	**Disadvantages**	**Representative Products**
Relational DBMS	Relational model	Suitable for most general scenarios	Inefficient when applying too many table joins	MySQL, Oracle, SQL Server
Key-Value Database	Hash Tables	High performance and throughput for queries	Data are unstructured	Redis
Document-based Database	Extensive key-value pairs	Flexible storage structure	Limited performance, lack of uniformed query language	MongoDB
Column-oriented Database	Columnar data storage	Scalability, read performance and compression rate	Limited application scenarios	HBase, BigTable
Graph Database	Graph model	Suitable for graph-oriented queries and algorithms	Limited on data sharding	Neo4j, OrientDB

- Neo4j provides native support on storing tree-based data structures like AST, which is a subset of graphs.
- Neo4j provides auto management on indexes once created, thus we don't take further actions on managing indexes after initialization.
- Cypher, the query language of Neo4j, provides flexibility when transforming a type signature to target queries.
- Third-party plug-ins, such as APOC, include optimized algorithms for out-of-the-box invoking.

Based on above reasons, applying Neo4j would guarantees the modifiability and scalability for future usage and changes on Moogle. Graph-based data model is the heart of Neo4j and it is the semantic model of a type signature and Neo4j will provide a semantic, uniform storage service in the design of Moogle.

4.4.2 Constructing data models

As the thesis mentioned in Subsection 3.2.2, a type signature expression can be converted into an AST that stores properties and relationships of each term and expression, making it the most widely-used data model for parsing a program. Although Moogle does not parse the entire source code of an Elm package, the API signatures and their AST are still the vital data source of our search engine. Hence, by applying Neo4j, we extract the

Table 4.2: Node schema for packages

Property Name	**Type**	**Description**
<id>	Integer	Automatically generated identifier for each node.
is_latest	Boolean	Whether a package is latest or not.
license	String	License information.
link	String	The documentation URL of the package.
name	String	The name information of the package, including its author.
is_official	Boolean	Whether it is the core(build-in) package from Elm.
version	String	The version of the package.

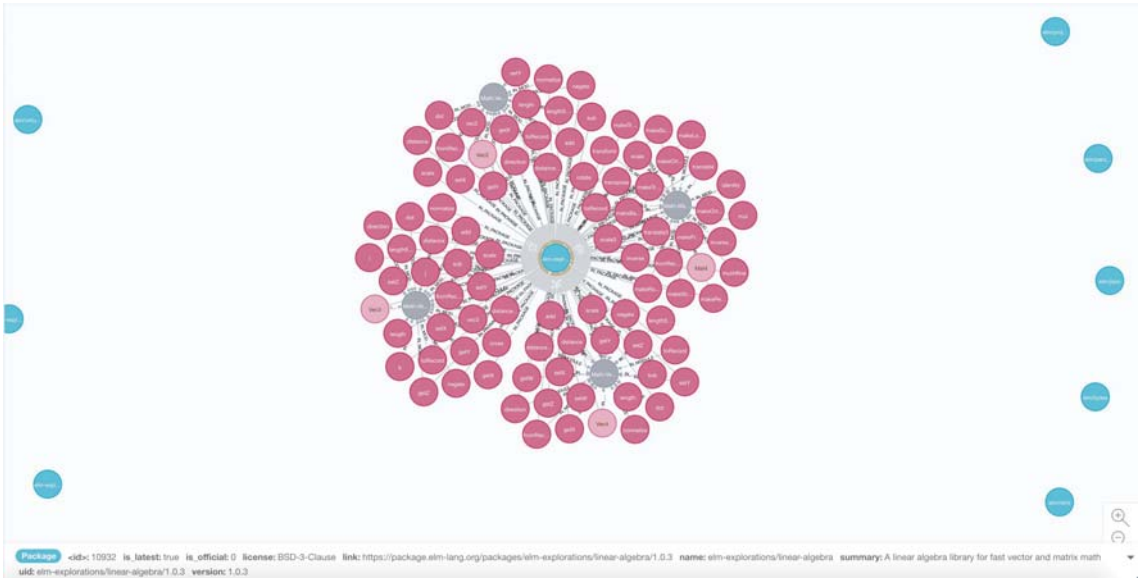

Figure 4.4.1: Package node in Neo4j

data generated from the signature parser and convert them into nodes and relationships then finally proceed persisting them.

Although AST itself covers enough information for a compiler to initiate semantic analysis, some improvements on nodes and relationships are employed in order to optimize storage schema. The following content lists the structure of different node types and the additional changes we have brought.

1. **Packages**: A package node presents a package entity, which is the root node of an entire package structure. It is consist of more than one modules, which is presented as a unidirectional relationship with label "HAS_MODULE". Table 4.2 shows the schema of a package node. An example package node "elm-explorations/linear-algebra" and its directly connected nodes in the graph are demonstrated in Figure 4.4.1.

2. **Modules**: A module is the secondary namespace of a composite code component in Elm. It contains APIs and guarantees there's no name collision between different

Table 4.3: Node schema for modules

Property Name	Type	Description
<id>	Integer	Automatically generated identifier for each node.
name	String	The name of the module.
link	String	The documentation URL of the module.
comment	String	Module specification in Markdown format.
package_name	String	The name of the package it belongs to.

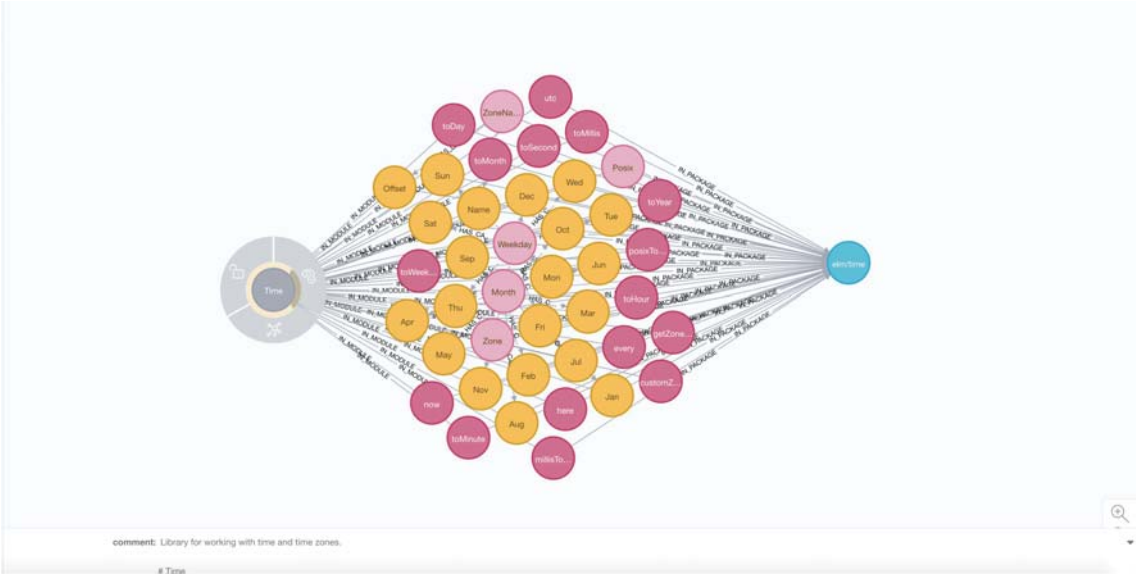

Figure 4.4.2: Module node in Neo4j

APIs. A module node connects to its package as a child node and also connects to its internal APIs as a parent node. Table 4.3 shows the schema of a module node. An example module named "Time" and its directly connected nodes in the graph are demonstrated in Figure 4.4.2.

3. **Values**: Values are the major component of Moogle's searchable nodes. A value is consist of a name and a type signature. The type of an value can be type variables, instantiated types and functions. The AST root of that type signature is directly connected under the value with the relationship labeled "IS_TYPE". Table 4.4 shows the schema of a value node. Figure 4.4.3 demonstrates a value node named "now" and the AST of its type signature `now : Task x Posix`. Notice that there is a relationship labeled "ARGS" between an "Apply" node (The green node) and its children. It records the index of the parameter with an internal property "order" so that Moogle can match the constructor without mismatching its arguments. Property "complexity" and "var_num" in the value node represents the metric concept that we have already introduced in Section 3.3.

4. **Unions and Constructors**: Union types are the customized types includes constructors and their arguments. The former can be regarded as a special function which accept zero or more constructor arguments and finally returns the Union

Table 4.4: Node schema for values

Property Name	Type	Description
<id>	Integer	Automatically generated identifier for each node.
name	String	The name of the value.
link	String	The documentation URL of the value.
comment	String	Value specification and usage in Markdown format.
module_name	String	The name of the module it belongs to.
package_name	String	The name of the package it belongs to.
signature	String	Type signature of the value
complexity	Integer	Type size measurement of the value
var_num	Integer	Variable diversity of the value

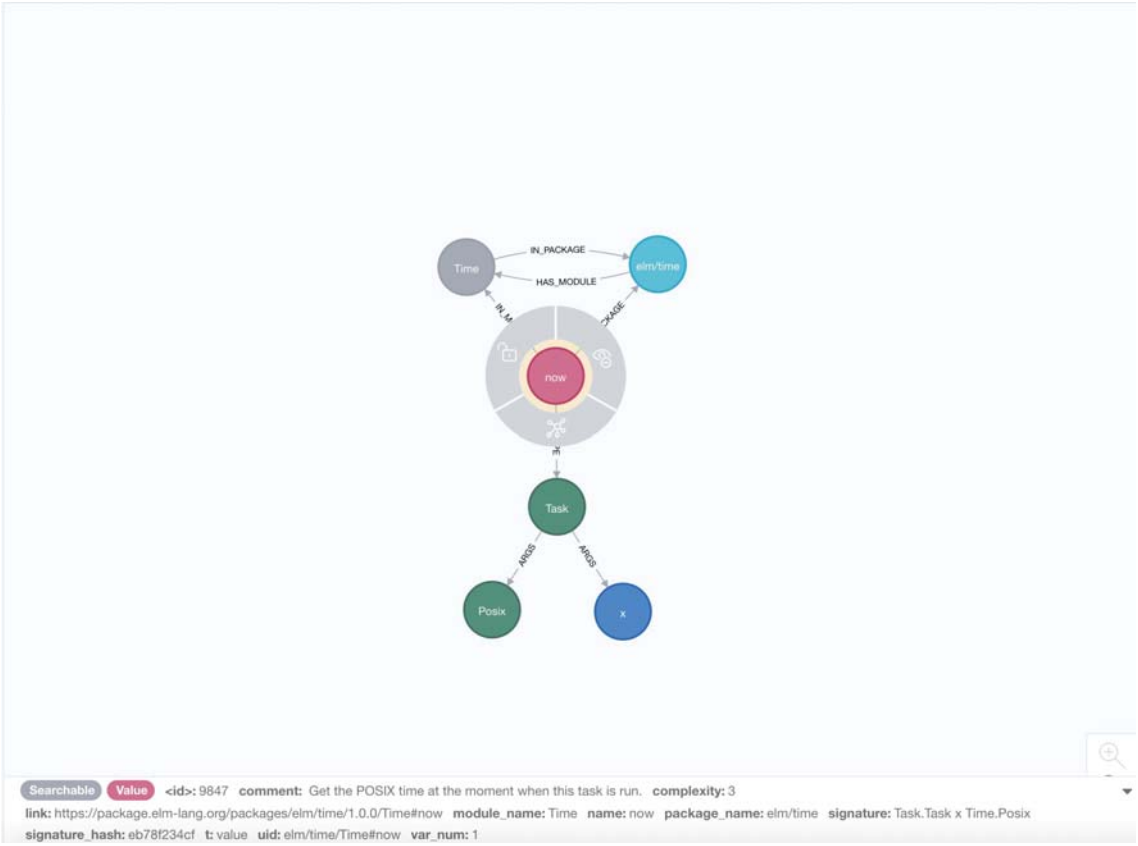

Figure 4.4.3: Value node in Neo4j

Table 4.5: Node schema for union types

Property Name	Type	Description
<id>	Integer	Automatically generated identifier for each node.
name	String	The name of the union type.
link	String	The documentation URL of the union type.
comment	String	Union type specification and usage in Markdown format.
module_name	String	The name of the module it belongs to.
package_name	String	The name of the package it belongs to.
signature	String	Type signature of the union type
var_num	Integer	Variable diversity of the constructor parameters

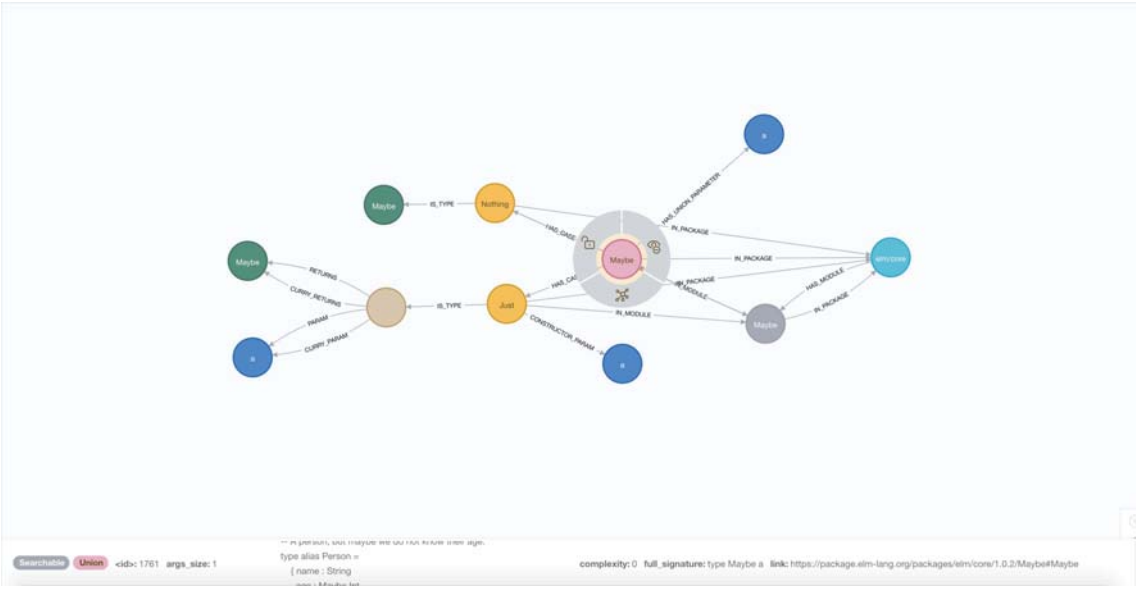

Figure 4.4.4: Union type node in Neo4j

type. In Moogle, both Union types and Constructors are searchable, which is a feature that we mentioned in Hoogle. The schema of Union types are listed in Table 4.5, and Figure 4.4.4 demonstrates the internal storage structure for build-in union type `Maybe`.

5. **Functions**: Functions are mappings which accepts one argument each time and returns another type. In Subsection 3.2.2, we had introduced the AST structure of functions. In database model, however, several changes are made for optimization or implementing some features.

   Every function node in Neo4j has a "CURRY_PARAM" relation links to its argument node in function currying format, and the returning one is linked by the "CURRY_RETURNS" node. In additional, we add links labeled "PARAM" directly from the root node of a function to all of its parameters and a link labeled "RETURNS" to the final return type, which is similar to a function that accepts multiple arguments and returns the final type. Figure 4.4.5 demonstrates the node structure for API `tileYCoord : Float -> Int -> Int -> Float`,

Table 4.6: Node schema for functions

Property Name	**Type**	**Description**
<id>	Integer	Automatically generated identifier for each node.
is_root	Boolean	Indicates whether this type node is directly connected by a searchable API.
signature	String	Type signature of the current node.
complexity	Integer	Type size measurement of the function.
args_size	Integer	Number of arguments accepted before mapping to the final return type.

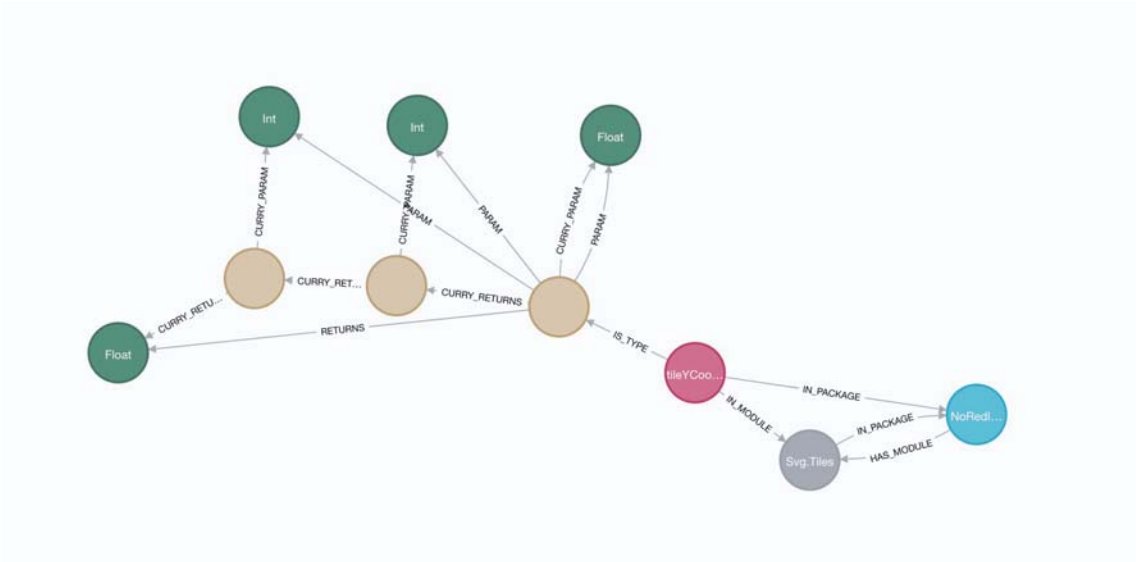

Figure 4.4.5: Function node structure example in Neo4j

where the root node of the function type connects to all of its intermediate parameters and the final mapping type. In relation "PARAM" there is also a property "order" which numbers the index of each argument. The intention of applying this change is that the search depth can be reduced to 1 instead of searching multiple levels. Also, applying this change allows us to implement the "reordered" search feature that has already been supported by Hoogle. The schema of a function node is shown in Table 4.6.

6. **Records**: Records are key-type pairs that holds a individual type signature. The key is a string which plays the role of indexing a inclusive type signature. Figure 4.4.6 demonstrates the internal storage structure of a record and Table 4.7 shows the property of root node.

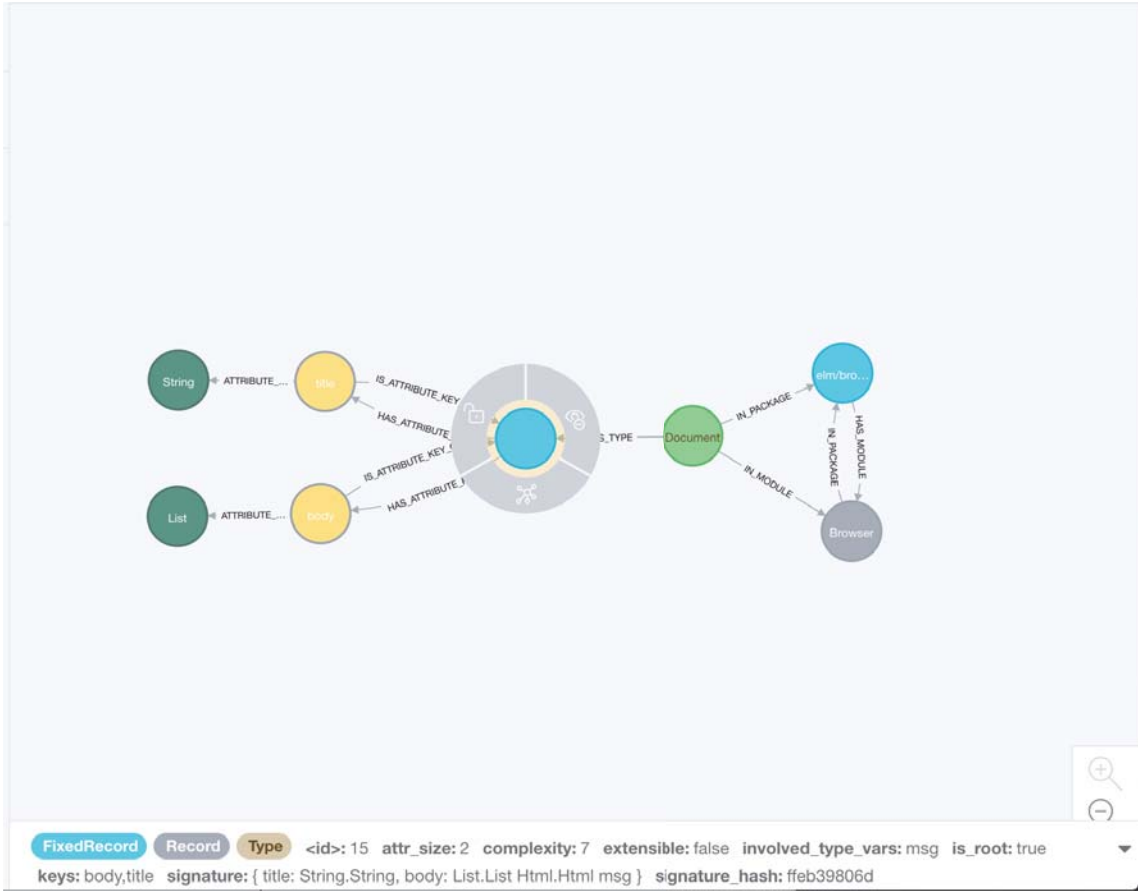

Figure 4.4.6: Record structure example in Neo4j

Table 4.7: Node schema for records

Property Name	**Type**	**Description**
<id>	Integer	Automatically generated identifier for each node.
is_root	Boolean	Indicates whether this type node is directly connected by a searchable API.
signature	String	Type signature of the current node.
complexity	Integer	Type size measurement of the record.
attr_size	Integer	Number of attributes(key-type pairs) inside this record.
keys	[String]	An array which stores the string value of keys.

4.4.3 Summary for Database design

In this section, the documentation has briefly introduced Neo4j database, and the reasons that we adopted it as the persistence layer of the architecture. AST nodes and their hierarchical relationships can be easily modeled as nodes and relationships in Neo4j, which establish the foundation for our further steps that grants Moogle with required functionalities.

In next section, we will introduce the procedure of handling a search request, where Neo4j will play a crucial role in filtering out APIs that do not conform to the user's requirements and provides nontrivial optimization on Moogle's performance.

4.5 Cypher generation

In the previous sections, we have implemented the MoogleQL parser and database models for Moogle. Also Subsection 3.2.3 brings out type unification algorithm, which can be used for the implementation of filter. Yet, there are several issues that worth attention:

- Type unification is, relatively speaking, time-consuming. [ACF+91] points out that Robinson's algorithm has an exponential time complexity based on the size of terms. And by applying optimization like *substitution delaying*, the average cost can be reduced to *at least linear*.

- Another problem is that the unification algorithm is *accurate*. It cannot proceed any fuzzy case (for instance, reordered match) nor tell is a type suitable for displaying in the result list.

Based on above reasons, it is necessary to apply pruning approaches to set up a *candidate set*, where API types are preselected from the database, before applying the unification checker one-by-one. As we mentioned in the previous section, Neo4j is a graph database that provides optimized queries on graph nodes and relations. Its query language, Cypher, allows developers to describe matchings with path-based syntax, where the constraints are properties of elements or relationships of nodes.

The procedure of handling queries can be separated into steps as Figure 4.5.1 shows. In the next subsections, the documentation will elaborate on the steps that lead a MoogleQL query to the final result set.

In most Web applications, the origination of data is the state layer which commonly refers to the database component. However, we seldom let our user write the query DSL like SQL directly, instead, we generate these queries through an intermediate representation like JSON requests. Another benefit part of this is that there are many third-party driver libraries for development languages. These drivers provides data access objects (DAO) which automatically maps data in the persistence layer to state objects so that a high-level abstraction is made.

Although MoogleQL is not loosely formed, it still provides usability to our target users, since it is closed to the real syntax in Elm. But MoogleQL is not a database query, since Neo4j adopts Cypher as its primary query language.

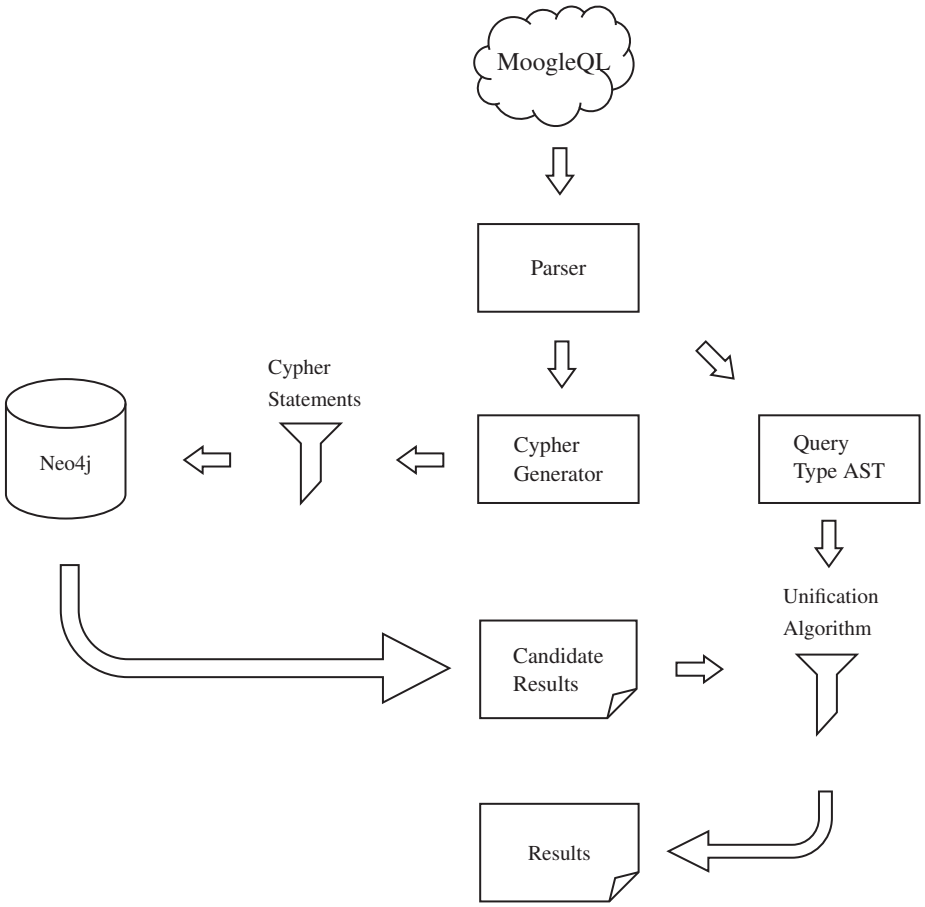

Figure 4.5.1: Filter pipeline

```
MATCH (n:Searchable { name: "map" }),
(n)-[:IN_MODULE]->(m:Module),
(n)-[:IN_PACKAGE]->(p:Package)
RETURN n as node, m as module, p as package
```

Figure 4.5.2: Cypher query example for name “map”

Cypher is a graph-based query language whose syntax is intimately similar to the graph structure. For instance, if we look for a searchable node named “map” and we also wants to know its module and packages, the Cypher statement could be written as Figure 4.5.2 shows.

But when the user is querying for a type signature, everything becomes complex, as there are multiple possibilities when type variables take part. For instance, when the user looks for APIs with type `(Int -> Float) -> List Int -> List Float`, it is not rigorous to map only the consistent type, since the API that the user need might be `(a -> b) -> List a -> List b`, where concrete types are the substitutions to type variables. From Subsection 3.2.3 we know that by applying unification algorithm we can always do a precise match on the two different types above. However, the target of generating Cypher is to **minimize** the range of candidate signatures so that the cost of applying unification algorithm could be reduced. In order to solve this problem, the best approach is to follow the idea of divide-and-conquer, and solve different cases by enumerating.

4.5.1 Generate Cypher for type variables

We start with another query `a -> b -> a`, where the first argument and return has the same type. In this case, we are looking for a *root* type node with following constraints:

- The type should be a function with at least 2 arguments.
- The first argument of above function, and curried return of the second argument should have the same type signature **if both of them are instantiated types**.

Note that the second constraint only affects instantiated types. The reason for this is that a type variable can be substitute by another type variable when applying unification algorithm. For example, `Int -> c -> c` and `d -> e -> f` would also unifies with `a -> b -> a` although the first argument and return has different representations.

Based on above constraints, we define Cypher generation rule as Figure 4.5.3 shows, where `(t1)` and `(t2)` refers to two type variables with same name that occurs in the type part of the query. When there are n occurrences t_0, t_1, t_2 ... t_{n-1} for a single type variable `a`, we add $n-1$ constraints rules for pairs from (t_0, t_1) to (t_0, t_{n-1}). A type variable with only a single occurrence would not add any constraint in `WHERE` clause since there is no further evidence we could extract.

```
MATCH (n),
...other constraints
(t1:Type),
(t2:Type),
...
WHERE
...
(
(NOT t1:Var) AND (NOT t2:Var)
AND(t1.signature = t2.signature)
) OR (t1:Var) OR (t2:Var)
...
RETURN n as node, m as module, p as package
```

Figure 4.5.3: Inserted Cypher queries for two type variables

4.5.2 Generate Cypher for Applies

Applies are a collection of types that instantiated from a union type constructor. Considering type query `Float -> Int -> Int -> Float`, which type signatures could it unifies? The match set could be extensive. For instance: (1) `a -> Int -> Int -> a`; (2) `a -> b -> b -> a`; (3) `a -> b -> c -> d` all fulfills the requirements. But `a -> a -> a -> a` or `a -> a -> b -> b` would not unify with the query type since the constraints on parameters or return are conflict. We conclude the following constraints for the case above:

- The type should be a function with at least 4 arguments.
- Types for the first argument and the return should be an Apply `Float` with 0 arguments once it is an not a type variable.
- Types for the second and third argument should be an Apply `Int` with 0 arguments once it is not a type variable.
- Once it is a type variable, the type signature of the first argument should be differ from the second and third one.
- Once it is a type variable, the type signature of the return should be differ from the second and third one.
- Once it is a type variable, the type signature of the second argument should be differ from the first argument and the return.
- Once it is a type variable, the type signature of the third argument should be differ from the first argument and the return.

Based on above constraints, we define Cypher generation rule as Figure 4.5.4 shows. The principle behind this is that we group all the Apply nodes in a query into different collections based on their names. In the case above there are two sets: $\{t|t.name =$ "Int"$\} = \{t_1, t_4\}$ and $\{t|t.name =$ "Float"$\} = \{t_2, t_3\}$. If there are multiple apply name sets derived from the query, we add Cypher constraint clause for each node pair (t_i, t_j), where (t_i, t_j) comes from the Cartesian product between each combination of two sets $S_m \times S_n$. An optimization approach is to remove duplicated pairs after the product, since there's no meaning for applying `ti.signature <> tj.signature` and `tj.signature <> ti.signature` at the same time in the `WHERE` clause.

```
MATCH (n)-[:IS_TYPE]->(f),
...other constraints
(t1:Type),
(t2:Type),
(t3:Type),
(t4:Type),
(f)-[:PARAM {order: 0}]->(t1),
(f)-[:PARAM {order: 1}]->(t2),
(f)-[:PARAM {order: 2}]->(t3),
(t3)-[:CURRY_RETURNS]->(t4),
...
WHERE
...
(
((t1:Apply AND t1.name = "Float") OR (t1:Var))
AND ((t2:Apply AND t2.name = "Int") OR (t2:Var))
AND ((t3:Apply AND t3.name = "Int") OR (t3:Var))
AND ((t4:Apply AND t4.name = "Float") OR (t4:Var))
AND (t1.signature <> t2.signature)
AND (t1.signature <> t3.signature)
AND (t2.signature <> t4.signature)
AND (t3.signature <> t4.signature)
...
RETURN n as node, m as module, p as package
```

Figure 4.5.4: Inserted Cypher queries for Applies

4.5.3 Generate Cypher for Records

Record is a type which includes a group of key-type pairs (k, t) with fixed length. But Elm made an extension for this type called extensible records, which, in some way, partially make up the previously missing language feature caused by lacking type-classes. Extensive records are similar to interfaces in other languages. For instance, extensive record R_x can unify with a record R when fulfilling the following constraints, where $K(R)$ denotes the key set of R, and $T(R.k)$ refers to the mapped type of key k in R:

- R is a fixed or extensive record type.
- When R is a fixed record, $K(R_x) \subseteq K(R)$ is true.
- When R is extensive, $(K(R_x) \subseteq K(R)) \vee (K(R) \subseteq K(R_x))$ is true.
- For any key k exists in both $K(R)$ and $K(R_x)$, $T(R.k)$ is unifiable with $T(R_x.k)$

For instance, `{x | value: String, position: a }` unifies with following types: (1) `{value: String, position: Compatible }`; (2) `{value: a, position: b }`; (3) `{y | value: String, position: Compatible, overflow: Compatible }`;

Although conforming to the final constraint requires unification algorithm, it is still possible to apply the first two constraints in advance by generating clauses in Cypher, we have already added a property "keys" in each record node. The property is an array which includes all the key strings that occurs in the signature of that record. And by applying `(ALL ({keyname} IN {record}.keys WHERE {keyname} IN {keyset}))` in the `WHERE` clauses we can implement the filter generating candidate records.

4.6 Ranking the results

In the previous sections we discussed the procedure of generating Cypher queries. In this section, we focus on ranking the results, which is a vital factor for usability.

Although candidate set has been scratched from Neo4j database, the rankings of its items can be a non-trivial issue which worth further consideration. The thesis has mentioned several metrics in Subsection 3.1.2 and Section 3.3 in order to solve the problem.

The solution is based on a combination of names and types, which can be briefly described as following steps:

1. If name-related constraints appear in MoogleQL, Moogle will leverage Levenshtein distance as the first-priority ranking metric. The name with smaller Levenshtein distance to the query name would have a higher ranking. Notice that when the name part in MoogleQL is null, we will skip this step.

2. Then, Moogle looks into type constraints. The size of types would be used as the prioritized metric in this step. If an item with type t_i has a closer type size to the query type t_q than another item with type t_j (can be denoted as $|size(t_q) - size(t_q)| < |size(t_j) - size(t_q)|$), then type t_i has a higher ranking. For instance, the user searches for type `Int -> a -> a` and its has a type size of 5. Then in the results, `and : Int -> Int -> Int` will have a higher ranking since its type size is also 5 and the distance to the query is 0, while `take: Int -> List a -> List a` will be put behind since its type size is 7 and the distance between will be 2.

3. If t_i and t_j has the same type size, then we compare their variable diversity to that of the query type as the distance. For instance, if we query with a type signature `a -> a`, then API `identity: a -> a` would have a higher ranking than `complement: Int -> Int` since the variable diversity distance between query and `identity` is 0, while 1 for the latter API.

4. Other metrics will be adopted once all the steps above cannot help, which includes: the alphabetic order of API name, module name, package name and whether the the API belongs to the core package (build-in APIs).

Chapter 5

Results and benchmarks

In this chapter, we demonstrate the work result and provide benchmarks on our search engines. Since Moogle is a type-based API search engine, each item of results will be evaluated independently and the unification algorithm is a deterministic process that only outputs **true** or **false** as the validation indicator. As Moogle guarantees the validity of results, thus our tests will focus on hit numbers, precision on the top k items, and response time as its performance indicators.

5.1 User interface demonstration

After the design phase and subsequent development, we complete the project source code including its user interface. The UI of Moogle is similar to Elm-search, with a single input box as its interactive element, where MoogleQL is entered and send to the REST API of the server. Figure 5.1.1 demonstrates UI of Moogle, where results are displayed in each row of a table with API descriptions, module and package names. Above of the list shows how many items are matched via applying unification algorithm.

In some cases, Moogle may not find enough unifiable cases, thus it will also returns those candidate APIs which does not make a full match to user's query. A typical case is shown in Figure 5.3.1 where the functionality of reordered match is required.

5.2 Prerequisites and constraints

5.2.1 Test environment

Beginning with this subsection, all the tests and benchmarks will commence in the following hardware and software environment.

Computer Model MacBook Pro (15-inch, 2017)

CPU 2.9 GHz Intel Core i7

Memory 16 GB 2133 MHz LPDDR3

External Storage 500 GB Flash Storage

Operating System macOS Mojave version 10.14.5

Software Requisitions Python 3.7.0, Nginx 1.15.12, Neo4j 3.5.5, Docker 2.0.0.3

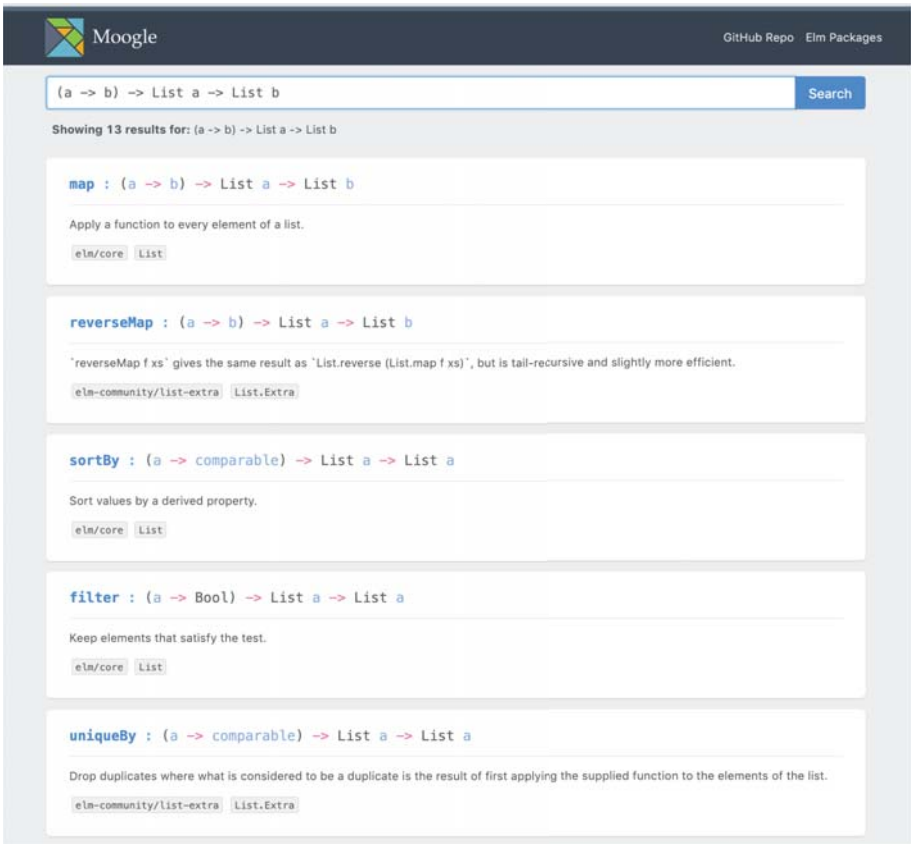

Figure 5.1.1: User Interface example of Moogle

5.2.2 Sample queries

The following list presents some typical sample queries and their intention or usage scenarios, which will be used as the testing inputs:

Q1. `length`
Intention: The user wants to search for APIs named "length" that inputs an "iterable" parameter and returns its size. This case is derived from use case 1 in Section 1.2. We list part of our expected APIs as below:

- `length : List a -> Int`
- `length : String -> Int`

Q2. `map`
Intention: The user wants to search for APIs named "map", which accepts an iterable parameter then apply a function to each of its members. This case is derived from use case 1 in Section 1.2. We list part of our expected APIs as below:

- `map : (a -> b) -> List a -> List b`
- `map : (a -> b) -> Array a -> Array b`

Q3. `fold`
Intention: The user wants to search for APIs whose name contains "fold" since he or she cannot remember the precise name of that API. This case is derived from use case 2 in Section 1.2. We list part of our expected APIs as below:

- `foldl : (a -> b -> b) -> b -> List a -> b`
- `foldr : (a -> b -> b) -> b -> List a -> b`

Q4. `: Int`
Intention: The user wants to search for APIs whose type is `Int`. Note that there is a colon in front of the type signature, which annotates that this is a type search instead of a name search. This case is derived from use case 3 in Section 1.2. In Elm-search the corresponding query should be "Int". We list part of our expected APIs as below:

- `maxInt : Int`
- `minInt : Int`

Q5. `: Maybe a`
Intention: The user wants to search for APIs whose type is `Maybe a`. Note that there is a colon in front of the type signature, which annotates that this is a type search instead of a name search. In Elm-search the corresponding query should be "Maybe a". This case is derived from use case 3 in Section 1.2. We list part of our expected APIs as below:

- `Nothing : Maybe a`

Q6. `Just`
Intention: The user wants to search for a type constructor named "Just". This case is derived from use case 4 in Section 2.3. We list part of our expected APIs as below:

- `Just : a -> Maybe a`

Q7. `Left`
Intention: The user wants to search for a type constructor named "Left". This case is derived from use case 5 in Section 2.3. We list part of our expected APIs as below:

- `Left : a -> Either a b`

Q8. `List a -> Int -> List a`
Intention: The user wants to search for APIs that return (or remove) the first/last n elements of a list. Note that in this case, the user mistakenly puts `Int` into the second argument. This case is derived from use case 6 in Section 2.3. We list part of our expected APIs as below:

- `take : Int -> List a -> List a`
- `drop : Int -> List a -> List a`

Q9. `(Int -> Float) -> List Int -> List Float`
Intention: The user wants to search for APIs which can transform a list of integers to float. But the best solution would be a generic function like "map". This case is derived from use case 7 in Section 2.3. We list part of our expected APIs as below:

- `(a -> b) -> List a -> List b`

Q10. `(a -> b) -> f a -> f b`

Intention: The user wants to search for an API that maps a type `a` to type `b` to each member of an iterable parameter. But this time, he or she cannot recall the name of that constructor. This case is derived from use case 8 in Section 2.3. We list part of our expected APIs as below:

- `map : (a -> b) -> List a -> List b`
- `map : (a -> b) -> Array a -> Array b`
- `map : (a -> b) -> Set a -> Set b`

Q11. `+core concat`
Intention: The user wants to search for APIs whose name contains "concat", but only limited to the core packages. This case is derived from use case 9 in Section 2.3. We list part of our expected APIs as below:

- `concat : List (List a) -> List a`
- `concat : List String -> String`

Q12. `Random.Pcg.Extended.Generator a`
Intention: The user wants to know if there is any API that has type `Generator a`, while the module of `Generator` is "Random.Pcg.Extended". This case is derived from use case 10 in Section 2.3.

Q13. `Int -> Int -> Int -> (Int, Int, Int)`
Intention: The user wants to combine three integers into a triple. This case is derived from use case 11 in Section 2.3. We list part of our expected APIs as below:

- `join : a -> b -> c -> ( a, b, c )`
- `tuple3 : a -> b -> c -> ( a, b, c )`

Q14. `{ x | onChange: a }`
Intention: The user queries for records which contains key "onChange". This case is derived from use case 12 in Section 2.3.

5.2.3 Other constraints

In consideration of the performance and test validity. For each query, Moogle sets 1000 as the maximum number of returning items.

5.3 Hit tests

In the previous section, the documentation listed several sample queries. But before commencing the benchmark, we introduce another metric, *precision@k*, for measuring the quality of results. In this test, precision@k is defined as Equation 5.1.

$$precision@k(results) = \frac{\text{number of expected results in the first } k \text{ items}}{min(k, \text{total number of items})} \tag{5.1}$$

The term "expected results" stands for the result items which meet the requirements according to our description in the corresponding use case. For instance, `classes :`

`List Color -> Int -> List Color` is an API that unifiable with our query type, but it is not the expected one (a "noise"). On the other hand, `take : Int -> List a -> List a` meets user's requirement (a "signal"). This indicator could effectively reflects the *Signal-Noise Ratio (SNR)* or *information entropy* of Moogle's returning results. The higher this figure is, the more "useful" information would our user gain from the result list, given the identical number of items.

To ensure the validity of test, we setup $k = 10$ and manually mark up each items as "useful"(expected) or not, since there is not an automatic approach for handling semantic information in the requirement of each scenario.

We initiated the test and compared the result with Elm-search if the case is compatible. For each case, we have also taken down the ranking of the first "useful" result, which represents the expected API with highest rank, as a baseline for reference. This figure may indicate "how fast" can our user retrieve his or her wanted APIs (or precisely, the efficiency of obtaining information, which also reflects the usability of product). Unifiable candidate refers to a result whose signatures can unify with the query type, while a non-unifiable candidate stands for a result that has been selected from the database but failed to pass the unification check. Table 5.1 demonstrates the statistical figures for each sample query.

Table 5.1: Hit test results and comparison

Query No.	Total Matches	Unifiable Candidates	Non-Unifiable Candidates	Highest Ranking of Expected Items	Precision@10 (Percentage %)	Elm-search Compatible	Number of matches (Elm-search)	Highest Ranking of Expected Items (Elm-search)	Elm-search Precision@10 (Percentage %)
1	241	241	0	1	100	Yes	45	1	100
2	768	768	0	3	50	Yes	376	1	80
3	187	187	0	5	60	Yes	98	2	90
4	45	45	0	1	20	Yes	25	1	100
5	1	1	0	1	100	Yes	0	N/A	N/A
6	194	194	0	1	10	No	N/A	N/A	N/A
7	559	559	0	2	10	No	N/A	N/A	N/A
8	59	2	57	4	30	Yes	24	1	30
9	24	2	22	1	20	Yes	0	N/A	N/A
10	249	249	0	1	100	No	N/A	N/A	N/A
11	3	3	0	1	100	No	N/A	N/A	N/A
12	251	4	247	1	40	Yes	1	N/A	0
13	3	3	0	1	100	No	N/A	N/A	N/A
14	23	23	0	1	100	No	N/A	N/A	N/A

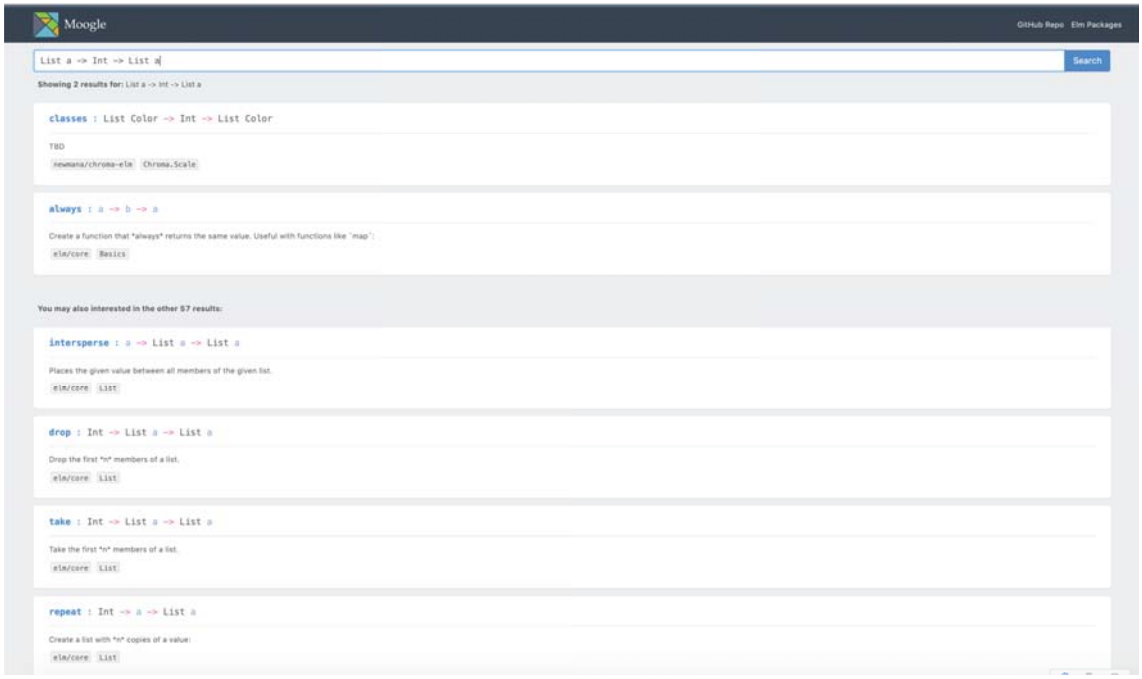

Figure 5.3.1: Reordered matching example in Moogle

What can be clearly seen in Table 5.1 is that the figure of total matches for every case, Moogle provides more results than Elm-search. In case 5, although unifiable result set only covers 2 items, Moogle is still able to present the desired APIs like “drop” and “take” at the head of non-unifiable candidates (as Figure 5.3.1 shows), even the parameters are reordered. Hence, in this case, Moogle still fulfills the requirement that we mentioned in Section 2.3.

Overall speaking based on the figures in Table 5.1, the result set provided by Moogle, in all example cases as we observed, outperforms the list that we can retrieve by Elm-search on both API coverage and result rankings.

5.4 Response time tests

The response time is one of the most critical factor that measures the performance of a software. In this test, we use the same dataset from the previous section as the test cases for response time. For each query, we also record its query type size and type variable diversity, which may help to find out potential factors on response time. The result of the test is shown in Table 5.2, where “average response time” refers to the mean time among 5 benchmarks on the identical test case.

As Table 5.2 indicates, a potential trend is that when the query type size and type variable diversity grows, the handling procedure of Moogle is prone to take more time in average, except that additional constraints like records (Q9) and package (Q7) are given.

Table 5.2: Response time test results

Query No.	Total matches	Unifiable Candidates	Non-unifiable Candidates	Query type size	Query type variable diversity	Average response time (seconds)
1	241	241	0	0	0	0.378
2	768	768	0	0	0	0.460
3	187	187	0	0	0	0.341
4	45	45	0	1	0	0.350
5	1	1	0	1	1	0.239
6	194	194	0	1	0	0.359
7	559	559	0	1	0	0.436
8	59	2	57	7	1	1.110
9	24	2	22	9	0	1.940
10	249	249	0	9	3	4.060
11	3	3	0	0	0	0.031
12	251	4	247	2	1	0.016
13	3	3	0	10	0	2.380
14	23	23	0	3	1	0.231

Chapter 6

Conclusion and Future work

In this final chapter, we conclude our work with discoveries, achievements and problems worth further considerations. We will also look to similar works that done by others and summarize the possible future work.

6.1 Conclusion

In this thesis, we have presented Moogle, a type-based search engine that supports API retrieving service for Elm. From this work, we have discoveries on many aspects including solutions, algorithms, methodologies and even with some lessons. In the following content we will depict these items in details, which might be helpful for the proceeding study.

6.1.1 Discoveries

1. **Type is useful when semantic(name) information cannot help.** An API sometimes can be ignored when a user tries to look up for a conceptual name. For instance, a JavaScript developer may search for "reduce" instead of "foldl" when he or she want to apply the corresponding function due to empirical knowledge. With type-based search engines, instead of searching for a name, the developer may search by parameter and return type which is a more accurate description for the API, since the signature describes mapping relations instead of semantic functionalities.

2. **Unification algorithm is powerful yet time consuming.** Unification algorithm is a powerful and flawless solution for type matching problems when type variables involve. However, its time complexity becomes a problem when the we put it into utilization. A potential optimization solution is to cache more information in memory so that traversing each pair of types could be faster, or improve the algorithm itself.

3. **Graph database is a feasible solution for indexing.** Different from related work like Hoogle or Elm-search, Moogle does not build indexes by itself. Instead, it uses graph database, Neo4j, as its persistence layer and API data source. Based on our observation on the test results and response time, it is convinced to say graph database such as Neo4j, can be utilized in type-based code search, which creates indexes for AST automatically and boosts the maintainability for the software.

4. **Metrics are vital.** According to our observation during the study, we found that ranking metrics play an important role in search engines, which not only affects the results but also on the design, a typical evidence of which is the data model we stored in the Neo4j database is strongly reflecting by the metrics that we defined. Instead of storing only straightforward AST structure, we have also added many extra properties and relations for type nodes, which could be useful since they cache the measures which can be later used in our ranking metrics. Also, a carefully designed metric usually outperforms the ambiguity ones on both usability(rankings) and performance.

6.1.2 Achievements

According to the tests against several use cases and comparing to the related softwares that we mentioned in Chapter 2, we conclude the features and improvements that we have achieved in this work.

1. **Moogle covers a larger range of results** . Hoogle and Elm-search has several flaws, according to our description in Chapter 2, which limits the searching range among the APIs. Moogle, however, covers larger range of results without significant loss on the performance thanks to leveraging Neo4j and a carefully-designed data model.

2. **A query system with its own DSL (MoogleQL)** Graph database has been used in many research scopes but few among them involves static analysis on source code. A recent work [UM15] adopts a similar approach to build up a query system for source code and it also uses Neo4j as its persistence layer. Nevertheless, the query system was built for Java, which is not a FP language and it did not develop a search engine for practical usage. Our work, on the other hand, designed a query system with its own DSL which has been put into actual uses in our search engine, Moogle.

6.1.3 Limitations

Although Moogle managed to implement many features, still there are several limitations observed and we list these issues as below for further study:

1. **Limited performance on large type size** : It is observed that when the size of type in MoogleQL grows the performance of Moogle goes down. A potential reason is that as the constraints in generated Cypher increases, the execution pipeline in Neo4j would grow to an inefficient level, where redundant records would cost much time.

2. **Limited abstraction capacity on functions** : Moogle is able to abstract initialized types into generic ones. Nevertheless, we observed that Moogle is not be able to convert a function parameter into a type variable. For instance, query type `(a -> b) -> (a -> b)` can unify with `identity: a -> a`, but Moogle may not be able to list the latter API in its result list. The root cause of this effect is that when parsing MoogleQL, we did not consider the cases that functions of a query might be substitute by a type variable, which should be fixed in future.

6.2 Future work

In this thesis, we design and implemented Moogle, the type-based API search engine which leverage the advantage of combining name and type information as the indexes of APIs. Based on this work, we advocate that there are several proceeding research directions that worth considering:

1. **Extend design for other FP languages.** Moogle is designed for Elm, which is a functional programming language that designed for Web development. However, other FP languages like Haskell has a stronger type system which includes typeclasses. A typeclass system can be another challenge when we are searching for a concrete type constructor with specific typeclass or vice versa.

2. **Utilize Neo4j database for indexing other typed languages.** In this work, we take the advantage of utilizing Neo4j for constructing the database models. However we also have noticed other work like [UM15], which uses Neo4j to build a query system for Java. Hence, we propose that it is promising to develop an API search engine by leveraging a graph database for other typed languages that even not belongs to a FP language.

3. **Optimize performance.** As we mentioned in the previous section, the performance of Moogle is acceptable but further optimization is required, since Moogle takes longer time on searching for a large-sized query pattern. The starting points could be optimizing the Cypher generation module, which may outputs a low-efficient query with a complex query. Another possible approach is that to leverage the scalability of Neo4j, which could be a possible solution for the performance issue.

Bibliography

[ACF+91] Luc Albert, Rafael Casas, François Fages, A. Torrecillas, and Paul Zimmermann. Average case analysis of unification algorithms. In Christian Choffrut and Matthias Jantzen, editors, *STACS 91*, pages 196–213, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[App04] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.

[BCK03] L Bass, P Clements, and R Kazman. Software architecture in practice. 2nd addison-wesley. *Reading, MA*, 2003.

[Bea19] David Beazley. Ply (python lex-yacc). https://www.dabeaz.com/ply/, 2019.

[Ben18] Eli Bendersky. Lecture 22: Type inference and unification. https://eli.thegreenplace.net/2018/unification, November 2018.

[Coc19] Jonas Coch. Elm-search. https://klaftertief.github.io/elm-search/, May 2019.

[Fie00] Roy Fielding. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, pages 76–85, 2000.

[flu19] Flux | application architecture for building user interfaces. https://facebook.github.io/flux/, April 2019.

[HFP92] Paul Hudak, Joseph H Fasel, and John Peterson. A gentle introduction to haskell. *Sigplan Notices*, 27(5):T1–T53, 1992.

[Hil19] Paul N. Hilfinger. Lecture 22: Type inference and unification. https://inst.eecs.berkeley.edu/~cs164/sp11/lectures/lecture22.pdf, 2019.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[Mit19] Neil Mitchell. Hoogle. https://hoogle.haskell.org/, 2019.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[Muk04] Madhavan Mukund. Lecture 22: Type inference and unification. https://www.cmi.ac.in/~madhavan/courses/pl2009/lecturenotes/lecturenotes/node85.html, April 2004.

[Nor91] Peter Norvig. Correcting a widespread error in unification algorithms. *Software: Practice and Experience*, 21(2):231–233, 1991.

[R+65] John Alan Robinson et al. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[UM15] Raoul-Gabriel Urma and Alan Mycroft. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming*, 97:127–134, 2015.

[V.18] Nachiappan V. Using yacc. https://silcnitc.github.io/yacc.html, April 2018.

[VB14] Rik Van Bruggen. *Learning Neo4j*. Packt Publishing Ltd, 2014.