Scaps*: Type-directed API Search for Scala

Author
Lukas Wegmann

Supervisor
Prof. Peter Sommerlad

Technical Adviser
Mirko Stocker

University of Applied Science Rapperswil

Master's Thesis, August 20, 2015

*scala-search.org

Abstract

Reusing existing functionality from legacy code and third party programming libraries is often hard, because the vast number of definitions and the complexity of the APIs require a detailed knowledge of the programming environment. To help developers discovering such functionality when working with a statically typed, object-oriented programming language, we designed an API retrieval system that supports querying indexed APIs with textual keywords and type signatures. The feature of the search engine is inspired by a similar tool for the Haskell programming language called Hoogle. Though, we decided to develop another approach to address the API search problem based on traditional information retrieval techniques which, we argue, is better suited for object-oriented languages.

As a proof of concept, we implemented the API retrieval system for the Scala programming language and provided a web-based user interface. Furthermore, we demonstrate how Scala-specific language features like user-defined implicit conversions and implicit parameters can be included into our basic approach. A comparison to a system using only exact matches of partial types of the query shows that the effectiveness of API retrieval can be substantially improved with our approach.

Management Summary

This thesis describes the design of a type directed search engine for statically-typed, object-oriented programming languages. Furthermore, we provide an implementation for the Scala programming language that incorporates Scala specific language constructs.

Status Quo

A crucial part of creating high quality software projects is the reuse of existing functionality provided by in-house or third party programming libraries. This ensures that functionality is not unnecessarily reimplemented and lowers the risk of introducing erroneous behavior. Code reused over various projects tends to be well-tested and battle-proofed.

Discovering existing functionality is a task that requires either a deep knowledge of the according libraries or appropriate tools that provide access convenient to the definitions in a library. Some examples of such tools are code completion assistants that list the accessible members of an object in question and automatically generated documentation pages for the available classes. Though, these tools tend to give an incomplete picture of the available operations, because functionality provided through utility classes or by additional libraries are usually not listed.

In this case, developers often resort to universal search engines like Google to find a specific implementation. While these search engines often provide suitable results, users have to manually mine the result pages for suitable content. Additionally, the result pages may refer to outdated revisions of a library.

In the functional programming community, a popular tools to overcome these issues are search engines, like Hoogle for Haskell, that accept type signatures as search queries and retrieve definitions of a similar type. In combination with a powerful type system, type queries can efficiently and precisely describe the functionality a user is looking for.

Unfortunately, the algorithms used to query functional programming libraries cannot simply be adapted to object-oriented programming languages. Especially subtype polymorphism, which is used more frequently in object-oriented programming, is a major obstacle.

Goals

The goal of this thesis is to develop a search engine for retrieving definitions in Application Programming Interfaces (API). The search engine should accept queries that consists of zero or more full-text keywords and a type signature and return an ordered list of identifiers that match the query. The system should be able to answer queries that

do not exactly match the available definitions. For example, a user queries for a method that accepts a list of string and returns the first element. But the according functionality may be provided by a method accepting a sequence of elements of an arbitrary type. To retrieve type signatures different from the query but with similar semantics, the retrieval system has to incorporate concepts used in the type system of modern, object-oriented languages. Namely, parametric and subtype polymorphism. Furthermore, we want to index several popular libraries with a total of more than 100'000 definitions and still provide results in less than one second.

To show the suitability of our approach, we implement the search engine for the Scala programming language. This implementation should demonstrate that language specific features can be incorporated into the basic algorithm and that the system is effective to discover available functionality.

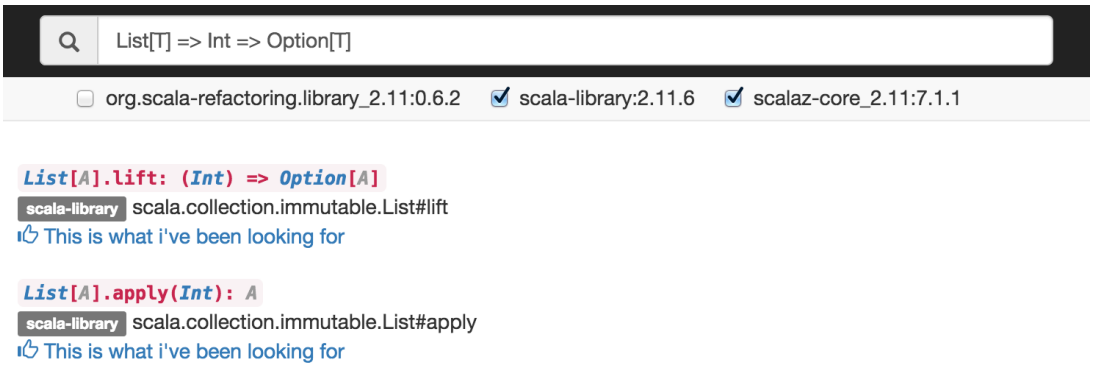

Figure 0.1.: Screenshot of the Scaps web client with the query input form, library filters and two search results

Results

We developed an API retrieval model based on the vector space model and integrated it into the Lucene search engine. This allows us to leverage existing optimizations like inverted indexes and in-memory caching. The search engine is managed by a web service that exposes various operations to control the index over an HTTP API. Additionally, the web service provides a user interface as shown in Figure 0.1 to query the index.

Furthermore, to evaluate our retrieval model, we created a test collection with more than 50 test queries covering three Scala libraries (including the Scala Standard Library). Each test query is associated with one or more expected relevant result. This collection has been used to compare the effectiveness of various instantiations of our retrieval model and to verify some assumptions made during the design of the system.

Because there is currently no similar search engine available for the Scala language, we created a baseline system that roughly corresponds to a search engine using only textual matches to retrieve type signatures. The main result of comparing the baseline

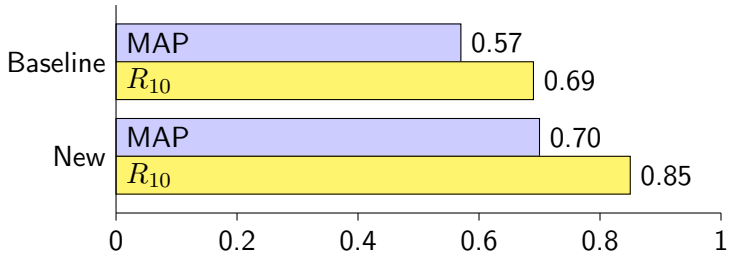

Figure 0.2.: Performance comparison of the baseline system and our retrieval model

system to our approach is given in Figure 0.2. The *Mean Average Precision* (MAP) values show that the overall precision of the search engine over the complete result set can be significantly improved. Perhaps more meaningful for users of the search engine is the *Recall at 10* (R_{10}) value: While the baseline system retrieved only 69% of the relevant results on the first result page (top ten results), our approach increased this value to 85%.

Declaration of Authorship

I, Lukas Wegmann, declare that this thesis and the work presented in it is my own, original work. All the sources I consulted and cited are clearly attributed. I have acknowledged all main sources of help.

Rapperswil, August 20, 2015

Lukas Wegmann

Contents

1. Introduction

This thesis introduces a new approach to retrieve entities from programming APIs. The resulting search engine accepts a query consisting of textual keywords and a type and returns a ranked list of identifiers of matching functions and values. The search engine is capable of indexing several libraries with more than hundred thousand of definitions and returning a result set in less than a second with a recall of > 0.85 in the top ten results. Furthermore, the approach embraces both parametric and subtype polymorphism which makes it applicable to many modern statically-typed, object-oriented programming languages.

This chapter describes the framing conditions of this thesis and states our motivation behind creating yet another search engine for programming libraries.

1.1. Problem

Working with complex APIs and big projects requires a deep knowledge of available functions, classes, methods and other artifacts. Writing source code without this information often leads to duplicated functionality or even worse, to erroneous code for which already exists a well-tested implementation. This is why good and well-structured documentation is essential for introducing developers to the provided functionality of libraries, legacy code and frameworks. But even if an exhaustive documentation is present, developers often do not have the time or patience to read through it or would simply not expect that the feature they are looking for is already available.

Beside documentation, IDEs also try to help developers exploring libraries by providing tools like code completion assistants, type hierarchy overviews or text search with more or less sophisticated semantic support. Especially code completion that proposes methods applicable on a concrete object is extremely helpful because it is easy to invoke and object-oriented libraries often provide useful functions as class members. But code completion is of limited suitability when working with code that is written in a more functional style or when the functionality one is looking for is provided by a utility class.

Another approach to tackle this problem are search engines like Hoogle for Haskell [Mit] that allow to search for functions based on their type signature. This assumes, that developers usually know what types they have and of what type the result should be but don't know how to get there. The great number of questions of the form "How to create X from Y" on popular Q&A websites seems to support this assumption (see Appendix C).

1.2. Vision

The long-term vision behind this project is to create a type based search engine that is seamlessly integrated into a developers workflow and becomes such an indispensable tool like code completion assistants are today. The search should be easily invocable from an IDE and provide accurate results almost instantaneous.

One possible invocation method is to allow users to directly start the search from the editor window as illustrated in Figure 1.1. This sketch extends Eclipse's quick fix menu that provides several useful refactorings and code rewrite actions depending on the current context and is well-known by developers. For example, Scala IDE [Sca] provides a "create method" quick fix whenever the parser sees a method invocation whose identifier cannot be resolved to an existing symbol.

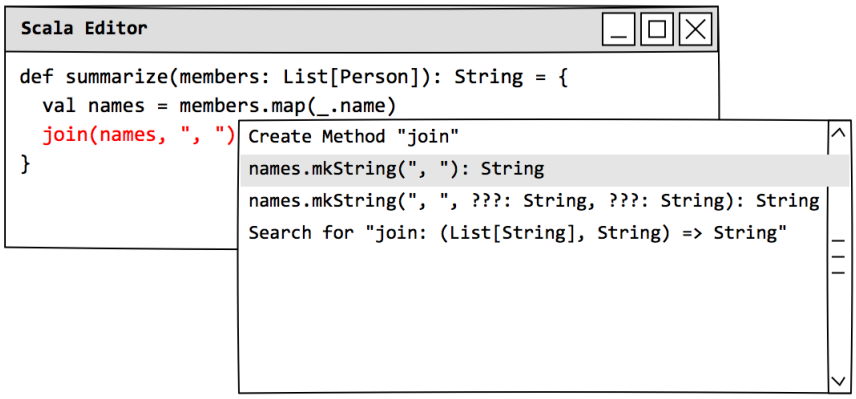

Figure 1.1.: The missing symbol `join` leads to a type error (marked red). The search engine extracts a query from the erroneous expression and proposes the search results in the quick fix menu.

Whenever the user writes an expression of the form `[identifier]` or `[identifier] ([arguments]*)` whereas `identifier` cannot be resolved to a symbol in scope, the search plug-in extracts a query from the erroneous expression. When the user opens the quick fix menu, the query is applied on the index and the best matching results are presented to the user along the other quick fixes (Figure 1.1). Furthermore, the quick fix menu contains an additional entry that allows to refine the search query and to fetch further results.

When the user chooses one of the proposed search results, the plug-in tries to replace the erroneous expression with the selected one as illustrated in Figure 1.2. If a result expects more parameters than supplied by the original expression, like, for instance, `names.mkString(", ", ???: String, ???: String)` in the paper prototype, the extra arguments are filled up with a placeholder expression `???`. These placeholder can then be replaced manually or used as a starting point for further queries.

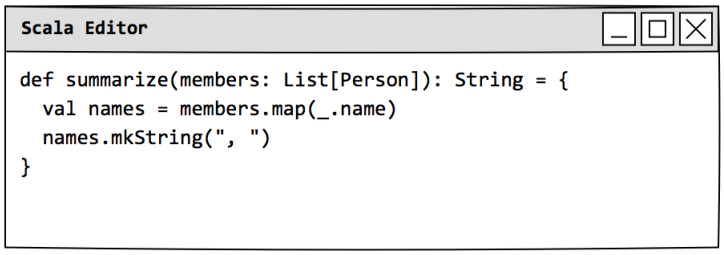

Figure 1.2.: After the user selected one of the search results, the plug-in replaces the erroneous expression with the proposed one.

1.3. Goal

The goal of this thesis is to develop a search engine that assists programmers finding available functionality in big software projects. The initial project objectives have been defined as following:

- Finding an approach to search programming APIs that meets the following requirements:
  - Identifiers, documentation comments and type signatures should be incorporated to fulfill a users information need.
  - The search should incorporate the semantics of type systems of modern object-oriented programming languages. This includes:
    - Subtype Polymorphism
    - Parametric Polymorphism
    - Implicit Type Conversions
  - Searching a corpus of 100'000 declarations should yield results in less than a second.
- Implementation of a proof-of-concept prototype of the aforementioned search engine for the Scala programming language.
- Integration of the prototype into the Scala IDE programming environment such that projects and library dependencies can be indexed.
- Providing an interface that accepts textual search queries.
- Integration of the search functionality into Scala IDE's quick fix menu.

Unfortunately, these objectives underestimated the complexity of the task. Developing a suitable search engine required more effort as expected because there was little preliminary work on this topic that fitted our requirements (see next section). This is why we decided during the first third of the project to drop the integration into Scala

IDE from the project schedule and to spend more time on improving the search engine. Instead, we have defined the following additional objectives:

- Integration of the prototype into a standalone web service.
- Providing a plug-in for the Scala Build Tool (SBT) that controls the web service and starts indexing jobs based on SBT build definition.
- Providing a web based user interface for invoking textual search queries.

Besides the reduced complexity compared to an integration into an IDE, a web based UI offers additional advantages: At one hand, it simplifies collecting queries entered by users. This usage data can then be used to improve the search engine based on empirical data. And, at the other hand, it is easier to promote a product that can be instantly tested by users without installing a plug-in.

1.4. Current Work and Motivating Examples

A developer looking for a certain functionality in a programming library has many tools available that will assist him. Broadly speaking, we can group these tools into the following categories:

General purpose search engines

Web search engines, like Google, are frequently used to solve programming problems. They are simple to use and personal experience shows that a good formulation of the problem as a search query often directs to one or more helpful websites. Especially for popular libraries, the web offers a vast repository of discussions on how to solve specific problems.

Besides web-based searches, querying local code repositories and documentation with text-based search tools may also be helpful. For example, the command line tool `grep` allows to search directory structures for files containing text matching a given regular expression. This can also be leveraged to match method signatures in source code.

Specialized search engines for retrieving definitions

The tool that comes closest to what we plan to develop is Hoogle for the Haskell programming language [Mit]. Hoogle accepts textual keywords and a type and yields identifiers of functions that match this query. Hoogle also matches definitions with a more generic type than the query. Another tool in this category is the "Java Search" functionality of Eclipse JDT [Ecl]. The "Java Search" tab supports queries for definitions with a certain pattern that may also contain types. Though, a definition has to precisely match the pattern in order to be retrieved. One attempt to provide a tool similar to Hoogle for Scala is Scalex [Dup]. Unfortunately, this project has been discontinued in 2013 and is not available for a recent version of

the Scala language. Furthermore, an early attempt to implement such a search engine for a dialect of the ML programming language by leveraging isomorphisms on types is given in [Rit90].

Specialized search engines for retrieving code snippets

Another approach is to help user find existing code snippets from various online resources. For example, Sniff for Java [CJS09] collects code snippets from open source libraries and intersects similar snippets and annotates them with documentation of the library definition. These snippets can then be retrieved by using a textual query. Similar tools are [WCGH15], which uses the Bing search engine to find snippets and re-ranks them, and Prospector [MXBK05], which mines repository for "jungloids" (typed code snippets) and synthesizes them to new expressions based on a typed user query.

Program synthesis tools

Besides proposing existing code snippets, some tools focus on synthesizing new statements based on user queries. Hence, based on definitions in a library a new sequence of expressions is synthesized that uses the input values from the query, executes the desired effect and returns a value of the expected type. Some recent work on this topic is Scala InSynth [GKKP13] and CodeHint [GRB+14].

Code completion assistant

A common feature of IDEs are code completion assistants that propose available operations on a value. IDEs for object-oriented programming languages usually propose class members of the object in question. Some code completion assistants can also filter the operations based on textual keywords or expected return types. Thus, invoking code completion usually means to query for an API definition with one input type and, optionally, an expected output type. Often, the specialized search engines mentioned above are also integrated into the code completion assistant, as this feature is well known by IDE users and easy to invoke.

This is a rather incomplete list of available tools and research papers but should give a raw overview on the general approaches how the problem can be addressed.

A concrete problem in Scala can be stated as "How to transform a list of integer values to a string containing the textual representation of all elements separated by a comma and a space?". Thus, a user is looking for a function of type `(List[Int], String) => String`. This functionality is provided in the Scala Standard Library as a member method `mkString` of `List`. A developer not yet familiar with the Scala environment, who does not know about this particular definition, may nevertheless suspect that something similar should already exist.

Another motivating example is a developer newly introduced to the code base of a customer relationship management (CRM) system. He performs a task on the UI code and wants to render an instance of an `Address` into an HTML document. While getting familiar with the CRM system, he observed that there is a common representation

of such objects. Thus, he suspects that there should be a function from `Address` to `HtmlElement`.

In the first case, the developer's question can be answered by querying a web search engine. The problem is common amongst developers new to Scala and answered many times on various websites. In the second case, this is not an option and the developer has to fall back to a local solution. Despite general purpose search engines are often effective, they are far from optimal. Googling a problem requires to manually browse several proposed web sites and to check if the solution given is applicable to the concrete system (e.g. does it involve the same version of the library). Using `grep` or similar tools to locally search the code requires developers to be familiar with such tools and to be able to effortlessly formulate appropriate regular expressions.

Also, using a specialized search engine or a program synthesis tool to retrieve code snippets that solve the programming tasks is not necessarily well-suited. Despite such tools are probably capable of retrieving a correct snippet, their main purpose is not to retrieve single definitions. Furthermore, it is debatable whether such tools foster good coding practice. First, while code snippets are useful for learning about the available APIs and common usage patterns, there is a certain risk that developers blindly use the proposed snippets without fully understanding all aspects of the inserted code. And second, having code snippets readily available may encourage a copy & paste coding style. Instead of creating reusable abstractions for reoccurring problems, developers start to repeatedly use the same snippets in different locations. This concern is somehow supported by the findings reported in [WCGH15]. The evaluation of the usage data of Bing Code Search showed that "some other users issue the same query on different days, showing they are using the tool as a faster, task-level auto-completion".

An often effective way to answer our example queries are code completion assistants. Concerning the first example query, a user may quickly find the `mkString` method by invoking the assistant on an instance of `List`. Especially, if the proposed members can be filtered by keywords and matching return types. In the case of the second example query, traditional code completion is most likely of little help as the function wont be defined as a member of `Address`.

As Hoogle like search engines are designed to answer these kinds of questions, they are naturally well-suited for the task. However, there are varying levels of how generously definitions are matched by the currently available systems. An ideal system should also retrieve definitions with types similar to the query. Concerning the second example, if there is no definition with a type `Address => HtmlElement`, the user may be interested in definitions that return a subtype of `HtmlElement`. For object-oriented languages, we found no current work that used a satisfactorily notion of similarity such that subtyping relations are fully considered. While solutions like InSynth and CodeHint are capable of retrieving definitions similar to the query, they try to address a more ambitious problem which has consequences on the size of the search scope that can be covered.

1.5. About this Document

This report describes the design, implementation and evaluation of a specialized search engine for retrieving definitions from APIs. We start in chapter 2 with formulating the API retrieval problem and outline the requirements and fundamental design decisions. In the second part of chapter 2 we additionally discuss how the type system of the targeted programming language influence the semantics of API retrieval. The core of this thesis in chapter 3 describes our API retrieval model by first introducing the mathematical foundation and then describing and motivating our modifications to the vector space model. The following chapter 4 shows how we implemented our model for targeting the Scala language. In chapter 5, we describe how this implementation has been evaluated to support the assumptions made during the design phase. And finally, we give in the concluding chapter 6 an outlook on further work and how our approach can be adapted for targeting additional programming languages.

2. API Retrieval and Type Systems

This chapter draws an outline of the API retrieval problem that is addressed during this thesis and describes how API retrieval is related to information and data retrieval. Furthermore, we show how the type system of a targeted programming environment influences the requirements to an API retrieval system.

2.1. API Retrieval

We define API retrieval as the process of retrieving declarations from a programming API based on a user's information need. Opposed to a program synthesis or code snippet retrieval system, an API retrieval system does not aim to inform a user about how a programming task can be solved, but rather gives pointers to existing functionality that may help to implement said task.

The information needs that should be answered by an API retrieval system are derived from problems a user faces during the implementation of a programming task (see Appendix C). A typical information need may have one of the following forms:

- Is there a value with characteristic X?
- How can I construct an X given a Y?
- What operations are defined on X?
- Is there a function accepting an X and performing the effect Y?

The API retrieval system satisfies these information needs by providing fully-qualified identifiers of definitions that may be useful to solve the user's problem.

2.1.1. API Entities

We use the term "API Entity" to describe any definition in the global value namespace of a program. This includes the source code of the program itself and the libraries the program depends on. An API entity has a name and a type. Some examples of API entities are functions, values, constructors and class members (except type members). Optionally, there is an attached textual description of the characteristics of the entity, e.g., its documentation comment.

An API entity represents a document in the API retrieval system. Thus, API entities are the indexed items that are retrieved during the search process.

Classes, interfaces and other type definitions are not considered as API entities and cannot be retrieved directly. The motivation behind this decision is, that a type by itself cannot be used to solve a programming problem. Rather, programmers compose expressions of a certain type. Nevertheless, types can still be discovered through the operations that are defined on them.

2.1.2. Types

Depending on context and background of the speaker, there are several interpretations of the term "Type" in computer science. In this thesis we mainly use the notion that a type describes a set of operations that can be applied on values inhabited by a type.

The main motivation behind types in programming is to prove the absence of programming errors of a certain category. The exact kind of errors that can be eliminated depends on the sophistication of the type system used to check the program.

When it comes to API retrieval, we can leverage the documentary characteristic of types to increase the precision of the search engine. In general, a type limits the possible implementations of an entity and therefore the functionality provided. For example, an untyped definition of an entity with the name `max` does not provide much information. It may represent some upper bound of a numeric range, a function returning the larger of two numbers or a person called Max. If we know that `max` is of type `Int`, there are certainly fewer possibilities of what this entity represents. Naturally, this information can also be provided by a documentation comment like "the largest value representable as an Int" which describes the `max` value even more precisely.

While types may often be less precise than a well-written documentation they have other advantages: Types are less ambiguous and often required by the compiler. Thus, an `Int` represents a well-defined range of integer values with a certain set of available operations. Furthermore, many languages either automatically infer the type of an entity or require programmers to explicitly specify its type. Types are therefore always present, independent of whether the programmer cares about good documentation or not.

In languages that encourage a strong use of types by providing a powerful type system, the documentary characteristic of types is even more distinct. For example, a function returning the `Option` type in Scala clearly states that the computation may return a value or fail without any error message.

Altogether, we assume that users can express many information needs more precisely with types:

- Is there a value of type X?
- How can I construct a value of type X given a value of type Y?
- What operations are defined on values of type X?

Or through a combination of types and natural language:

- Is there a function accepting a value of type X and performing the effect Y?

These information needs are less ambiguous and should therefore be more likely to be answered satisfactorily.

2.1.3. Queries

A query language should allow users to express their information need such that it can be processed by the retrieval system. In API retrieval, a user should be able to describe API entities that provide a certain functionality. As we have discussed earlier, an important property of an API entity is its type. Furthermore, some aspects may only be described in natural language. These aspects should be expressible by textual keywords.

Altogether, we use the following high level syntax when discussing queries in this report:

```
query:
  type
  keyword* ':' type
```

Hence, a query is either a type or a list of keywords followed by a colon and a type. The exact query syntax in our prototype implementation is more elaborated but follows this basic pattern. In general, we use a syntax similar to Scala's to describe types. Following phrases are frequently used during this report:

Function Types
: The type of a function with a parameter type `A` and a return type `B` is written as `A => B`. We do not explicitly distinguish between method types and function types. Hence, `Int => String` can refer to a value with a function of this type, a method accepting an `Int` and returning a `String` or a member of `Int` returning a `String`.

Tuple Types
: A tuple is a finite, heterogeneous list of fixed length. A tuple's type is denoted by parentheses and the types of its components are separated by commas. Hence, `(Int, String)` is the type of a tuple containing an `Int` and a `String`. Tuples are also used to describe function types accepting more than one parameter, e.g. `(A, B) => C`.

Type Arguments
: A type `T` with a type argument `A` is written as `T[A]` and reads as "a `T` of `A`". Some examples are `List[Int]` or `Map[Int, String]`.

Type Variables
: We use the convention that type names consisting of a single uppercase letter do not refer to a concrete type but rather to a generic type variable. Hence, the query `A => List[A]` reads as "a function taking values of an arbitrary type `A` and returning a `List` of the same type `A`".

We assume that this syntax is sufficient to express most information needs. Textual keywords can be used to describe the expected side effects and characteristics of transformations not expressible by types and the type part describes the involved data.

2.2. Data and Information Retrieval

According to the terminology defined in [Van79], data retrieval is the activity of finding records in a set of well-structured and unambiguous data. Using a SQL query for retrieving records from a relational database management system (RDBMS) is a typical data retrieval task. Data retrieval systems give exact answers to the user's information need. For example, the query "what car models have been manufactured in 2015?" can be translated to a formalized query and be answered precisely by a RDBMS.

Information retrieval systems on the other hand, do not directly answer the users information need but rather inform them of the existence of documents that are relevant to the query. Hence, an information retrieval system does only give pointers to documents that may be useful to answer a given question.

Table 2.1.: Comparison of Data, API and Text Retrieval

	Data Retrieval	API Retrieval	Information Retrieval
Information Structure	Structured data	Structured data	Free text
Information Semantics	Well-defined	Well-defined	Ambiguous
Query Language	Artificial	Artificial	Natural
Query Specification	Complete	Incomplete	Incomplete
Expected Results	Matched Records	Relevant Records	Relevant Records
Robustness	Sensitive	Insensitive	Insensitive

Table 2.1 gives an overview of the differences between data and information retrieval systems and shows how API retrieval is related to both of them. The individual items of this comparison are further discussed in the following three subsections.

2.2.1. Information Structure and Semantics

Like data retrieval, API retrieval works on structured data with mostly well-defined semantics. Hence, we can use a data model to describe semantics and relations of each datum. This is in contrary to information retrieval where documents are at most semi-structured and the semantics of a word is often ambiguous.

Listing 2.1 shows a typical record from the Scala standard library's source code [ET]. It demonstrates the hierarchical structure of documents in Scala APIs as `MaxValue` is part of the object `Int` which again is a part of the package `scala`. Those *has a* relations are described in the Scala language reference [Ode14] and could be used to describe an according data model. Additionally, the modifiers `final` and `val` and the return type `Int` are further properties with well-defined semantics, whereas `Int` is a reference to the according type definition.

```
package scala

// ...

object Int {
  // ...

  /** The largest value representable as a Int. */
  final val MaxValue: Int = java.lang.Integer.MAX_VALUE
}
```

Listing 2.1: Excerpts from a source file of the Scala standard library

Beside these well-defined properties, the documentation comment between the tokens `/**` and `*/` and the name `MaxValue` of the value are less formalized. While the syntax of names is defined in the specification and naming often follows conventions or guidelines, the documentation comment is a description of the entity in natural language. Unfortunately, name and doc comments are often highly relevant to answer a users information need. E.g., a user looking for an entity of type `Int` is most likely interested in a specific value representing a physical constant whose concept is not formalized by the programming language but described by an identifier. In this case, the API retrieval system has to consider this information to retrieve the most relevant results.

2.2.2. Query Language and Specification

Data retrieval systems typically use an artificial query language with a restricted syntax like SQL and users give a complete specification of the data they are interested in. The user knows about the structure of the data and is therefore able to use specific filters and transformations to describe his information need.

Usually, a less formalized language is used to query information retrieval systems. The query language accepts a sequence of arbitrary words and there are only a few syntactic constructs for annotating keywords. E.g. some query languages allow users to precede keywords with a minus sign to denote that this term must not occur in the retrieved documents. Also, an information retrieval query is not an exact specification of the documents a user is looking for but rather a vague description of the documents with terms that seem relevant to the user.

In API retrieval, we can use an artificial language to describe an information need but we cannot assume that a user has sufficient knowledge to completely specify the type signatures he is looking for.

2.2.3. Search Results

The nature of the query language has major consequences on how search results can be retrieved. If there is a exact and complete specification of the documents it is sufficient

to report those documents that match the query. Also, the order of the results can be arbitrary unless the user explicitly specifies a sorting key.

In case of an incomplete query specification, users expect that only relevant documents are retrieved or, at least, that documents are sorted by their relevance to the information need. Hence, an information or API retrieval system must come with some notion of relevance that conforms with the users perception of relevance.

2.2.4. API Retrieval as an Information Retrieval Problem

As we have shown in the previous sections, API retrieval shares properties with both retrieval problems. Like in data retrieval, it operates on structured data with well-defined semantics and allows the use of an artificial query language. But, like in information retrieval, query specifications may be incomplete and a relevance based result set is often preferred.

This leads to the fundamental question whether we want to use a deterministic (Data Retrieval) or a probabilistic (Information Retrieval) approach to retrieve terms from API definitions. Please note, that choosing a deterministic approach does not necessarily exclude relevance based ordering of the result set. Also, with a probabilistic approach it is always possible to filter out entries that do not match the query. This is more a question about what technique is used to retrieve an initial set of results that may be passed to further processing steps.

The following points describe why we decided to try an approach based on information retrieval during this thesis:

Explorative Nature of the Problem

The tool we want to develop should help users to explore APIs. And exploring also means to face the unexpected and to discover that this is what you actually were looking for. Information retrieval systems are typically built to support queries of explorative nature [Bro02] and therefore seem to be naturally better suited for this use case.

Names and Documentation

Developers invest a significant part of their time to come up with descriptive term names and meaningful documentation comments. Both give an additional insight into the functionality of a method or the meaning of a value that is not covered by the type signature alone. We assume that the information content of doc comments and term names is in a similar order of magnitude as the information contributed by type signatures and therefore should be treated as a first class citizen for matching and scoring documents. Querying unstructured text documents is the main application area of information retrieval systems.

Scalability

The Scala Standard Library alone defines about 80'000 terms and 2'000 types and most non-trivial projects are likely to exceed these numbers sooner or later. This indicates that a responsive search engine that covers the scope of a complete project

including all library dependencies must be able to filter hundred thousands of documents within milliseconds. Furthermore, type hierarchies combined with parametric types lead to a super-linear increase of type signatures potentially matching a query with increasing number of type definitions (see subsection 2.3.4). This is why many similar tools that use a deterministic approach have to limit either the search scope or the expressiveness of queries. E.g. Prospector [MXBK05] allows only one input and one output type per query. Altogether, these two observations indicate that implementing a scalable search engine that uses exact matching is likely to be much harder than following a probabilistic approach.

As we want to address API retrieval like an information retrieval problem, it is possible to use a slightly adapted formulation of the IR problem [Van79] to describe API retrieval: Given a collection of API entities $C = d_1, ..., d_n$, a user query q and a set of entities relevant to the query $R(q) \subset C$, compute a ranked list of entities such that the entities in $R(q)$ have a ranking in $1, ..., |R(q)|$. Hence, we are looking for a way to compute the best ranking for the entities in C such that the user sees the most relevant results first. Displaying non-relevant results is not desired but also does not affect the effectiveness of the system as long as these entities are listed below the relevant results.

This notion of API retrieval is also important for the evaluation of our retrieval system in chapter 5. Furthermore, this interpretation assumes that users browse the result set sequentially from highest rank to lowest. If a user sees no results that are relevant to his query, he can assume that there are no according entities in the document collection.

2.3. Type Systems and API Retrieval

Many programming languages have a more or less sophisticated type system that checks if the types in a program are consistent. Hence, a type system consists of a set of rules that associates expressions and definitions in a program with a type. If none of these rules can be applied without a contradiction or if a assigned type conflicts with the type annotated by the programmer, a type error is raised.

This section introduces some important concepts of type systems used in popular object-oriented programming languages and discusses their impact on API retrieval.

2.3.1. Polymorphism

Polymorphism is the general term for techniques to provide a functionality for a range of types with one uniform interface. There is probably no modern mainstream programming language that does not support at least one kind of polymorphism.

Usually, polymorphism is subdivided into three major kinds:

Ad Hoc Polymorphism

Several functions with the same name are defined with different parameter and return types. This is often referred to as function or operator overloading.

Parametric Polymorphism
: A function or data type is defined such that the implementation does not depend on the type of some of the values it is applied on. Parametric polymorphism is often implemented with type parameters (or sometimes called "generics").

Subtype Polymorphism
: A type `A` is defined as a subtype of another type `B` if every operation that can be invoked with a value of `A` can also be invoked with a value of `B`.

Note, that this is just a theoretical classification. Programming languages sometimes use a single concept to achieve different forms of polymorphism or provide variations of these forms.

Concerning API retrieval, parametric and subtype polymorphism are the biggest sources of complexity compared to other features of type systems. Both forms implicate that the number of entities potentially matching a type query grows substantially. E.g., if a language does not support polymorphism, the query "a function accepting a value of type `A`" can be answered by using a simple regular expression that matches all function declarations that use `A` in the parameter list. This is no longer sufficient if polymorphism is involved. In this case, functions accepting a supertype of `A` and type parameterized functions have to be included in the search scope.

2.3.2. Subtyping

Subtype polymorphism is an inherent property of most (if not all) object oriented programming languages. The ability to include subtype relations in type queries is therefore crucial for an API retrieval system that should work on mainstream programming languages.

A type `S` is a subtype of a type `T` ($S <: T$) if a value of `S` can be used whenever a value of `T` is expected. This relation is usually defined reflexive, such that $S <: S$, and transitive, such that if $S <: T$ and $T <: U$ then $S <: U$.

Subtype polymorphism can be further classified into two groups:

Nominal Subtyping
: Types are explicitly defined to be a subtype of another type. One typical example of nominal subtyping is Java's interface inheritance. An an interface `A` is not a subtype of `B` unless the definition of `A` states that `A implements B`.

Structural Subtyping
: Subtype relations are implicitly derived from a type's structure. Given two record types `A` and `B`, $A <: B$ holds if the set of elements of `A` is a superset of elements of `B`. E.g. $\{a : Int, b : Long\} <: \{a : Int\}$.

A language that exclusively uses structural subtyping is Go. More precisely, Go allows the definition of interfaces that are implicitly implemented by a record if its structure matches the interface.

In Scala, both structural and nominal subtyping is supported. Unlike Go, structural subtyping in Scala does not require a named type. Instead of that, it is possible to describe the expected types by using a structural type. A structural type is a class body that only contains abstract member definitions. Listing 2.2 gives an example of a method that uses a structural type to describe the expected type. The `greet` method can be invoked with any object that has a member `name` of type `String`.

```
def greet(named: {def name: String}) = {
  println("Hi " + named.name + "!")
}
```

Listing 2.2: Structural Subtyping Example in Scala

When it comes to API retrieval, structural subtyping requires more statical analysis to mine all subtype relations in a program. While a compiler does only have to type check the structural subtype relations that are effectively used, mining for all subtype relations in a program requires to check all structural types (or Go like interfaces) against all other type definitions.

Because of this additional complexity and the observation that structural subtyping is almost never used in popular Scala libraries, we decided to not support structural subtyping in our API retrieval system for now.

2.3.3. Top and Bottom Types

One further distinction of type systems using subtype polymorphsim is whether the type system is unified or not. A unified type system has a type $\top$ for which $S <: \top$ holds for every S. Thus, every type is a subtype of a common root type $\top$. In type theory, this root type is referred to as the "Top" type.

Both Scala and C# are examples of languages with a unified type system. In Scala the top type is called `Any` and in C# `Object`. While Java also uses `Object` as a root type for every type defined by a class, its type system is not strictly unified as there are also primitive types that do not extend `Object`.

Usually, top types are used to provide some basic language features. E.g. Scala's `Any` defines the `toString` method that ensures that there is a textual representation for every object which may be useful to programmers during debugging.

On the other end of the type hierarchy, a language may also use a bottom type $\bot$ for which $\bot <: S$ holds for every S. In Scala, the bottom type is called `Nothing`.

One use of the bottom type is to use it as the type of expressions that never return a value. E.g., the `throw` expression in Scala is of type `Nothing` which allows method definitions like

```
def ???: Nothing = throw new NotImplementError
```

Using `???` as a stub for a method implementation as in

```
def theAnswer: Int = ???
```

will successfully type check because `Nothing <: Int` but terminates with an exception at runtime.

Having a unified type system with a top and bottom type has some consequences when it comes to API retrieval. First, it allows the definition of functions that have a type with minimal information content. A method with the signature

```
def identity(a: Any): Any
```

may be a valid declaration in Scala but its type `Any => Any` does not carry much information. As a consequence, applying API retrieval on libraries with many such declarations will never achieve high precision without including names and documentation comments into the query.

Another consequence is, that a unified type system allows extremely unspecific queries like `Nothing => Any` that can potentially match a huge number of entities.

2.3.4. Parametric Types

Beside subtype polymorphism, parametric polymorphism is another approach to write generic code that can be applied to a certain range of types. Like subtyping, some form of parametric polymorphism can be found in many popular languages like Java, C# and C++.

In Scala, parametric polymorphism is implemented through type parameters that can be added to method, class, trait and type alias definitions. For example, the definition

```
class Box[T](element: T)
```

defines a generic class with one type parameter `T`. A `Box` can be instantiated with any type argument for `T`. E.g. `new Box[Int](1)` creates a `Box` with the type argument `Int` which is written as `Box[Int]`. Because type arguments can usually be inferred from the according value arguments, it is also possible to write `new Box(1)` which results in a value of the same type.

A more useful example is the generic method declaration

```
def reverse[T](ts: List[T]): List[T]
```

which accepts a `List` containing elements of an arbitrary type and returns a `List` containing elements of the same type.

Many languages also offer the ability to specify upper and lower bounds to restrict the type argument of a parametrized type. Scala uses the syntax `T <: U` to specify that `T` has an upper bound of `U` or that `T` must be subtype of `U`. Accordingly, `T >: L` is used to specify that `T` has a lower bound of `L`. An example of a generic method using an upper type bound is

```
def sort[T <: Ordered](ts: List[T]): List[T]
```

This method can only be invoked on a `List` whose elements extends the `Ordered` trait. This trait specifies the ordering relation required to sort a list.

An API retrieval system that supports parametric types has to allow users to use both generic and concrete type queries and retrieve suitable results in both cases. For instance, a user searching for a function similar to `reverse` may be aware that there is probably a generic implementation and therefore uses a type query `List[T] => List[T]` parametrized by `T`. On the other hand, a user searching for how to sort a `List[Date]` may use a concrete type query `List[Date] => List[Date]` because he does not have a generic solution in mind.

2.3.5. Variance

The ability to use both subtyping and parametrized types leads to the question whether a `Box[Apple]` can be used where a `Box[Fruit]` is expected if `Apple <: Fruit`. Or, in other words, whether `Box[Apple] <: Box[Fruit]` or `Box[Fruit] <: Box[Apple]`. The answer to this question is, that it depends on how one intends to interact with the content of the `Box`.

For example, given a method

```
def unpack(b: Box[Fruit]) = {
  val f: Fruit = b.get
  // ...
}
```

that only reads the content of the `Box`, it is safe to call `unpack` with a `Box[Apple]`. Hence, we can state in this case that `Box[Apple] <: Box[Fruit]`. This can also be transfered to the real world analogy: If one ordered a box full of fruits and gets a box full of apples, one may be disappointed by the lack of variety but there is technically no reason to refuse the delivery. Using exclusively operations on a `Box[Fruit]` that either retrieves a `Fruit` or does not involve the type argument at all (e.g., an `isEmpty` operation) is called **covariant access** to a `Box` of `Fruit`.

But the matter is different if we need **contravariant access** to a `Box` of `Fruit`: Given a method

```
def pack(b: Box[Fruit]) = {
  b.set(new Fruit())
}
```

calling `pack` with a `Box[Apple]` should rise a type error because the box will no longer contain an `Apple` after `pack` returned. Though, it is type safe to call `pack` with `Box[Any]`. In this case, we can state that `Box[Any] <: Box[Fruit]`. Once again, the fruit analogy holds surprisingly well: If one prepares a fruit delivery and gets a box with apple-shaped cut-outs, one would not be able to insert an arbitrary fruit.

A third case is a method that both reads a `Fruit` from the `Box` and writes a `Fruit` back into it. This operation can only be invoked with a `Box[Fruit]`. Passing a `Box[Apple]`

would not type check because the method may put a `Fruit` into it and passing a `Box[Any]` would result in an error because the method expects that a `Fruit` is in the box. Thus, neither `Box[Apple] <: Box[Fruit]` nor `Box[Fruit] <: Box[Apple]` holds. In this case, we speak of **invariant access** to a `Box` of `Fruit`.

Finally, **bivariant access** is used when the receiving method calls only operations on the `Box` that do not involve the type argument. For example, a function that repaints a `Box[Fruit]` by setting `Box`'s `color` attribute to `"blue"` is invokable with a `Box[Apple]` as well with a `Box[Any]` which implies that `Box[Apple] <: Box[Fruit]` and `Box[Fruit] <: Box[Apple]`.

Despite it is possible to write a compiler that automatically derives which type of access is used in a certain expression [AHS11], most languages offer particular variance annotations to let programmers choose if a type constructor is used invariantly, covariantly or contravariantly. Altogether, there are two main approaches to support variance annotations:

Definition-site Variance Annotations

Languages using definition-site variance move the responsibility for deciding whether a parametrized data type can be accessed co- or contravariantly to the author of the data type. In Scala, this is denoted by a `+` for covariant and a `-` for contravariant before the definition of the type parameter (no annotation defaults to invariant). E.g. the following definition defines a `Box` with a type parameter `T` that can only be accessed covariantly:

```
class Box[+T](content: T) {
  def get: T = content
}
```

The `+T` annotation has a consequence that `Box` must not have a method using `T` contravariantly. Hence, a `set(t: T)` member would violate the variance annotation.

Use-site Variance Annotations

The other approach is to let users of a data type decide with what variance it is accessed. Hence, all parametrized data types are per default invariant over their type parameters but it is possible to use a type with a different variance. The most popular language using use-site variance is Java. But, mainly for Java interoperability, Scala also support use-site variance in form of existentially quantified types. When defining a method with a parameter of type `Box[Fruit]` used covariantly one can denote this by using `Box[_ <: Fruit]` as the parameter type. `_ <: Fruit` is an existentially quantified type with an upper bound of `Fruit`. The correct signature for `unpack` is therefore

```
def unpack(b: Box[_ <: Fruit])
```

Accordingly, we can define `pack` with an existentially quantified type with a lower bound of `Fruit`:

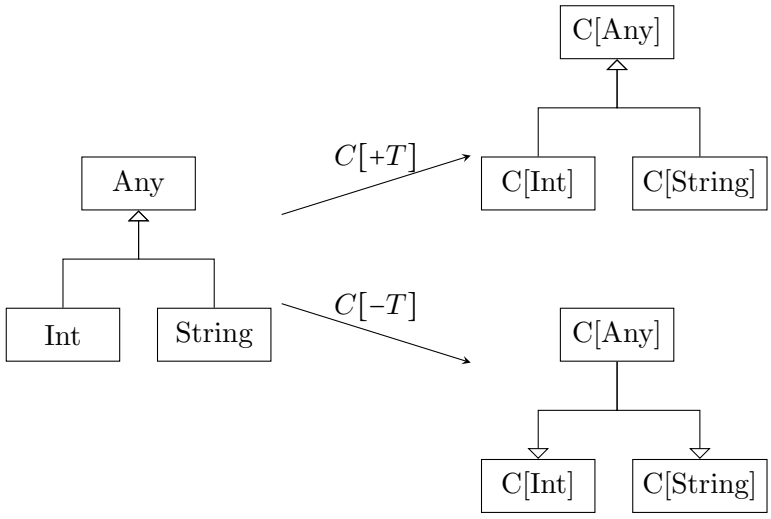

Figure 2.1.: Co- and contravariant type constructors

```
def pack(b: Box[_ >: Fruit])
```

Existentially quantified types also allows to use a type bivariantly by specifying no bounds:

```
def repaint(b: Box[_])
```

The equivalent types in Java would be `Box[? extends Fruit]`, `Box[? super Fruit]` and `Box[?]`.

Comparing these two approaches, use-site variance is more flexible than definition-site variance. Though, definition-site variance has been adopted by more programming languages as it is generally assumed to be simpler to use and easier to comprehend for programmers [Tat13].

Definition-site variance is also strongly related to covariant and contravariant functors in category theory [Pet12]. Functors are the theoretical foundation of type constructors and allow to reason about the variance of composed type constructors.

Figure 2.1 shows how co- and contravariant type constructors map subtype relations to the constructed types. The covariant constructor `C[+T]` associates to every subtype relation `A <: B` a subtype relation `C[A] <: C[B]` with the identical direction. A contravariant constructor `C[-T]` on the other hand, reverses the direction of a relation `A <: B` such that the associated subtype relation becomes `C[B] <: C[A]`.

This is also the reason why Scala uses + and − to encode variance as it allows to multiply variance annotations in nested applications of type constructors. Hence, given the type constructors `List[+T]`, `Promise[-T]` and `Array[T]` (invariant), it is possible to determine the variance of `T` in `List[Promise[List[T]]]` by multiplying $+1 * -1 * +1$ which results in a variance of -1 (contravariant). Invariance is the destructive element

0. Thus, both `List[Array[T]]` $(+1 * 0)$ and `Array[Future[T]]` $(0 * -1)$ are invariant over `T`.

Concerning API retrieval, variance is important for two reasons. First, the retrieval model should be able to incorporate variance annotations and match results accordingly. Hence, if `List[T]` is defined covariant over `T` and as a subtype of `Iterable[T]`, a query `List[Apple] => Unit` should retrieve an entity of type `Iterable[Fruit] => Unit`.

And second, definition-site variance annotations add valuable information about the relation between a parametric type and its type parameters. Variances tells if a type is consumed, produced or both which we assume to be of high relevance to users. E.g. a user searching for a function returning a `Fruit` may also be interested in functions returning a covariant container containing `Fruit`, e.g., a `List[Fruit]`, because there is an operation defined on this container that produces a `Fruit`. Though, a function returning a contravariant container of `Fruit`, e.g., a `Promise[Fruit]`, will much less likely be of interest to the user. By definition, this type cannot produce a `Fruit` or it could not have been defined contravariantly over its type parameter.

2.3.6. Summary

As we have seen, a programming language's type system has some influence on the requirements to an API retrieval system. For this thesis, we decided to address the problem for the Scala language first and therefore designed our retrieval system with a particular type system in mind. Altogether, our search engine should incorporate the rules of a type system with the following properties:

- Static Type Checking
- Nominal Subtyping
- Bounded Parametric Polymorphism
- Definition-Site Variance Annotations

The ability to statically check types at compile time is required to extract type information from source code. This does not strictly require that the targeted language uses a static type system by itself. Technically, an external tool that infers and checks types in source files would also suffice. The limitation to only support nominal subtyping is also not a strict requirement but simplifies the extraction of subtype relations.

A more drastic restriction is that we do currently only aim for supporting definition-site variance annotations. This limits the support for the Java language and existential types in Scala. This does not imply that Java will not be supported at all, but the precision of some queries whose most relevant results use existentially quantified types may be impacted.

Altogether, these framing conditions led to the design of an API retrieval model discussed in the next chapter.

3. An API Retrieval Model

Due to the reasons discussed in chapter 2, our goal is to integrate API retrieval into an existing information retrieval system. More precisely, we want to integrate our API retrieval model into the well-established Lucene search engine. Thus, we will first introduce the Vector Space Model (VSM) which is the mathematical model used by Lucene. The next section discusses our retrieval model derived from VSM. And finally, we elaborate how certain Scala specific language features are integrated into the basic retrieval model.

3.1. Vector Space Model

The Vector Space Model [SWY75] is a framework that can be used to discuss and compare information retrieval systems that share the basic principle of representing documents and queries as vectors in a document space. In this section, we use the notion of VSM as a framework as presented in [Zha15].

Each document in a collection $D = d_1, ..., d_m$ is represented as a n-dimensional vector $d_j = (w_{1,j}, ..., w_{n,j})$ where $w_{i,j}$ stands for the weight of term i in document j. The weight of a term should indicate how well the term characterizes a document. Typically, a weight greater than zero indicates that a term is present in the document.

Identically to document vectors, a query can be encoded as a vector $q = (w_{1,q}, ..., w_{n,q})$ where $w_{i,q}$ is again the weight of term i in the query. Here, the weight can be interpreted as how relevant a term is to the query. The weight function for documents and queries has not necessarily to be identical.

The last part that constitutes to a VSM system is a similarity function $sim(d_i, d_j)$ that computes a similarity coefficient between two vectors. These vectors may represent two documents or, in case of information retrieval, a document and a query.

Altogether, VSM groups information retrieval systems that share the following basic assumptions:

- Documents and queries are composed of independent, atomic parts (terms) that have a certain weight.
- The relevance of a document to a query correlates with the similarity of the two corresponding vectors.

Both of these assumptions show, that VSM uses a simplified notion of what a document identifies. For example, semantic dependencies between terms are not considered when

querying documents. Nevertheless, it is still possible to retain semantic information given a suitable method for extracting terms.

In order to implement a VSM system, we have to define the following aspects:

- How is a term defined and extracted from documents?
  - This determines the dimensionality n of term vectors.
- How are term weights assigned...
  - ...to document vectors?
  - ...to query vectors?
- What is the similarity function $\text{sim}(q, t)$?

In the following subsections, we show some possible answers to these questions.

3.1.1. Bag-of-Words Representation

A simple, but common implementation of how terms are mapped to vector dimensions is the Bag-of-Words representation (BoW).

BoW uses each word in the vocabulary V of a document collection D as a dimension in the vector space, such that $n = |V|$.

A naive approach to parse the document collection and build the vocabulary might be to treat all continuous sequences of alphanumeric characters as a word. But, depending on the kind of documents processed by the VSM system, applying further filters may be reasonable:

- Transforming upper-case to lower-case letters.
- Removing words that are very common in the document's language (Stop Words).
- Reducing inflected and derived words to their word stem (Stemming). E.g., "dogs" to "dog" and "ate" to "eat".
- Replacing words with more commonly used synonyms.

Often, these filters are motivated by performance considerations and aim to reduce the size of the vocabulary. Additionally, stemming ensures that various flections of a word in a query will match other flections of the same word in a document. Hence, if a user searches with the term "writing", he will likely also be interested in documents containing terms like "write", "wrote" and "writer".

3.1.2. Term Frequency

Assuming that a higher count of a word in a document indicates that the word is more relevant to the content of the document, term frequency seems intuitively to be a good measurement for weighting terms. The term frequency $\mathrm{tf}_{i,j}$ denotes how often the term i occurs in document j.

In combination with the BoW representation, the documents

> A fox jumps over the hedge.

and

> A jumping fox is a fox.

can be transformed to the term vectors d_1 and d_2 as given in Table 3.1.

$V = \{$	fox	jump	over	hedge	$\}$
$d_1 = ($	1	1	1	1	$)$
$d_2 = ($	2	1	0	0	$)$

Table 3.1.: Two documents represented as term vectors

The vocabulary V consists of all word stems occurring in one of the documents without the stop words “a”, “the” and “be”. Furthermore, each element in a term vector holds the according type frequency.

3.1.3. Inverse Document Frequency

One shortcoming of using term frequencies for term weights is that every term is weighted equally independent of its specificity. For example, terms that occur frequently in this chapter are “vector” and “term”. A quick Google search reveals, that there are approximately three times more websites matching “term” than “vector”. Hence, “vector” provides more information when describing the content of this chapter. Or in other words: “vector” is more specific than “term”. Incorporating this specificity into the term weight would be useful to ensure that a query like “term vector” favors documents containing “vector” over documents containing “term”.

One popular measure for the specificity of a term is the inverse document frequency (IDF):

$$\mathrm{idf}(w) = \log \frac{|D|}{|\{d \in D : w \in d\}|}$$

Where w is a query or document term and D is the document collection. Hence, IDF contains the quotient of the total number of documents and the number of documents containing the term w. The divisor $|\{d \in D : w \in d\}|$ is often referred to as the document frequency of w.

Depending on the specific VSM instantiation, IDF is incorporated into the weights of either the document vectors, the query vectors or both.

3.1.4. Dot Product Similarity

Probably the simplest similarity function for a VSM is the dot product similarity between a document d and a query q:

$$\text{sim}(d, q) = d \cdot q = \sum_{t \in V} w_{t,d} w_{t,q}$$

The motivation behind using the dot product is that the resulting scalar describes the angle between the two vectors. If d and q are more similar, the angle becomes smaller and the dot product is increased.

Another similarity function that is mentioned more frequently in VSM literature is the normalized dot product:

$$\text{sim}(d, q) = \frac{d \cdot q}{|d||q|} = \cos\theta$$

This is the dot product normalized by the product of the euclidean norms of d and q. Because this equals to the cosine of the angle θ between d and q, the relation to the similarity between d and q becomes more apparent.

But normalizing over the norm of the document vector comes with the downside that the document length is eliminated from the result. Hence, a long document with many occurrences of a query term does score equally as a shorter document with the same ratio of the query term. This may not be intended if a longer document works out the topic in more details.

Also, normalizing over the norm of the query vector is only necessary if one wants to compare scores from different queries.

3.1.5. Doc Length Normalization

One issue with using no normalization of the document length as proposed in subsection 3.1.4 is, that longer documents have a higher probability of containing a term by chance than shorter documents. For example, this report uses the term “fox” in the example documents in subsection 3.1.2 despite we do not provide any useful information about foxes. Without any normalization, this document would achieve a better score for the query “fox” than a short essay about foxes.

Thus, a normalization over the document length is still desirable, but the penalty for long documents should not increase linearly as in $1/|d|$. A simple function for document normalization, which is also used in Lucene’s default similarity function, is

$$\text{norm}(d) = \frac{1}{\sqrt{\sum_{t \in d} \text{tf}(t, d)}}$$

Note, that this function uses the sum of the term frequencies in a document, which resembles the length of the original document, instead of the norm of the document vector.

3.1.6. TF/IDF

One popular instantiation of VSM is TF/IDF (term frequency /inverse document frequency). All TF/IDF instantiations use term and query vectors derived from the BoW representation. Furthermore, the TF and IDF factors are used in at least one of either the query or document weights. A possible assignment of the weights would be

$$w_{t,d} = \mathrm{tf}(t, d)$$

for the weight of term t in document d and

$$w_{t,q} = \mathrm{idf}(t)^2$$

for the weight of term t in the query vector. An according similarity function is then

$$\mathrm{sim}(d, q) = \mathrm{norm}(d) \sum_{t \in V} w_{t,d} w_{t,q}$$

While this specific instantiation shows the basic idea, it resembles more or less the default implementation used in Lucene without additional features like document and field boosts. Other than that, there are more elaborated instantiations like Okapi BM25 [RWJ+94] that share the same foundation as TF/IDF but generally achieve a higher precision when applied on text documents.

3.2. Fingerprint Evaluation Model

As TF/IDF is a VSM instantiation designed for retrieving text documents, it is not particularly well-suited for retrieving API entities with type queries.

First, a naive mapping from type signatures to a BoW like representation is probably too destructive. E.g., one approach would be to create a BoW that simply contains each typename appearing in a type signature. Thus, the entity

```
def filter[T](ts: List[T], pred: T => Boolean): List[T]
```

can be represented as a BoW

```
List, T, =>, T, Boolean, List, T
```

But such a representation does not reflect how the types are related to each other. For instance, during retrieval it is not possible to tell whether the function accepts a function `T => Boolean` or `List[Boolean] => T`. Furthermore, what if a user searches with a type parameter named `A` instead of `T`?

Therefore, we need a better way to decompose type signatures and type queries into term vectors such that some of the semantic information of the type is still preserved.

Second, the TF/IDF similarity function does favor documents that have a higher term frequency for a term occurring in the query. Hence, a document `a a b c a` scores higher

than `a b c` against the query `a`. But this is not the expected behavior when retrieving types. Instead, a user searching for a type `Int => String` expects entities that have exactly one `Int` parameter. A function `Int => Float` should therefore score higher than a function `Int => Int => Float` and we need a similarity function that reflects this expectation.

During an empiric process, we developed an API retrieval model called *Fingerprint Evaluation Model* that addresses the aforementioned issues. While our model had its origins in VSM, it clearly diverted from VSM and can no longer be seen as an instantiation of it. In this section, we first introduce our model and then discuss its applicability on the API retrieval problem.

3.2.1. Normalized Types

The type fingerprint representation that we will discuss further in subsection 3.2.2 requires that every API entity has an assigned proper type. A proper type is a type that can be inhabited by a value [Pie02, p. 442]. It is not a type constructor and therefore is not parametrized. E.g., the types `Int` and `Int => String` are proper types, but not `List[T]` if `T` is refers to a type parameter.

In order to get a proper type from a parametrized type we use the following substitution rule:

Definition 1 (Type Parameter Normalization) *A type parameter P with the type bounds L and U such that $L <: P <: U$ is substituted by*

- *L if P is used covariant*
- *U if P is used contravariant*
- *$Unknown$ if P is used invariant*

Furthermore, we normalize proper types to a curried form like `A => B => C` instead of `(A, B) => C`. This simplifies the definition of type fingerprints and destructs implementation details that are not necessary for API retrieval. This is based on the assumption, that the user querying for an API entity is usually not interested in whether the functionality is provided by a class member or by a global function. Furthermore, functions and methods may be implemented in curried or uncurried form. These considerations lead to following definitions for normalized types of API entities:

Definition 2 (Normalized Value Types) *The normalized type of a global value that is not a class member is just its type.*

Definition 3 (Normalized Function and Method Types) *The normalized type of a function or method with the parameter types P_1 to P_n and a return type R is P_1 => ... => P_n => R.*

Definition 4 (Normalized Constructor Types) *A constructor accepting parameters of type P_1 to P_n and creating an instance of class R has a normalized type P_1 => ... => P_n => R.*

Definition 5 (Normalized Member Types) *The normalized type of a class member is the type of a curried function taking an instance of the class and returning a function with identical parameter and return types as the member.*

The currying of functions, methods and constructors is also backed by the equivalence relations used in [Rit90].

For example, the method declaration

```
def x(i: Int, f: (Int, Bool) => String): String
```

defines the API entity `x` with a normalized type

```
Int => (Int => Bool => String) => String
```

A more complete example is the polymorphic method

```
def head[T](xs: List[T]): T
```

As the type parameter `T` is unbound it is implicitly bound by the top and bottom types $\top$ and $\bot$, which are `Any` and `Nothing` in Scala, `head` has a normalized type of

```
List[Any] => Nothing
```

Altogether, the class definition in Listing 3.1 defines two API entities: The constructor `Box` of type `Cargo => Box` and the member method `unbox` of type `Box => Nothing`. Here, the upper bound of `T` is explicitly specified as `Cargo` and the lower bound is once again `Nothing`.

```
class Box[T <: Cargo](content: T){
  def unbox: T = content
}
```

Listing 3.1: A polymorphic class with a type parameter T that must be a subtype of Cargo

3.2.2. Type Fingerprint Representation

One of the basic assumptions of VSM is that a document can be decomposed into distinct terms of which each term characterizes the document independently of the other terms. Hence, to find documents that are relevant to a query, it is sufficient to only consider what terms are used in the document and the semantic relations between these terms are not of concern during the actual retrieval process. This assumption allows to use a

simple representation of the documents, e.g., bag-of-words, and fast retrieval techniques like inverted indexes.

In order to adopt these techniques to solve the API retrieval problem, we need a mapping from types to terms that retains enough information to give a good relevance scoring. The representation we developed and use during this thesis are *Type Fingerprints*.

Definition 6 (Fingerprint of Atomic Types) *The type fingerprint p_T of a normalized type T that has no type arguments is $p_T = [(+1, T_{name})]$.*

For example, the type fingerprint of `Int` is $[(+1, \text{Int})]$. The plus one term +1 refers to the variance of the type. We encode the variance (or polarity) of a type with the terms +1 for covariance, −1 for contravariance and 0 for invariance.

Definition 7 (Fingerprint of Types with Arguments) *The fingerprint of a normalized type $T[A_1, ..., A_n]$ with n type arguments is $p = [(+1, T_{name})] \oplus (p_{A_1} \times A_{1,var}) \oplus ... \oplus (p_{A_n} \times A_{n,var})$.*

Definition 7 uses $\oplus$ as the list concatenation operator. $p_{A_i} \times A_{i,var}$ is the application of the variance of the i-th argument to the fingerprint p_{A_i}. This means that the variance of each element in p_{A_i} is multiplied by the variance of A_i.

Given a type `List[A]` whereas `List` is covariant over the type parameter `A`, the type fingerprint of `List[Int]` is according to the second rule $[(+1, \text{List}), (+1, \text{Int})]$. Argument types at contravariant positions are, for example, function parameter types. Hence, `Int => String` has an according fingerprint of $[(+1, =>), (-1, \text{Int}), (+1, \text{String})]$.

Please also note, that variance multiplication is applied recursively. Therefore, the fingerprint of `(Char => Int) => String` is

$$[(+1, =>), (-1, =>), (+1, \text{Char}), (-1, \text{Int}), (+1, \text{String})]$$

Or, in other words, the variance of an argument of a function that itself is an argument again is +1.

One issue with this representation is that the specificity of the `=>` type becomes extremely low. Almost all normalized definitions in a programming library are of a function type and the document frequency of `(+1, =>)` would therefore be almost equal to the size of the document collection. In order to make queries more specific where a user searches for functions accepting callbacks, we additionally use Definition 8 to define fingerprints.

Definition 8 (Fingerprint of Function Types) *The type fingerprint of a normalized type $(A_1, ..., A_n) => R$ that is not by itself an argument type of a function type is $p = (p_{A_1} \times -1) \oplus ... \oplus (p_{A_n} \times -1) \oplus p_R$.*

This results in fingerprints that do not include the $(+1, =>)$ terms of the outermost function applications. For instance, `A => B => C` becomes $[(-1, A), (-1, B), (+1, C)]$.

On the other hand, arrows of inner function calls are preserved: `(A => B) => C` becomes $[(-1, \texttt{=>}), (+1, A), (-1, B), (+1, C)]$.

In summary, a type fingerprint is a unordered list of tuples $(variance, type)$. We use this representation to define the dimensionality of a term vector in our VSM implementation.

For further discussions, we use a shorter notation `<variance><type>` where the variance is encoded as `+` for covariant, `-` for contravariant and `/` for invariant. E.g., the fingerprint $[(+1, A), (-1, B), (0, C)]$ is written as `+A, -B, /C`.

3.2.3. Justification of Type Fingerprints

The discussion about type fingerprints did, up to now, not include the question, whether this representation retains enough information to successfully retrieve API entities with a similar type. Naturally, we believe that the answer to this question is "yes" and the strongest point speaking for this assumption is that we achieved reasonably good results when testing the approach with a test collection. But there are also arguments for type fingerprints from a more theoretical point of view.

First, the variance with which a type occurs in a signature is an important indicator on how the type is used. In general, covariance indicates that the caller receives or reads the according value and contravariance indicates writing or submitting a value. Accordingly, a value at an invariant position is both read from and written to. The impact of the variance can be seen when comparing the fingerprints of the following four declarations:

`def m1(a: Int): Unit`	`-Int, +Unit`
`def m2(): Int`	`+Int`
`def m3(f: () => Int): Unit`	`-=>, -Int, +Unit`
`def m4(f: Int => ()): Unit`	`-=>, +Int, +Unit`

Both `m1` and `m3` eventually expect that the user passes a value of type `Int`. Whereas `m1` directly accepts an `Int` argument, `m3` takes a function that returns an `Int` if called. This is represented in both fingerprints by the term `-Int`. On the other hand, `m2` and `m4` both return an `Int` as indicated by the `+Int` term in the according fingerprints.

Additionally, many type constructors in Scala (and other modern programming languages) are designed such that they only allow either read or write access on their boxed types. Thus, the read/write semantics of variance annotations apply transitively:

```
def m5(a: List[Int], b: Promise[Char]): Unit
```

The fingerprint terms of the first argument are `-List` and `-Int` because `List[T]` is covariant over `T`. On the other hand, a `Promise[T]` is a handle that eventually can complete a `Future[T]` [HPM+12]. It is contravariant over `T` and there are only operations defined on a promise that accept either a `T` or an error. Thus, a caller of `m5` receives an `Char` value through the completion of the promise which explains the resulting fingerprint terms `-Promise` and `+Char`. The complete fingerprint of `m5` is

```
-List, -Int, -Promise, +Char, +Unit
```

Second, creating type fingerprints is not such a destructive operation as one would initially think, because the variance of the atomic types is retained. The number of possible type signatures matching a certain fingerprint is usually quite small. For instance, there is only one possible type signature whose fingerprint exactly matches `-Int, +Unit`: `Int => Unit`.

The main loss of information typically happens during the substitution of type parameters by their upper and lower bounds. This is probably the greatest liability of the type fingerprint representation and can influence accuracy when searching for highly generic signatures with more than one type parameter involved. For instance, a polymorphic declaration of `foldLeft` in Scala is

```
def foldLeft[A, B](as: List[A], init: B, f: B => A => B): B
```

which has a normalized type of

```
List[Any] => Any => (Nothing => Nothing => Any) => Nothing
```

This type is not distinguishable from the normalized type of `foldRight`. Thus, querying for `foldLeft` inevitably yields the same score for `foldLeft` and `foldRight` which is probably not what a user would expect.

Altogether, the information loss due to the type fingerprint transformation is not too severe when working with libraries that have a similar level of genericity as the Scala standard library. For instance, the `List` class in the Scala collection library defines 177 operations with 118 distinct type signatures (including inherited members). When further transforming these signatures we get 107 distinct fingerprints. From these fingerprints there are only 7 that are derived from different types. Furthermore, almost all of these fingerprint collisions occur on operations that have very similar semantics like `foldLeft`/`foldRight` and `reduceLeft`/`reduceRight`.

The only exceptions include methods inherited from Java's `Object` type that do not leverage the full capacities of the type system. One exception is `equals` and `contains`:

```
def equals[A](as: List[A], other: Any): Boolean
def contains[A](as: List[A], a: A): Boolean
```

Both methods share the fingerprint `-List, -Any, -Any, +Boolean` and are therefore not distinguishable during type retrieval. Although, the signature of `equals`, which is inherited from Java's `Object`, is too generic in most cases and would be considered bad practice in modern Scala code.

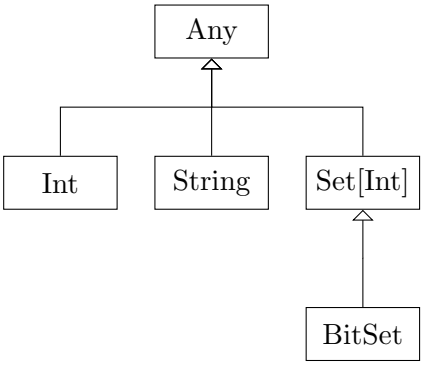

Figure 3.1.: Simplified type hierarchy as used in the sample queries

3.2.4. Type Queries

We interpret querying an API for a type T as asking for all entities e for which $e : T$ holds. Or in other words, asking for all entities that can be assigned to a variable of type T. Given a type system that uses subtype polymorphism, this also includes all entities of a subtype of T. Hence, given an entity $e : U$, $e : T$ also holds if $U <: T$.

If we use the fingerprint representation of a query type T as the query vector, we can retrieve entities that are of a similar type to T. This is demonstrated in Table 3.2. t_1 encodes a entity of type `BitSet => Set[String]` and t_2 is of type `Set[Int] => Int => String`. The set of terms used in both entities form the vocabulary V. Each element of a term vector indicates the term frequency of the term at the according position in V. Finally, q encodes the query type `Set[Int] => String`.

$V = \{$	-BitSet	-Set	-String	-Int	+Set	+String	$\}$
$t_1 = ($	1	0	0	0	1	1	$)$
$t_2 = ($	0	1	0	2	0	1	$)$
$q = ($	0	1	0	1	0	1	$)$

Table 3.2.: Example query vector q and two terms t_1 and t_2

Without defining the specific similarity function, we can see, that t_2 is probably more similar to q. In fact, the function represented by t_2 differs only in the additional `String` parameter from q.

Unfortunately, this approach does only retrieve entities that inhabit the query type `T` but none that inhabit a subtype of `T`. To also include these types, we can create a query vector for each subtype of `T` and use each of these vectors to query for terms.

Given the inheritance hierarchy as shown in Figure 3.1, the query

```
BitSet => String
```

has the following subtypes:

```
Set[Int] => String
Set[Any] => String
Any => String
```

Note, because `=>` is contravariant over its parameter type, we have to substitute the first type argument with its basetype. Mapping these alternative query types to the according fingerprint results in a total of four query vectors as illustrated in Table 3.3. While the original query vector q_1 mainly overlaps with t_1, there is now also some similarity between q_2 and t_2.

$V = \{$	-BitSet	-Set	-String	-Int	-Any	+Set	+String	$\}$
$t_1 = ($	1	0	0	0	0	1	1	$)$
$t_2 = ($	0	1	0	2	0	0	1	$)$
$q_1 = ($	1	0	0	0	0	0	1	$)$
$q_2 = ($	0	1	0	1	0	0	1	$)$
$q_3 = ($	0	1	0	0	1	0	1	$)$
$q_4 = ($	0	0	0	0	1	0	1	$)$

Table 3.3.: The expanded queries q_1 to q_4

While this approach is suitable for giving a picture of the underlying idea, it is inapplicable because the number of possible subtypes explodes given a query type of a certain complexity. For example, if `T` has a total of 3 subtypes, there are already $3^2 = 9$ subtypes for the tuple `(T, T)`.

3.2.5. Query Expressions

This is why we moved away from the idea of representing a query type as a term vector. Instead of that, we use an expression to represent the query and all possible subtypes. Furthermore, the expression incorporates the similarity function *sim* such that the expression can be evaluated against an entity's type fingerprint to calculate a score.

We use a recursive data structure to represent query expressions:

```
Sum(parts: List[Max | Leaf])
Max(alternatives: List[Sum])
Leaf(v: Variance, tpe: String, boost: Float)
```

```
def sum(t: NormalizedType, v: Variance) =
  Sum(
    Leaf(v, t.name) :: t.arguments.map(arg => max(t, arg.variance * v))
  )
```

Listing 3.2: Construction of a sum node

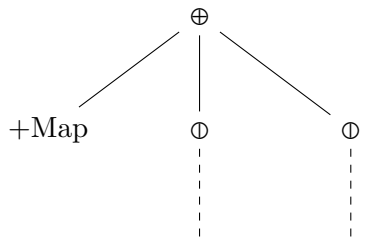

Figure 3.2.: Construction of a Sum node for the type Map[A, B]

A `Sum` node is a composite of at least one part that is either a `Max` or a `Leaf` node. To construct a `Sum` node we can use a function `sum` as sketched out in Listing 3.2.

`sum` creates a `Sum` node that contains a `Leaf` for the type itself and a `Max` node constructed by `max` for every type argument. E.g., `sum(Map[A, B], Covariant)` creates a tree as shown in Figure 3.2. This illustration uses the ⊕ operator to indicate `Sum` nodes and ⊙ to indicate `Max` nodes.

Furthermore, a `Max` node contains one or more `Sum` nodes and is constructed by the `max` function (Listing 3.3).

```
def max(t: NormalizedType, v: Variance) = {
  val alternativeTypes = v match {
    case Covariant     => subtypesOf(t)
    case Contravariant => supertypesOf(t)
    case Invariant     => t :: Unknown :: Nil
  }

  Max(sum(t, v) :: alternativeTypes.map(alt => sum(alt, v)))
}
```

Listing 3.3: Construction of a max node

`max` looks up types that can be used instead of the original type t. This includes subtypes of t if t appears in a covariant position or supertypes of t if it is used contravariantly. Type arguments at invariant positions form a special case: They are represented by the synthetic alternative type `Unknown`.

```
def queryExpression(t: NormalizedType) = {
  def parts(t: NormalizedType) = t match {
    case Function(a, r) => max(a, Contravariant) :: parts(r)
    case _              => max(t, Covariant) :: Nil
  }

  Sum(parts(t))
}
```

Listing 3.4: Construction of a query expression from a normalized type t

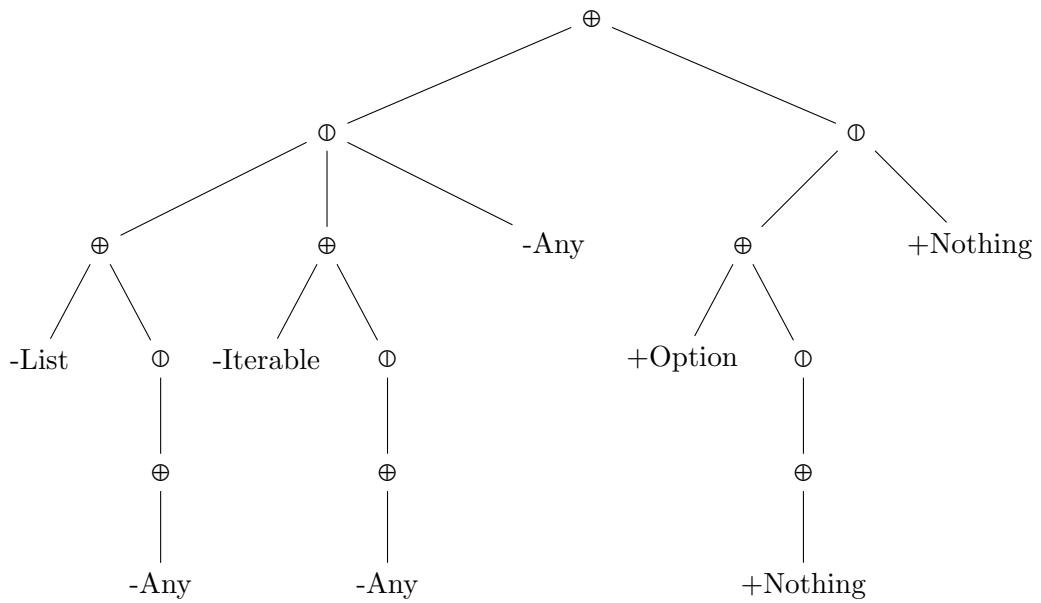

Figure 3.3.: Expression tree for the query `List[T] => Option[T]`

Finally, we use a function `queryExpression` to construct an expression from a normalized type as shown in Listing 3.4. As discussed in subsection 3.2.2, we don't want to include the outermost function applications. This is why `queryExpression` does not directly call `sum` on the input type but rather uses the `parts` function to get the `Max` nodes of all argument and return types of the query function.

An example of a complete query expression is given in Figure 3.3. This tree illustrates the expression for the query `List[T] => Option[T]` with a normalized type of `List[Any] => Option[Nothing]`. We assume in this example that `List[T]` extends `Iterable[T]`. Furthermore, every type extends the top type `Any` and every type is extended by the bottom type `Nothing`.

The query expression given in Figure 3.3 may also be written as

```
⊕(
   ⊖(
      ⊕(-List, -Any),
      ⊕(-Iterable, -Any),
      -Any),
   ⊖(
      ⊕(+Option, +Nothing),
      +Nothing))
```

Furthermore, we can factor out redundant subexpressions. This is possible because query expressions are assumed to be distributive over ⊕:

$$\oplus(\oplus(A, B), \oplus(B, X)) = \oplus(\oplus(A, B), X)$$

Hence, the expression in Figure 3.3 can be simplified to

```
⊕(
   ⊕(
      ⊕(
         ⊕(-List, -Iterable),
         -Any),
      -Any),
   ⊕(
      ⊕(+Option, +Nothing),
      +Nothing))
```

Altogether, a query expression is a concise representation of a query that also covers subtype relationships and does not suffer from combinational explosion.

3.2.6. Term Weights

Up to now, we have discussed how to create query expressions, but we did not yet mention how weights are assigned to the leaves of an expression tree. We have identified four properties that influence the weight of a leaf. Each of these properties is explained in the following subsections.

Inverse Document Type Frequency

The inverse document type frequency (ITF) is inspired by the IDF statistic (subsection 3.1.3) and closely related to it. Like IDF, ITF should capture the specificity of a fingerprint term such that types occurring more frequently in the document collection can be penalized.

But using the plain number of documents that contain a fingerprint term is not very accurate in capturing how often a type occurs: Given the types A and B with the subtype relation $B <: A$. If there are 10 API entities with a normalized type `B => Unit` and one entity `A => Unit`, the fingerprint term `-B` has a document frequency of 10 and `-A` one of 1. As a result, searching for a function `B => Unit` would overpenalize documents containing `-B`. Thus, the function `A => Unit` probably gets a higher score despite the fact that there are 10 other entities that have a lower distance to the query type.

In order to incorporate type hierarchies in the document frequency factor we use a slightly different notion of specificity:

> The specificity of a fingerprint term is the inverse of the probability that the term will occur in a query expression.

Hence, a term occurring in almost all query expressions like `-Any` and `+Nothing` has a very low specificity.

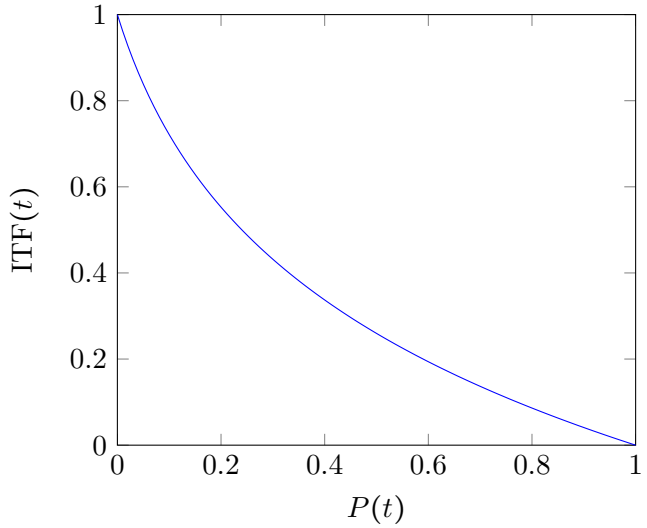

Figure 3.4.: ITF as a function of the relative type document frequency $P(t)$

To determine a term's probability $P(t)$ we use the type signatures of all entities in the document collection D as hypothetical queries. We then count the number of the derived query expressions that contain the term t which results in the absolute document type frequency $\mathrm{df}(t)$. The probability is then

$$P(t) = \frac{\mathrm{df}(t)}{|D|}$$

This approach has the advantage that it does not require an exhaustive collection of sample queries and that it remains conceptually close to IDF.

Finally, the inverse document type frequency is

$$\mathrm{ITF}(t) = \log_{10} \frac{10}{10P(t) + (1 - P(t))}$$

The term $(1-P(t))$ is added to normalize $\mathrm{ITF}(t)$ in the range $[0,1]$ when $P(t) \in [0,1]$ as shown in Figure 3.4. This is not a necessity but makes it easier to incorporate ITF in term weights and relate it to the other weighting factors.

Depth Boost

Given a type query with several nested atomic types, one can assume, that not all of these atomic types are equally relevant to the user's information need. For example, a user may use the query `List[Int] => String` when he is looking for a function that takes a `List` and concatenates all elements to a resulting `String`. In this case, the `Int` type argument of `List` is probably of less relevance than the remaining parts of the query.

One possible measurement for weighting the individual parts of a query is the nesting level in which the type occurs. Thus, a type with a higher nesting level is considered less relevant. The nesting level of an atomic part of a query type is equal to the number of enclosing types. Given the example query `A[B[C]] => D` the atomic types `A` and `D` have a nesting level of 1 because they are enclosed by the `=>` type (the same query can also be written as `=>[A[B[C]], D]`). Accordingly, `B` has a nesting level of 2 and `C` one of 3.

The according weighting factor is called *depth boost* which is the inverse of the nesting level. The depth boost $\text{deb}(i)$ of the i-th atomic type in the query type is

$$\text{deb}(i) = \frac{1}{l(i) + 1}$$

where $l(i)$ refers to the according nesting level.

Distance Boost

We can assume that a user always searches with the types that are most relevant to his information need. This would imply that types added to query expressions due to subtype relations are generally less relevant than the original type in the user query.

We can reflect this assumption by incorporating the distance in the inheritance hierarchy between a derived type and the original type into term weights. E.g., the expression tree in Figure 3.6 represents the query `List[T] => Option[T]`. During the creation of the tree we derived alternative types for `List[T]` (`Iterable[T]` and `Any`) and for `Option[T]` (`Nothing`).

The weight factor that we use to reflect this is called *distance boost*. Once again, we use the inverse to calculate the distance boost $\text{dib}(t_o, t_d)$ where t_o is the original type and t_d is the derived type:

$$\text{dib}(t_o, t_d) = \frac{1}{\text{distance}(t_o, t_d) + 1}$$

There are several possible approaches to calculate $\text{distance}(t_o, t_d)$. But an implementation must adhere to the following constraints:

- $\text{distance}(T, T) = 0$ for every T
- If $S <: T$ and $T <: U$ then $\text{distance}(S, T) < \text{distance}(S, U)$

One function that meets this constraint and that we use in our implementation is the index of t_d in the class linearization order of t_o [OZ05]. For example, the type definitions

```
class A extends B with C
trait B extends D
trait C extends D
trait D
```

result in a class linearization {`A, B, C, D`} for `A`. Hence, $\text{distance}(A, A)$ is 0 and $\text{distance}(A, C)$ is 2.

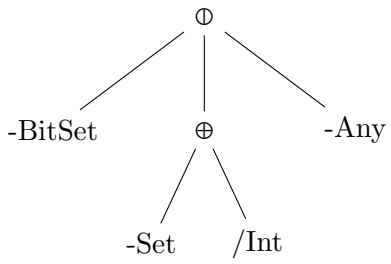

Figure 3.5.: Simplified expression tree for the query `Set[Int]`

Fractions

Each leaf in an expression tree represents a certain portion of the original type query. E.g., the query `BitSet => _` expands to the expression tree as shown in Figure 3.5. If each leaf would be weighted equally, the terms `-Set` and `\Int` would be overrepresented. Hence, the API entity `Set[String] => Array[Int] => Unit` would receive a higher score than the entity `BitSet => Unit` despite the later is much more likely to be a better match.

To adjust this over-scoring of subexpressions with more leaves than the adjacent subexpressions we assign *fractions* to branches of the expression tree. A fraction indicates what portion of the original query is represented by a certain branch or leaf.

Fractions are assigned to the leaves of an expression according to the following rules:

- The root node has a fraction value of 1.
- Each child of a `Max` node receives the same fraction value as the parent node.
- The fraction value of a child of a `Sum` node with n children is f_p/n where f_p is the fraction value of the `Sum` node.

When applied on the expression tree in Figure 3.5, the nodes `-BitSet` and `-Any` will each receive a fraction value of 1 and the nodes `-Set` and `/Int` a value of 0.5.

Combined Term Weight Factors

Finally, the distinct weighting factors have to be consolidated into one weight per leaf. Experimenting with various functions revealed that the product of the fraction value of the leaf and the weighted geometric mean of the remaining factors is a feasible option:

$$w(l) = l_{fraction} * exp\left(\frac{w_{dib}\ln l_{dib} + w_{deb}\ln l_{deb} + w_{ITF}\ln l_{ITF}}{w_{dib} + w_{deb} + w_{ITF}}\right)$$

l_{dib} refers to the distance boost associated with the leaf l, l_{deb} to the depth boost and l_{ITF} to the inverse document type frequency. The according weights w_{dib}, w_{deb} and w_{ITF} are positive rational numbers and can be adjusted to improve the effectiveness of the retrieval model (see evaluation (chapter 5)).

Despite the empirical evidence that supports the use of this function, there is also some rationale behind this choice. First, the geometric mean is well-suited for averaging normalized values and ratios [FW86]. Because the weighting factors represent a relative relevancy of a term (a ratio), this seems like a good match.

Second, applying the fraction factor after averaging the remaining factors seems arbitrary but it ensures that no branch of the expression tree will ever dominate all other branches. Including the fraction into the mean with a relatively low weight results in high scores for deeply nested terms with a high ITF value. On the other hand, a relatively high weight for the fraction factor would address this issue but the remaining factors would be underrepresented.

3.2.7. Evaluating Query Expressions against Type Fingerprints

To score an API entity against a type query we have to apply the type fingerprint to the according query expression. Evaluating a query expression means to mark the leaves in the expression tree such that the sum of the scores of the marked leaves is maximized. Furthermore, the following constraints apply when marking leaves:

1. For each term in the fingerprint at most one leaf can be marked.
2. A Leaf(v_q, t_q) can only be marked if there is a fingerprint term (v_p, t_p) such that $v_p = v_q$ and $t_p = t_q$.
3. Only one subexpression of a `Max` node can contain marked leaves.

This optimization problem is surprisingly hard. The third constraint prevents us from using a local approach that evaluates subexpressions whose result can be merged together: Switching a mark in one subexpression may lead to modifications of marks in several other subexpressions. This gives the problem an appearance of non-locality that is inherent in many NP-hard problems. Thus, we suspect that finding the maximum score of a type fingerprint given a query expression is very likely to be part of the set of NP-hard problems.

Fortunately, we have found a good heuristic to attack this problem efficiently if we can make some restrictions on the scores assigned to the leaves. We will introduce this heuristic in subsection 4.6.5 and, for now, just assume that there is an algorithm that can evaluate query expressions in $O(n * m)$ where n is the number of terms in the fingerprint and m is the size of the expression tree.

Figure 3.6 shows how the expression tree derived from the query `List[T] => Option[T]` can be evaluated with the fingerprint `-List, -Any, -Any, +Nothing` (derived from the type `List[T] => T => Option[T]`). The tree also includes the score of each leaf in superscript and the optimal marking is highlighted in green. Thus, the optimal score of the fingerprint in this example is $1 + 0.1 + 0.5 = 1.6$.

Note, that not all terms of the fingerprint resulted in a marked node. For the second occurrence of `-Any` there is no matching leaf that can be marked without violating the third constraint that only one subexpression of a `Max` node can contain marked leaves.

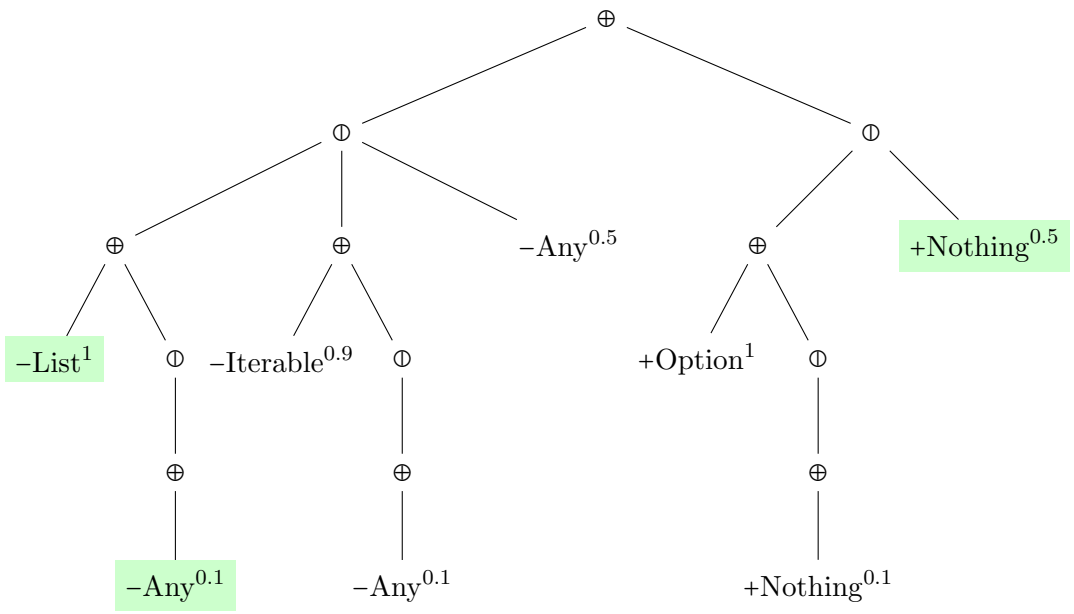

Figure 3.6.: Expression tree for the query `List[T] => Option[T]` marked with the fingerprint `-List, -Any, -Any, +Nothing`

3.2.8. Fingerprint Length Normalization

Because fingerprint length normalization factor is applied per document, it is not part of the term weight. Nevertheless, it is one of the key factors contributing to a good scoring function.

Like document length normalization, fingerprint length normalization incorporates the observation that additional terms in the document vector increase the possibility of matching a query term. Applied to type retrieval, we can state that API entities with more parameters are more likely to include fingerprint terms occurring in a query. This also applies to entities with more complex parameters and return types (e.g., `List[Int]` instead of `Int`).

We reflect these observations by incorporating the length of the fingerprint into a normalization factor for each document. But opposed to document length normalization as discussed in subsection 3.1.5, we can also make an assertion about the length of the expected fingerprint. Hence, we assume that the best matching entities should have a fingerprint of approximately the same length as the query's fingerprint. This is why we use the square of the difference between the query fingerprint length and the entity fingerprint length to normalize document scores:

$$\text{norm}(d) = \frac{1}{(w_{\text{norm}}(|\text{fingerprint}(q)| - |\text{fingerprint}(d)|))^2 + 1}$$

The additional w_{norm} weight is used to control the influence of $\text{norm}(d)$ on the final score. Two examples of the $\text{norm}(d)$ function with an expected fingerprint length of 5

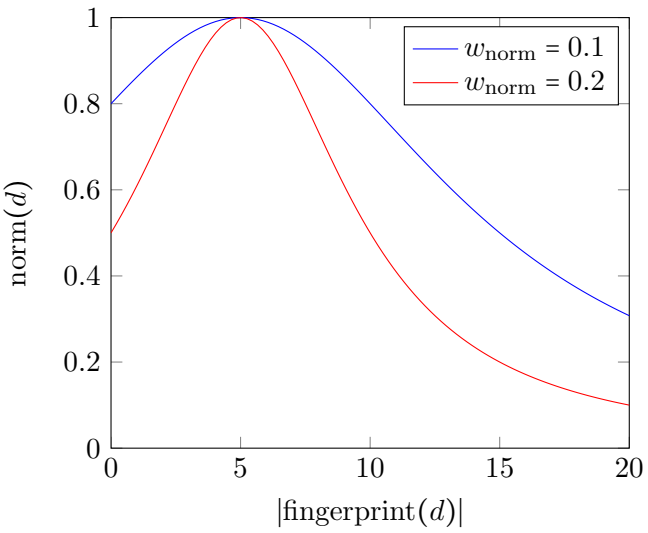

Figure 3.7.: The fingerprint length normalization function with a query fingerprint length of 5 and different values for w_{norm}

and a weight of 0.1 and 0.2 respectively is given in Figure 3.7.

3.2.9. Summary

During the development of our API retrieval model we had to abandon the idea of representing types as a composite of independent and atomic terms as suggested by VSM. Thus, the score contributed by an individual fingerprint term is no longer independent of the scores contributed by the remaining terms. Any attempt to implement a retrieval model that retained this assumption was either impractical due to combinatorial explosion (as discussed in subsection 3.2.4) or failed to fully incorporate the semantics of subtype polymorphism.

Probably the main strength of the resulting fingerprint evaluation model is that subtype relations can be fully incorporated during the retrieval process. On the other hand, support for parametric polymorphism is weaker as type parameters have to be substituted by proper types during type normalization which leads to a significant loss of information. Nevertheless, the fingerprint evaluation model still yields results with a high enough precision to be a useful tool for retrieving functionality based on type signatures (see chapter 5).

Figure 3.8 illustrates the retrieval process with the various representations for types introduced in this chapter.

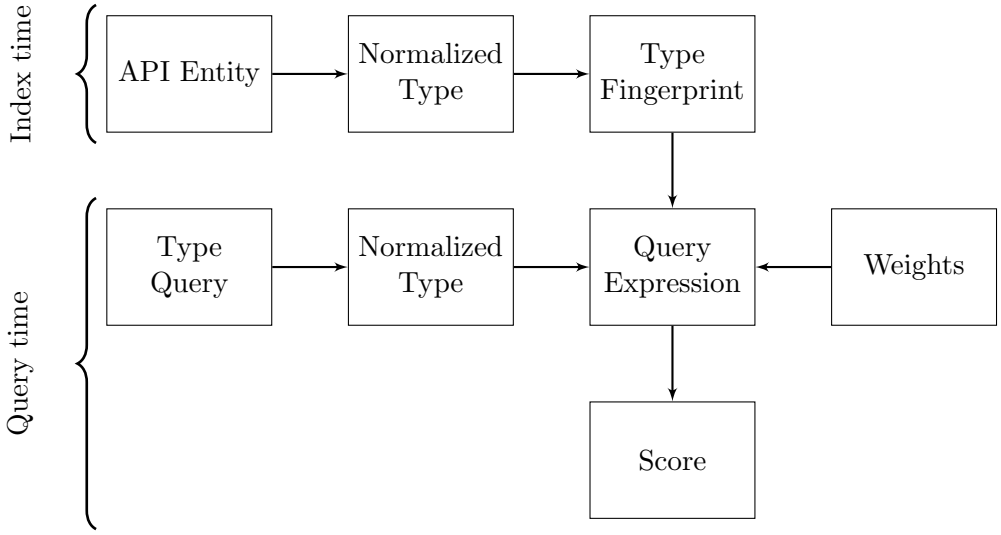

Figure 3.8.: Overview of the various data structures associated with the fingerprint evaluation model with the dataflow during index and query time

3.3. The Fingerprint Evaluation Model and Scala

As the fingerprint evaluation model has been designed with Scala's type system in mind, the model is already well suited for supporting the basic language features. Nevertheless, Scala offers some additional features that also have to be considered to successfully retrieve entities from Scala libraries. This section discusses those features and how they are integrated into the fingerprint evaluation model.

3.3.1. Implicit Conversions

Implicit conversions are a common feature in most programming languages. Typically, they are used to allow expressions of certain types to be used in a context that expects another type. For example, the declaration `val a: Long = 1` type checks successfully even though the expression `1` is of type `Int`. Compilers do often apply these implicit conversions if they can be performed without any loss of information.

We can identify, amongst others, four kinds of implicit conversions that are commonly used in various programming languages:

Widening Subtype Conversion
Given the types `S` and `T` such that $S <: T$, an expression of type `S` can be used wherever a type `T` is expected. This conversion is sometimes also referred to as "Widening Reference Conversion" and is a consequence of subtype polymorphism.

Widening Coercive Conversion
This is a conversion between the types `S` and `T` that do not have a subtype relation.

An implicit conversion can be applied if there is an injection from `S` to `T`. Hence, every value in `S` can be represented by a distinct value in `T`. For example, many languages provide implicit widening conversions for primitive types like from `Int` to `Long` or from `Long` to `Double` but not vice versa.

Adaptive Conversion

An adaptive conversion may be used if there exists a bijection between `S` and `T`. Hence, every value in `S` can be represented by a distinct value in `T` and vice versa. Java's automatic boxing and unboxing can be seen as an example of this conversion. For example, the primitive type `int` is implicitly converted to the reference type `Integer` if a reference type is required.

Pragmatic Narrowing Conversion

Some languages also provide destructive implicit conversions in some context. For example, Java allows to use any object as an operand to the string concatenation operator: In the expression `"hello " + new Subject("World")`, the right hand side of the operator is implicitly converted to a `String` by calling the `toString` method defined on `Subject`.

Scala offers a generalization of the latter three kinds that allows user to define custom implicit conversions [Ode14, section 7.3]. If a user writes an expression of a type `A` that does not match the expected type `B` of the current context, the Scala compiler searches the scope for a function or method defined with the `implicit` keyword of type `A => B`.

```
object BoolishConversions {
  implicit def int2bool(i: Int): Boolean = i != 0
}

object App {
  // brings int2bool into scope
  import BoolishConversions._

  val i: Int = 1

  // if expects a condition of type Boolean
  if(i)
    println("i is truish")
}
```

Listing 3.5: A user-defined implicit conversion from `Int` to `Boolean`

Listing 3.5 shows an example of a user defined implicit conversion from `Int` to `Boolean`. This conversion can be categorized as a pragmatic narrowing conversion. It may add some value for users as it allows a terser syntax for using integers as conditions. But it is narrowing because many values will map to `true`.

```
class IntOps(i: Int) {
  def isTruish = i != 0
```

```
}

object IntOps {
  implicit def int2intOps(i: Int) = new IntOps(i)
}

object App {
  // brings int2intOps into scope
  import IntOps._

  val i: Int = 1

  if(i.isTruish)
    println("i is truish")
}
```

Listing 3.6: Extending `Int` with a `isTruish` member

Another use of implicit conversions is to extend existing classes with new functionality as shown in Listing 3.6. In this example, the expression `i.isTruish` causes the compiler to look for an implicit conversion that converts an `Int` to a type with the member `isTruish`. This use of implicit conversions is frequently applied in the Scala Standard library and also known as the "Enrich-my-library" pattern.

Please note, that this section is only a very brief introduction to implicit conversions and omits many details of the language specification. First, there are further subtleties concerning how implicit conversions are resolved and how they are prioritized. And secondly, Scala defines additional implicit conversions that are hard-wired into the compiler: For example, narrowing conversions from `Int` literals to `Char` or `Byte` and the eta-expansion that converts method types to function types (see [Ode14, section 6.26] for a complete list).

In order to support implicit conversions during type retrieval, we leverage the strong mutuality of implicit conversions and subtyping. In fact, a stricter form of implicit conversion is discussed in type theory under the term *coercive subtyping* [LSX13].

From a less formalized perspective, implicit conversions often follow both interpretations of subtyping:

- The subset relation between values in the subtype S and values in the supertype T.
- Inheritance of the operations defined on the supertype T to the subtype S: Every operation applicable on a T is also applicable on a S.

E.g. the set of values of type `Int` is a subset of the values that can be represented by a `Long`, or `Int` $\subseteq$ `Long`. Accordingly, `Long` can be seen as a subtype of `Int`.

The inheritance interpretation of subtyping can be observed on the `int2intOps` conversion given in Listing 3.6. Through implicit conversion, all operations defined on `IntOps` become applicable on `Int` and therefore, we can conclude that `Int <: IntOps`.

Because `int2intOps` is a bijective coercion from `Int` to `IntOps`, the subset interpretation of subtyping also holds in this case.

As Scala also allows the definition of narrowing conversions the subset interpretation may also be violated by a user defined conversion. For example, `int2bool` in Listing 3.5 defines an implicit conversion despite `Int` $\not\subseteq$ `Bool`. Fortunately, this is not an issue when it comes to API retrieval because we are only concerned about the operations defined on a type. The values that inhabit a type, on the other hand, are not of concern.

Altogether, we use the following two measures to incorporate implicit conversions into our API retrieval model:

- An implicit conversion from `A` to `B` defines the synthetic subtype relation `A <: B`.
- The distance function discussed in section 3.2.6 is $\text{dist}(A, B) = 0.5$ if there is an implicit conversion from `A` to `B`.

The distance factor of 0.5 is relatively arbitrary but helps to ensure that members of the original type are listed below members from the converted type. Furthermore, 0.5 is lower than the distance to any direct sub or super type.

3.3.2. Implicit Parameters and Type Classes

With implicit parameters, Scala offers a relatively unique language feature. A method parameter defined with the `implicit` keyword can be automatically passed by the compiler if there exists an implicit value of a matching type in the callers scope. This is best explained by an example as given in Listing 3.7.

```
def check(i: Int)(implicit max: Int) =
  if(i > max) println("the number is too big")

{
  check(1) // error, no implicit value of type Int in scope
  check(1)(10) // OK, explicitly passing an implicit parameter
}

{
  implicit val threshold: Int = 100

  check(1) // OK, implicit value of type Int is in scope
  check(1)(threshold) // OK, equal to the above expression
}
```

Listing 3.7: Contrived example of a function requiring one implicit parameter

The function `check` defines one implicit parameter `max` of type `Int`. An according argument can now either be passed explicitly as in `check(1)(10)` or through an implicit value in scope (`threshold` in this case). The lookup of implicit values follows the same scoping rules as for other identifiers. Thus, implicit values can also be inherited or

imported. If no implicit value of a matching type is in scope, the compiler also considers the *implicit scope* of the expected type which includes, for example, the companion object of the expected type (a full overview of the lookup rules for implicit values is given in [Ode14, section 7.2]).

In summary, implicit parameters are a way to parameterize methods without too much boilerplate code. Since their introduction in Scala 2.0, language users found many ways to leverage implicit parameters:

Configuring methods with semi-global settings

Some methods depend on configuration parameters that can be shared between large parts of a program. For example, many operations on Scala's `Future` require an `ExecutionContext` that is able to execute asynchronous tasks. For most use cases the default execution context that executes tasks on a thread pool provides a suitable behavior. But in some cases, e.g. having many tasks that perform blocking I/O, a user might want to provide a different implementation. Because the execution context is provided through an implicit parameter, users of `Future` just have to import the appropriate execution context once and it is automatically passed when required.

Sharing a Context Variable

Sometimes, a computation requires a variable that describes the context in which it is executed and that has to be passed to every subsequent function call. For example, the logic of an HTTP request handler may call several functions that extract information from the current request:

```
def handle(req: Request) = {
  if(path(req) == "/") {
    respond("index.html", req)
  } else {
    val name = queryParameter("name", req)
    // ...
  }
}
```

Using implicit parameters, this can be simplified while still using purely functional methods:

```
def handle(implicit req: Request) = {
  if(path == "/") {
    respond("index.html")
  } else {
    val name = queryParameter("name")
    // ...
  }
}
```

The Type Class Pattern

The type class pattern is an answer to the problem of providing additional operations for a range of types without the need of directly modifying the type definitions. The pattern is inspired by Haskell's type classes but translates the language construct to object-oriented programming. It has probably been introduced the first time in [Ode06].

One example of a type class in the Scala standard library is `Ordering`:

```
trait Ordering[T] {
  def compare(a: T, b: T): Int
}
```

An instance of `Ordering[T]` provides a method `compare` that takes two `T`s and returns an `Int` that indicates whether both arguments are equal (0), the first argument is greater (a value > 0) or the second argument is greater (a value < 0). We can now define a generic method that depends on the `Ordering` trait:

```
def max[T](a: T, b: T)(implicit o: Ordering[T]): T =
  if(o.compare(a, b) >= 0)
    a
  else
    b
```

This method can be invoked with any type `T` for which exists an implicit instance of `Ordering[T]`. One interesting instance may be `Ordering[Int]`:

```
implicit val intOrdering = new Ordering[Int] {
  def compare(a: Int, b: Int) = a - b
}
```

Given this instance, we are able to use the `max` function on `Int`: `max(1, 2)` which the compiler will expand to `max(1, 2)(intOrdering)`.

One strength of this pattern is, that type class instances can nicely be composed:

```
implicit def pairOrdering[T, U](
    implicit ot: Ordering[T], ou: Ordering[U]) =
  new Ordering[(T, U)] {
    def compare(a: (T, U), b: (T, U)) =
      ot.compare(a._1, b._1) match {
        case 0 => ou.compare(a._2, b._2)
        case r => r
      }
  }
```

This definition provides an implicit `Ordering` instance on pairs `(T, U)` given there exists an according instance for both `T` and `U`.

Because type class instances are just values, it is also possible to transform existing instances:

```
def reversed[T](implicit o: Ordering[T]) = new Ordering[T] {
  def compare(a: T, b: T) = -o.compare(a, b)
}
```

To find the smaller of two pairs of int it is now sufficient to call

```
max((1, 2), (1, 1))(reversed)
```

which will be expanded by the compiler to

```
max((1, 2), (1, 1))
  (reversed(pairOrdering(intOrdering, intOrdering)))
```

Naturally, this pattern can also be applied in languages without support for implicit parameters as it helps to maintain a high reusability of components and high cohesion. In fact, the interface `java.util.Comparator<T>` is Java's equivalent to the `Ordering` type class.

Especially the type class pattern has gained some popularity and is frequently used in various Scala libraries. Also the Scala standard library defines, beside `Ordering`, several other traits whose use more or less resembles the type class pattern.

Technically, implicit parameters do not directly affect the working principle of our retrieval model. Types of implicit parameters can be incorporated into type fingerprints like any other parameter type. Hence, the fingerprint of

```
def max[T](a: T, b: T)(implicit o: Ordering[T]): T
```

becomes

```
-Any, -Any, -Ordering, /Unknown, +Nothing
```

Nevertheless, the frequent use of implicit parameters brings up two issues that affect how well a user's information need can be answered:

- Using implicit parameters for configuration reduces the accuracy of the length normalization factor. Implicit configuration parameters increase the length of an entity's fingerprint but they are unlikely to be considered by users formulating a search query. This may lead to a lower ranking for an entity with an implicit parameter that would otherwise achieve a top ranking because the same signature without the implicit parameter would perfectly match the query.
- Functionality provided via type classes is only discoverable if the user is aware of the according type class. For example, the `max` method mentioned above does not match a query like `(Int, Int) => Int`, despite the method can be used like a

method with exactly this signature given there is an instance of `Ordering[Int]` in scope. To successfully retrieve the `max` method a user has to search with a query like `(A, A, Ordering[A]) => A` which requires that he knows about the `Ordering` type class.

The first issue is currently addressed by choosing a sensible value for the length normalization weight that does not over-penalize entities with additional parameters. While this is in many cases sufficient, there is still some potential for a more sophisticated optimization. Another approach that did not result in the expected precision gain was to only incorporate fingerprint terms derived from non-implicit parameters into an entity's fingerprint length.

The second issue can be addressed by an additional transformation on API entities that is described in the next subsection.

Another approach that has been considered is to express type class implementation as subtyping. Hence, `Int` would be treated as a subtype of `Ordering`. In fact, Hoogle follows this approach to model Haskell's type classes [Mit11]. But this approach has some flaws when transfered to Scala and our retrieval model. Mainly, this approach collides with upper bounds on type parameters. Currently, a type parameter can be constrained by at most one upper bound which is used as a substitute for the type parameter at contravariant positions when transforming generic signatures to a proper type (see subsection 3.2.1). But type classes can introduce additional constraints that also have to be incorporated into the proper type.

3.3.3. Type Class Instantiation

As discussed in subsection 3.3.2, the frequent use of the type class pattern and similar techniques complicates the retrieval of certain API entities. To overcome this limitation we create additional synthetic API entities for every type class instance and every entity that expects an instance of the type class as an implicit parameter. For example, given the `Ordering` type class introduced in the previous subsection and the definitions

```
implicit val intOrdering: Ordering[Int] = ???
```

and

```
def max[A](xs: List[A])(implicit o: Ordering[A]): A
```

it is possible to derive a synthetic entity

```
def max(xs: List[Int])(implicit o: Ordering[Int]): Int
```

whose constrained type parameter `A` has been substituted by the type class member `Int`. With these synthetic entities added to the index, it is possible to answer queries like `List[Int] => Int` with a higher accuracy.

Because there is no need to create synthetic entities for non-generic implicit parameters, we use the term *type class* in the further discussion to address implicit parameters that can be substituted. Though, the term describes a slightly different concept to type classes in Haskell. Altogether, we use the following nomenclature:

Type Class
A type with at least one type parameter. E.g., `Ordering[T]` or `Eq[T, U]`.

Type Class Instance
An globally accessible implicit value, object or method of a type with at least one type argument that is not a type variable. E.g.,

```
implicit val intOrdering: Ordering[Int] = ???
```

defines a type class instance `Ordering[Int]`. But

```
implicit val i: Int = 1
```

does not define an instance because `Int` has no type arguments. Also,

```
implicit def list[T]: List[T]
```

dose not define a type class instance because `T` is a type variable.

Implicit parameter lists are ignored in the instance definition. Thus,

```
implicit def listOrdering[T](implicit o: Ordering[T]):
  Ordering[List[T]]
```

defines an instance `Ordering[List[T]]`.

If the instance definition is a method with at least one non-implicit parameter, it defines an instance of the eta-expanded type of the method.

```
implicit def opt2list[T](o: Option[T]): List[T]
```

defines an instance of `Option[T] => List[T]`.

Type Class Member
A type for which an instance definition for a type class exists. E.g., `Int` is a member of the type class `Ordering[_]`. If a type class uses more than one type parameter, an instance definition defines multiple members. Hence, an instance of type `Eq[Int, String]` defines `Int` as a member of `Eq[_, String]` and `String` as a member of `Eq[Int, _]`.

Every implicit and globally accessible entity defines potentially one or more type class instance. Because type classes can also extend other type classes, it is necessary to consider all base types as potential type of a type class instance. For example, given the following three type class definitions

```
trait PartialOrdering[T]
trait Equal[T]
trait Ordering[T] extends PartialOrdering[T] with Equal[T]
```

the implicit value

```
implicit val intOrdering: Ordering[Int] = ???
```

defines in total three type class instances `PartialOrdering[Int]`, `Equal[Int]` and `Ordering[Int]`.

Given the list of all types of the defined type class instances, it is now possible to instantiate all entities that use at least one of the according type classes. Instantiating refers in this context to substituting all type parameters of the entity by the according type arguments of the type class instance. E.g., instantiating the entity

```
def notEqual[A, B](a: A, b: B)(implicit eq: Eq[A, B])
```

with the type class instance `Eq[Int, String]` means to substitute all occurrences of `A` by `Int` and all occurrences of `B` by `String`. This results in the non-generic entity

```
def notEqual(a: Int, b: String)(implicit eq: Eq[Int, String])
```

It is also possible that the substitution results in additional type parameters. E.g. instantiating the `max` method described above with `Ordering[Map[K, V]]` results in

```
def max[K, V](xs: List[Map[K, V]])
  (implicit o: Ordering[Map[K, V]]): Map[K, V]
```

Despite type class instantiation has, for a lack of time to provide a stable implementation, not been included in the final version of our prototype, a partial implementation of this feature already yielded promising results.

3.3.4. Context and View Bounds

Because the type class pattern is such a frequently used language idiom, Scala offers with context bounds an alternative notation for briefly define type parameters that are bounded by an implicit parameter.

A type parameter `T` can be defined with one or more context bound which is written as a colon `:` followed by a bound `B`. This type parameter can then be instantiated only if there exists an implicit value of type `B[T]` [Ode14, section 7.4]. For example, the `max` function mentioned in subsection 3.3.3 can be declared more briefly with a context bound:

```
def max[T: Ordering](xs: List[T]): T = ???
```

This declaration is desugared by the compiler to

```
def max(xs: List[T])(implicit $evidence: Ordering[T]): T
```

whereas `$evidence` refers to a uniquely generated identifier.

View bounds are a similar notation to require that a type parameter has to be implicitly convertible to a given type. Although, view bounds will soon be deprecated in an upcoming release of Scala [Sca13].

A type parameter `T` can be defined with one or more view bounds which is written as `T <% B` where `B` is a type. This type parameter can only be instantiated if there is an implicit value of type `T => B` [Ode14, section 7.4]. The following method defines a view bound on the type parameter `T`:

```
def stringify[T <% String](x: T) = ???
```

which is equivalent to

```
def stringify[T](x: T)(implicit $evidence: T => String) = ???
```

It is also possible to mix lower and upper type bounds with view and context bounds in the same type parameter definition.

Concerning our API retrieval model, view and context bounds do not introduce additional complexity as they can always be expressed by an according implicit parameter. Thus, we will use the desugared representation of type signatures to build the index.

3.3.5. Variadic Parameter Lists

Scala allows to define method parameter lists with variable arity by appending the `*` modifier to the type of the last parameter in the list. For example,

```
def sum(is: Int*): Int
```

defines a method that accepts an arbitrary number of `Int` parameters. This method can either be called by providing zero or more `Int` arguments like `sum(1,2,3)` or by passing a subtype of `scala.Seq` followed by the `_*` type ascription [Ode14, section 4.6.2]:

```
val is = List(1, 2, 3)
sum(is: _*)
```

To support variadic parameter lists, we use a synthetic type `<repeated>[T]` to represent parameters defined with the `*` modifier. Hence, the normalized type of `sum` is `<repeated>[Int] => Int`. This ensures that both queries `Int => Int` and `Int* => Int` will match `sum`.

Furthermore, we want that queries like `List[Int] => Int` also match `sum` because `List[Int]`, as a subtype of `Seq[Int]`, is a valid argument where a `Int*` is expected. This is obtained by creating additional implicit conversions from all subtypes of `Seq` to `<repeated>`.

3.3.6. Summary

This section explained how some extended features of a particular type system can be integrated into our retrieval model. This is also an important requirement for targeting additional programming languages.

Note, that we purposefully omitted some features of Scala's type system that we do currently not represent in our model. As already mentioned in subsection 2.3.2, this includes *structural types* because they are rarely used in most Scala libraries. Furthermore, *higher kinded type parameters* are also not yet incorporated in the model. This was, at one hand, a necessity as their expansion to query expression may never terminate when implemented correctly. And, at the other hand, a pragmatic decision because we assume that the type class instantiation discussed in subsection 3.3.3 should cover most cases where a proper expansion of higher kinded types would improve search results.

Altogether, this chapter laid the conceptual foundation for our prototype implementation described in the next chapter. Furthermore, the evaluation of the retrieval model is discussed in chapter 5.

4. Implementation

This chapter describes our prototype implementation of the fingerprint evaluation model for the Scala programming language, written in Scala. It provides a web service called "Scaps", which is short for Scala API Search. Scaps allows users to invoke indexing jobs of Scala libraries and issue search queries against these indexed libraries.

4.1. Overview

The Scaps architecture is divided into various components as illustrated in Figure 4.1.

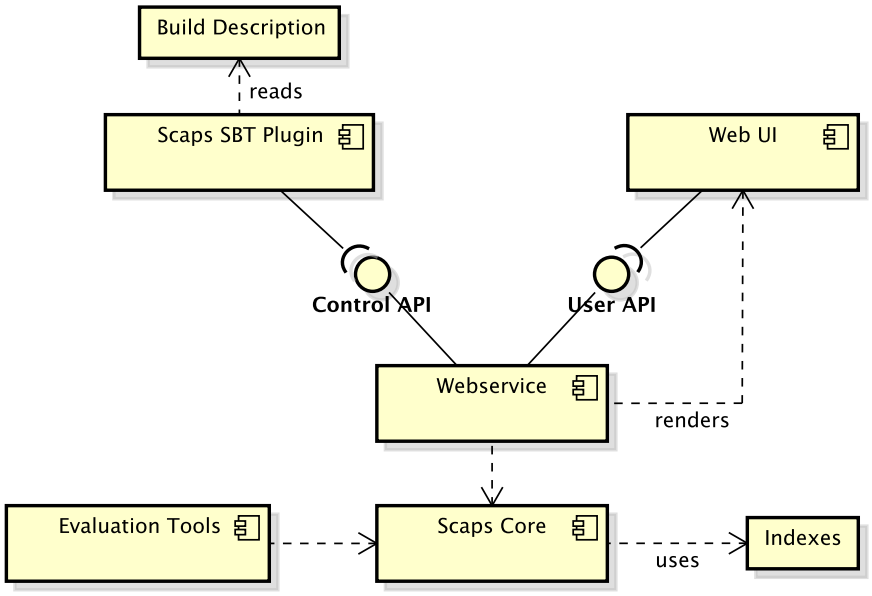

Figure 4.1.: High-level overview of the Scaps components

Scaps Core
: Implements the fingerprint evaluation model. The core library also provides methods to extract entities from Scala source files and for managing the Lucene indexes. Lucene is integrated into Scaps Core as a library dependency.

Evaluation Tools
: Provides command line tools for evaluating the search engine against a test collection of queries and relevant entities.

Webservice
: Exposes the Control and User APIs as HTTP services and manages an instance of the search engine. Also renders the Web UI and provides it as a web application.

Control API
: Defines methods for managing indexes. This includes the invocation of new index jobs and clearing the indexes.

User API
: Defines methods for querying indexes.

Scaps SBT Plugin
: A plugin for the Scala Build Tool (SBT) that is a client of the Control API. The plugin uses SBT's infrastructure for resolving project dependencies and fetching source files from repositories.

Web UI
: A website allowing users to query the indexed libraries.

This architecture has been chosen to create solution that can easily be extended and integrated into other environments. Thus, the core library may also be used to implement an Eclipse plug-in that can operate the indexes without depending on the SBT integration. Furthermore, the architecture allows to create additional clients for the Control and User APIs. A lightweight IDE integration of Scaps may just offer a search field that queries an online service without the possibility to index custom libraries.

The separation between Control and User API serves mainly as a replacement for a proper access rights management. This allows to expose the APIs on different ports such that access to the Control API can be restricted via network configuration. Hence, operations that are potentially harmful to the availability and functionality of the service are only exposed through the Control API. Uncritical operations, on the other hand, are provided through the User API.

4.2. Package Structure

From the architecture described above emerged the package structure shown in Figure 4.2.

`scaps.webapi` defines the Control and User APIs as well as the data model shared by all dependent packages. `webapi` is defined in a separate project and therefore available as a library dependency for other projects.

`scaps.featureExtraction` and `scaps.searchEngine` constitute the Scaps core functionality and are referred to as the Scaps core library. `featureExtraction` provides methods for extracting API entities from source files. `searchEngine` covers query processing and management of the Lucene indexes.

The `scaps.sbtPlugin` and `scaps.evaluation` packages represent the according components mentioned in section 4.1.

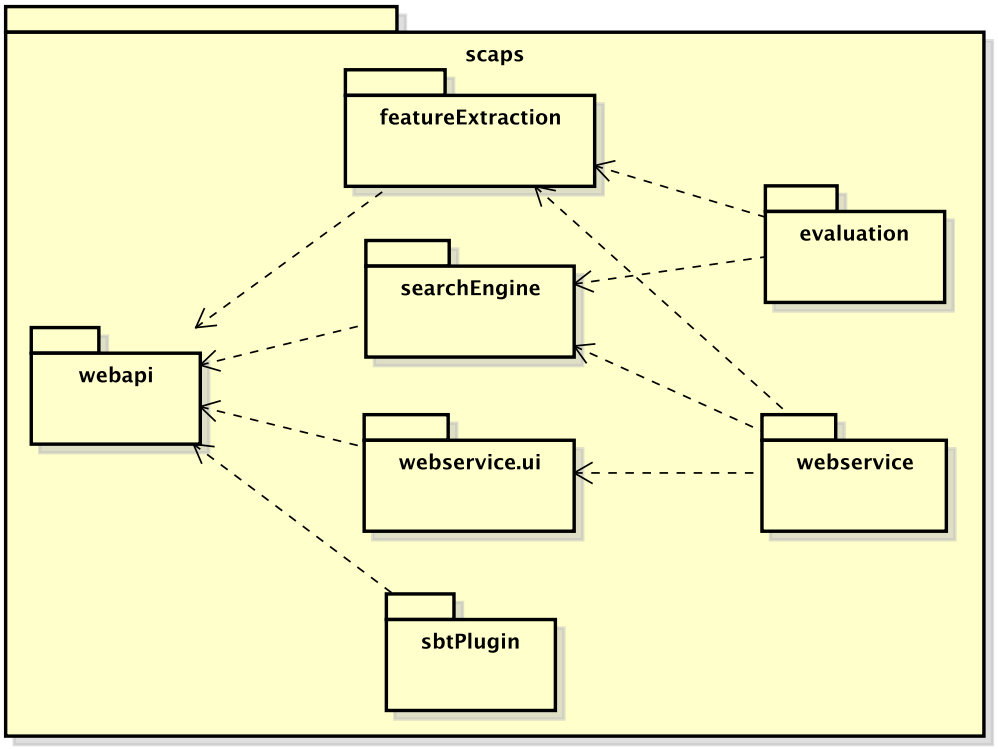

Figure 4.2.: Package dependencies in the Scaps project

`scaps.webservice.ui` is the implementation of the User API web client. Because we use the Scala.js compiler plugin to compile Scala code to executable JavaScript, it is possible to also reuse the data model defined in `webapi` on the web client.

And finally, the `scaps.webservice` package represents the Webservice component. `webservice` depends on `webservice.ui` as the service also renders the web client.

4.3. Data Model

Our data model has been designed to meet some particular requirements. It should be possible to represent types and API entities in a relatively language agnostic way. Hence, specific language features should be abstractable. E.g., whether a type has been defined by a class, a trait or an interface is not of concern for our retrieval model. Also, there is no need to distinguish between class members and global, static values or between values, methods and functions.

Furthermore, transformations of types should be easy to define, as these transformations are a central processing step of the retrieval model. Examples of such transformations are eta-expansions and type normalization.

Altogether, we need a notion of types and values in our data model together with the ability to represent the two fundamental type system concepts of subtyping and bounded

parametric polymorphism.

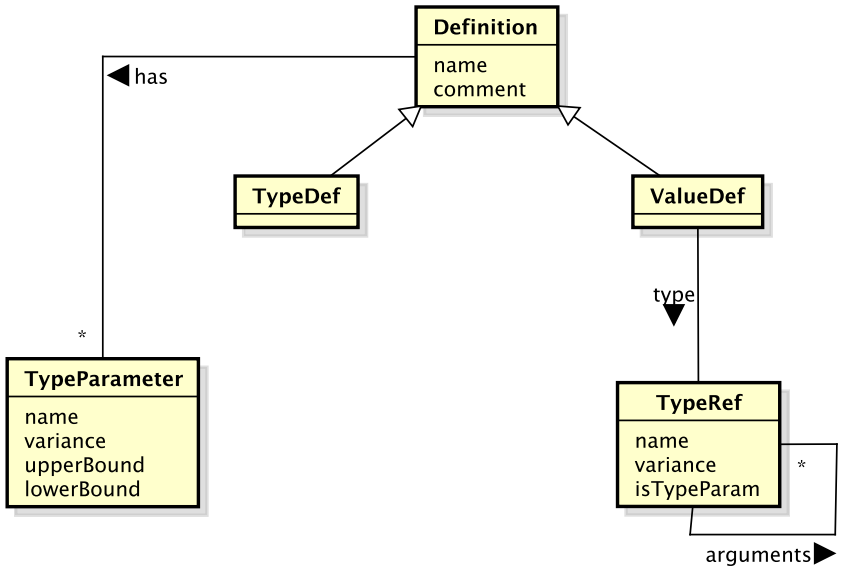

Figure 4.3.: Data model used by the Scaps components

As a result of these considerations we came up with a data model whose central entities are illustrated in Figure 4.3.

We consider an API as a collection of `Definition`s that define either a value (`ValueDef`) or a type (`TypeDef`). Each definition has a `name` and an optional documentary `comment`. Additionally, a definition may be parametrized by zero or more `TypeParameter`s.

Each value definition has an associated type that is represented by a `TypeRef`. A `TypeRef` is essentially a `name` that references a type definition or a synthetic type. Additionally, each `TypeRef` is annotated with its `variance` in the context it is used in. For example, the definition

```
val fn: Int => String = ???
```

results in a `ValueDef` with a type `Int => String`. In this context, `Int` is used contravariantly and therefore the `TypeRef` for `Int` will have an according `variance` value of `Contravariant`.

Furthermore, a `TypeRef` has zero or more type arguments which are again references to types. Hence, the above definition of `fn` is represented as

```
ValueDef("fn",
  TypeRef("=>", Covariant, [
    TypeRef("Int", Contravariant, []),
    TypeRef("String", Covariant, [])]))
```

A definition of a type simply states that there is a type with a certain name, an optional comment and zero or more type parameters. For example, the definition

```
class Set[A] { }
```

is represented as

```
TypeDef("Set", [
  TypeParameter("A", Invariant, "Any", "Nothing")])
```

As already mentioned, a definition may be parametrized by several type parameters. A `TypeParameter` has a parameter `name`, a `variance` annotation and an upper and lower bound. Bounds are identifiers referring to the name of the according type and defaulted to $\top$ or $\bot$ respectively. For instance, the definition

```
def flatten[A, C <: Iterable[A]](as: List[C]): List[A]
```

is represented as

```
ValueDef("flatten", [
  TypeParameter("A", Invariant, "Any", "Nothing"),
  TypeParameter("C", Invariant, "Iterable", "Nothing")],
  TypeRef("=>", Covariant, [
    TypeRef("List", Contravariant, [
      TypeRef("C", Contravariant, [])]),
    TypeRef("List", Covariant, [
      TypeRef("A", Covariant, [])])]))
```

As we can see, this representation is, in this case, only an approximation of the value's actual type. The upper bound of `C` is only defined as `Iterable` instead of `Iterable[A]`. An accurate representation would require full `TypeRef`s as lower and upper bounds of type parameters. We mainly decided to use this approximation because it heavily simplifies type parameter substitutions while yielding accurate results in most cases. A fully accurate representation of Scala's types would also require a vast reimplementation of the Scala type system. Luckily, type signatures like the one of the above `flatten` example are relatively rare in the Scala standard library.

4.3.1. Views

With mechanisms like implicit conversions (see subsection 3.3.1), languages may use additional subtyping like relations between types that are not defined within the type definition. Because these relations are not represented in `TypeDef`s, we use the concept of *views* to describe subtyping relations detached from type definitions. A `View` defines that a value of a certain type can be used when a value of another type is expected.

Figure 4.4 illustrates how we represent views in our data model. A `View` uses two `TypeRef`s to describe source and target types. This allows to represent complex subtype

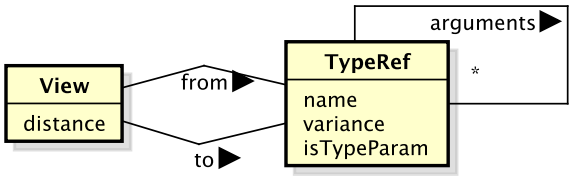

Figure 4.4.: Views as an abstraction of subtype relations

relations like `Map[Int, A] <: Int => A`. Additionally, we use a `distance` attribute to represent the distance factor discussed in section 3.2.6.

For example, a class definition like

```
class BitSet extends Set[Int] {}
```

results in a view

```
View(
  1,
  TypeRef("BitSet", Covariant, []),
  TypeRef("Set", Covariant, [
    TypeRef("Int", Invariant, [])]))
```

This view can be read as "A `BitSet` is viewable as a `Set[Int]` with a distance of 1". Hence, if a value of type `Set[Int]` is expected, one can also use a value of type `BitSet`. Note, that views describe possible alternative types at covariant positions. If we are interested in alternative types used at contravariant positions, e.g. function return types, we just need to reverse the direction of the view and flip the variance. For instance, the above view for contravariant positions is

```
View(
  1,
  TypeRef("Set", Contravariant, [
    TypeRef("Int", Invariant, [])]),
  TypeRef("BitSet", Contravariant, []))
```

This representation enables a mapping of subtype relations independent of the targeted language's subtyping mechanism. In the case of Scala, it combines both implicit conversions and inheritance.

4.3.2. Modules

If a project references multiple libraries, a user may want to limit the search scope to a subset of the indexed libraries. Hence, he uses a filter that specifies which libraries

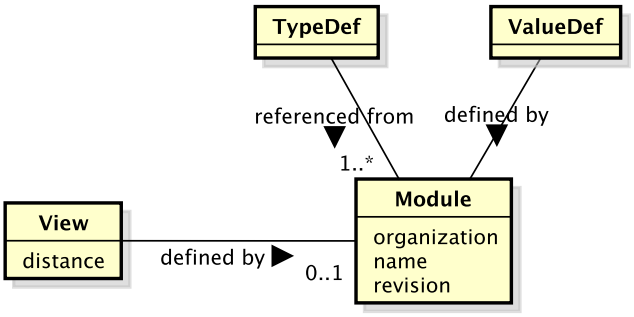

Figure 4.5.: Association of definitions and views with modules

should be included in the search. In order to support this feature, we use *modules* to represent libraries and other artifacts that define values. Analog to Maven artifacts, a module has an organization, a name and a revision.

Figure 4.5 shows how definitions and views are associated with modules. Hence, `ValueDef`s use an additional reference to the module that defines the according entity.

We distinguish between views that need a certain context to apply and views that are always active. For example, implicit conversions need to be imported to be applicable. In this case, the view is associated with the module that provides the realization of the view, e.g. the method defining the implicit conversion. Views that are always active, like subtype relations defined with the class definition, are not associated with a specific module.

4.3.3. Implementation of the Data Model

The actual implementation of the data model in `scaps.webapi` differs in some details from the model discussed in this section. Because the design of the retrieval model required a lot of experimentation with the available data from the Scala compiler, some data types include additional members that cover Scala specific aspects. Altogether, we decided to prioritize fast progress over providing a completely language agnostic implementation.

4.4. Feature Extraction

The `scaps.featureExtraction` package provides infrastructure to extract API entities together with type definitions and type views from programming artifacts. An extractor is a function from a file to a sequence of either entities (value, type and view definitions) or error messages.

Currently, we use two extractors. `ScalaSourceExtractor` for extracting entities from Scala source files and `JarExtractor` for extracting from jar-files containing Scala source code. The `JarExtractor` simply scans jar-archives for files with the suffix `.scala` and

then delegates the extraction of the source file to `ScalaSourceExtractor`. This composite design allows to add further extractors for targeting class-files containing Java byte code, documentation pages (which can also be packaged in jars) and Java source files.

Furthermore, the `ScalaSourceExtractor` requires access to an instance of the Scala presentation compiler. This is an asynchronous version of the Scala compiler running only the first few compilation phases up to and including type checking. To successfully resolve names, the presentation compiler needs to be provisioned with a class path including all binaries the source file in question is depending on. Thus, an indexing job should provide information on what has to be extracted (a file system path pointing to a jar-file) and how it can be extracted (the class path).

Note, that the current state of the source extractor is not yet capable of extracting all features of every valid Scala source file. For example, inherited doc comments from base classes are not yet supported. Also, we extract the raw doc comments and do not process Scaladoc annotations and the markdown syntax. Furthermore, there are issues with extracting some definitions that use higher-kinded or path-dependent types. These issues are partly caused by the presentation compiler and partly by incomplete specifications in the Scala source extractor. Because these extraction errors did not influence our retrieval results and mainly occur when extracting the Scalaz library, we decided to currently ignore these errors.

4.5. The Search Engine Facade

`scaps.searchEngine.SearchEngine` provides a facade that abstracts all operations on the index, such as indexing extracted entities and querying the index for definitions.

All operations are implemented blocking and either yield an exception or return with a consistent state of the search engine. Concurrent calls to any of these operations except `search` are prohibited.

Listing 4.1 shows how the search engine can be instantiated and fed with entities extracted from a Scala source jar. The factory method `SearchEngine.apply` can be used to obtain a search engine instance that persists indexes on the file system. Alternatively, `SearchEngine.inMemory` provides an in-memory instance with volatile indexes.

```
// build the search engine with default settings
val engine = SearchEngine(Settings.fromApplicationConf).get
// delete all indexed entities
engine.resetIndexes()

val sourceJar = new File("/path/to/source.jar")
val classPath = List("/lib/1.jar", "/lib/2.jar")

// obtain an managed instance of the presentation compiler
CompilerUtils.withCompiler(classPath) { compiler =>
  val extractor = new JarExtractor(compiler)
```

```
  // start the extraction
  val entityStream = extractor(sourceJar)
  // collect all entities while ignoring extraction errors
  val entities = ExtractionError.ignoreErrors(entityStream)

  // index all entities and associate them to an "unknown" default module
  engine.indexEntities(Module.Unknown, entities)
}
```

Listing 4.1: Extracting entities from a source jar and feeding them to the search engine

`CompilerUtils.withCompiler` provides an instance of the Scala presentation compiler. The first parameter list accepts the compiler configurations, like the class path. The second parameter list expects a callback that uses the managed compiler instance. After the callback returns, the compiler is disposed and all acquired resources are freed. `JarExtractor.apply` returns a lazy list of entities. This allows to continuously build the index while extracting entities but requires that the compiler instance remains open until `indexEntities` returns.

To search definitions in the index, the search engine can be used as shown in Listing 4.2. `search` returns either an error if a problem occurred while parsing and analyzing the query or a result set of the best matching definitions. The number of yielded results can be adjusted in the configuration file.

```
val engine = SearchEngine(Settings.fromApplicationConf).get

val query = "max: Int => Int => Int"

engine.search(query).get.fold(
  error => {
    println("An error occurred while parsing or analyzing the query")
  },
  results => results.take(10).map(_.signature).foreach(println))
```

Listing 4.2: Using the search engine for querying the index and printing the top 10 results

Additionally, we use Scala's `Try` monad to mark operations that may throw I/O exceptions. Thus, the complete return type of `search` is

```
Try[QueryError \/ Seq[TermEntity]]
```

This allows to separate between expected errors while processing user input and truly unexpected failures like corrupted index files.

4.6. Index

All logic interacting with the Lucene API is bundled in `scaps.searchEngine.index`.

4.6.1. Lucene

Apache Lucene is a search engine library with focus on high-performance and extensibility [Apa]. The core ranking model is based on the Vector Space Model but offers various abstractions to customize query processing, document matching, scoring and many other aspects of the retrieval model. This section introduces the major concepts used to adapt Lucene to our needs.

Indexes, Documents and Fields

A Lucene index is a collection of documents. A user can add documents to an index and remove or retrieve documents from it. A document is again a collection of one or more fields whereas each field associates a field name to a textual or binary value and some options on how the value is processed when the document is added to the index.

Some of the more relevant options on a field are:

`indexed`
: To retrieve documents based on the value of a field, the field must be defined with `indexed` set to true. Lucene derives a sequence of tokens from an indexed field by applying a specified analyzer. These tokens are then fed to the index and associated with the enclosing document.

`stored`
: A field's value may also be stored in the index. If the field is stored, Lucene persists a copy of its value such that it can later be retrieved with the document.

Often, it is sufficient to only index the fields of a document and just store a unique identifier of the according entity.

In contrast to many other database systems, Lucene uses no schema for describing the structure of the documents in an index. Hence, each document may have a distinct structure.

Analyzers

Analyzers are used by Lucene to extract tokens from indexed field values and queries. The input text is first split into a stream of tokens with a tokenizer. For example, the `WhitespaceTokenizer` splits the text at whitespace such that the input "Hello, world!" is transformed to the token stream `["Hello,", "world!"]`.

The initial token stream is then further transformed by a customizable filter chain. For example, a filter may strip punctuation marks from tokens, transform all characters to lower case to ensure case-insensitive search or remove some frequent stop words like "a", "an" and "the". Stemming filters are also available for various natural languages.

Queries

Besides a built-in query parser, Lucene also allows to programmatically compose queries. This requires programmers to ensure that keywords are accordingly analyzed but offers great flexibility on how distinct parts of the query are weighted and what type of queries are used.

A query basically implements two aspects of retrieval. At one hand, it matches documents based on field values. And, on the other hand, it calculates a score for the document that represents its relevancy to the query.

The following list introduces the query types that we use to retrieve entities from the indexes:

Term Query
: A query matching documents containing a certain term in a certain field.

Boolean Query
: A query matching documents based on one or more sub-queries. Each sub-query is associated with an operator that specifies how a match of the sub-query relates to a match of the boolean query. The *must* operator causes that the boolean query only matches if the according sub-query matches. The *should* operator is used to specify optional sub-queries that only contribute to the total score of the boolean query but do not affect the set of matched documents. And finally, the *must not* operator causes that the boolean query only matches if the the sub-query does not match.

  Altogether, the name *Boolean Query* has to be taken with a grain of salt as it is somewhat misleading. While boolean queries may be used to represent logical conjunction and disjunction on query terms (e.g. "car AND (travel OR rental)"), they do not strictly implement boolean logic.

Function Query
: A query matching all documents and scores them by applying a custom function. Typically, the function uses some fields of the documents as input and returns a single floating point number.

Match All Documents Query
: A query matching all documents with equal score. This query can be used to retrieve all documents from an index.

Custom Score Query
: Allows, like function queries, to manually implement the scoring logic but uses a sub-query to limit the set of matched documents.

Beside the various built-in query types, Lucene also allows to implement custom query types. This is necessarily if non of the offered query types provide the required matching logic. But implementing such a query exposes developers to many internal aspects of

Lucene. This requires a fundamental knowledge of the internal data structures and on how Lucene's optimizations can be leveraged.

Similarity

Lucene's `Similarity` class and its various sub-classes define how terms are weighted during retrieval. In simplified terms, `Similarity` represents the $\text{sim}(d, q)$ function used by retrieval models like the Vector Space Model as discussed in section 3.1. Implementations access Lucene's term statistics to efficiently calculate metrics like term frequency, IDF and doc length normalization at index or query time.

4.6.2. Indexes

We use a distinct index for each entity type that needs to be persisted. This ensures that all documents in an index have an identical field structure and simplifies some bulk operations on all entities of a certain type. Each entity index inherits from the generic `Index[E]` class that provides common functionality like managing Lucene's index readers and writers. Overall, we use the following indexes to persist and retrieve entities:

Value Index
: Persists instances of `ValueDef` and provides a `search` method that retrieves API definitions based on a processed user query.

Type Index
: Persists instances of `TypeDef`. Type definitions can either be retrieved by their full-qualified name or a suffix thereof. Hence, the type definition of `java.lang.String` can be retrieved by searching for "java.lang.String", "lang.String" or "String".

View Index
: Persists type views and provides an efficient method for retrieving sub and super types. This requires that `ViewIndex` knows the according typing rules which are currently hard-coded into the implementation.

Module Index
: Persists descriptions of all indexed modules.

Representative for the basic principles we use to store entities as documents in the indexes, Table 4.1 shows the document schema used in the value index.

In general, the entity represented by the document is serialized as JSON and stored in the `entity` field. When retrieving documents, the original entity can easily be restored by deserializing the JSON string to an instance of the according type. Other fields like `name` and `doc` are included in the document to support queries on the according values. These fields need not to be stored but are analyzed and indexed. Additionally, some data like `fingerprintLength` has to be accessible during querying to calculate document scores but are not used for document matching. These values are stored but not indexed.

Table 4.1.: Document schema used in the value index

Field	Description	Type	Stored	Indexed
name	Qualified name of the value	Text		✓
doc	Associated documentation	Text		✓
fingerprint	Individual terms of the fingerprint	Text	✓	✓
moduleId	Id of the defining module	Text		✓
fingerprintLength	Number of terms in the fingerprint	Numeric	✓	
entity	Serialized `ValueDef` entity	Text	✓	

4.6.3. Analyzing Names and Documentation

The conventions used for naming types and values in source code and documentation make some specific demands to the token analysis process. Identifiers should be split into the words they are composed of. For instance, the identifier `searchEngine.index.ValueIndex` is composed of "search", "engine", "index", "value" and "index". Furthermore, symbolic method names like `+`, `==` or `<*>` should also be retained during tokenization.

This led to the following rules to transform names and documentation of `ValueDef`s to a token stream:

1. Tokens are delimited by whitespace and `'.'` characters.
2. A token is split at changes from lower case to upper case. E.g. "ValueIndex" becomes "Value" and "Index".
3. A token is split at changes from alphabetical characters to non-alphabetical characters and vice versa. E.g. "int2string" becomes "int", "2" and "string".
4. All alphabetic characters are transformed to the according lower case character.

This behavior is implemented by chaining Lucene's built-in `WordDelimiterFilter` and `LowerCaseFilter` to a customized `CharTokenizer` that additionally uses the `'.'` character as a delimiter.

4.6.4. Retrieving Value Definitions

The `find` method of the value index accepts a preprocessed query containing a string of textual keywords and a type query expression as introduced in subsection 3.2.5. Furthermore, the search can be filtered by a set of module ids such that only values defined in these modules are returned. These input parameters are transformed to a single Lucene query which is run against the value index. The retrieved documents are then converted to a `ValueDef` by deserializing the stored `entity` value and returned to the caller.

The core of the generated Lucene query is an instance of the custom query type `TypeFingerprintQuery`. This query reads the stored fingerprint terms from documents

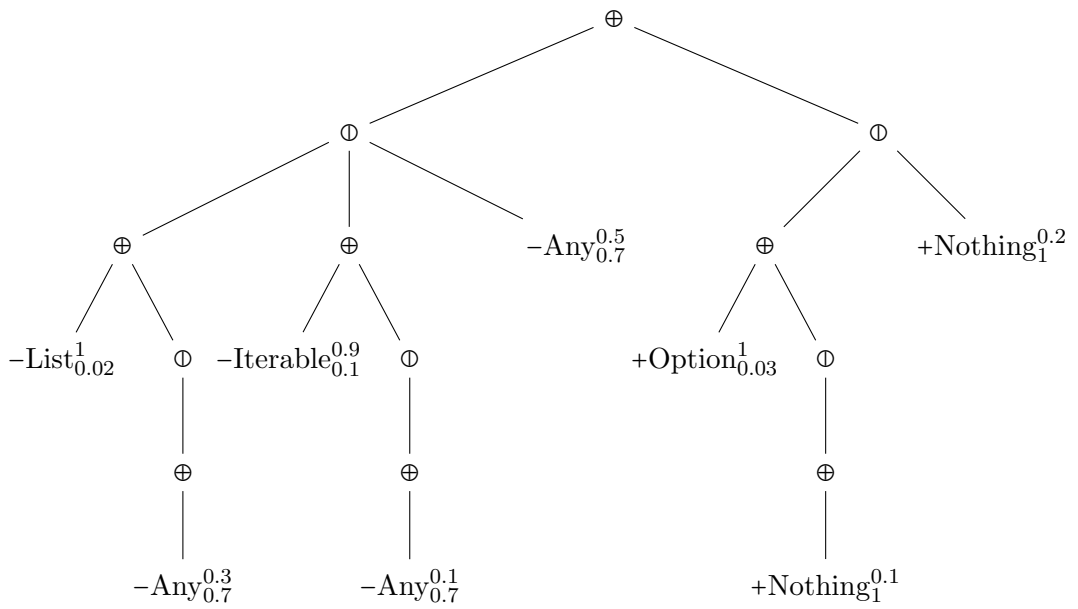

Figure 4.6.: Expression tree for the query `List[T] => Option[T]`; each leaf node's fingerprint term is annotated with the according weight (superscript) and type frequency (subscript)

and evaluates the query expression by applying the fingerprint. The resulting score denotes how well the document's fingerprint matches the type of the user query.

Because evaluating the query expression is a rather expensive operation, the number of scored documents by the type fingerprint query should be limited. This is achieved by using a sub query for matching documents that are likely to result in a high fingerprint score or whose `name` or `doc` field contain a keyword from the user query. Only documents matched by this sub query are scored by the type fingerprint query.

The heuristic used to filter documents that are likely to achieve a good fingerprint score is called *fingerprint frequency cutoff*. The idea is to only match documents that contain a fingerprint term with a low type frequency that has a high impact on the total score. Given the example expression tree in Figure 4.6, the fingerprint term `-Any` has a relative high type frequency. A query including `-Any` potentially matches 70% of the value definitions in the index and its specificity is therefore rather low. Furthermore, the highest possible score contributed by `-Any` is 0.3 which is relatively low compared to other nodes like `-List` or `+Option`. As a consequence, documents containing `-List` are more likely to achieve a good score than documents containing `-Any` but none of the other terms.

To find the fingerprint terms in the query expression that have a low type frequency but a high weight, all fingerprint terms of the query expression are ordered by descending weight. The longest prefix of this list with an accumulated type frequency below a certain threshold is then used to build the matcher query. The threshold is a configurable

parameter called `fingerprintFrequencyCutoff`.

Applying this heuristic to the example expression tree in Figure 4.6, we get the following list of ordered fingerprint terms:

$$-\text{List}^{1}_{0.02}, +\text{Option}^{1}_{0.03}, -\text{Iterable}^{0.9}_{0.1}, -\text{Any}^{0.5}_{0.7}, +\text{Nothing}^{0.2}_{1}$$

Using a fingerprint frequency cutoff of 0.2, the resulting fingerprint terms are then `-List`, `+Option` and `-Iterable` as the accumulated type frequency is 0.15 and including `-Any` would exceed the threshold.

This approach ensures that only a small fraction of the indexed value definitions needs to be evaluated against the type query expression. With a cutoff of 0.2, at most 20% of the index has to be evaluated.

4.6.5. Fingerprint Evaluation Algorithm

As mentioned in subsection 3.2.7, evaluating a type query expression against a type fingerprint is a relatively costly operation. In this section, we explain the heuristic used in our implementation and why it is a good approximation to an optimal algorithm.

The core of the algorithm is a `scoreTerm(e, t)` function that returns the best possible score when evaluating a query expression `e` against a single fingerprint term `t`. This is achieved by locating the matching leaf with the highest associated score. Additionally, `scoreTerm` returns a copy of the expression tree `e` pruned accordingly to the following rules:

- A leaf node associated with the term t_l is removed if $t == t_l$.
- An inner node is removed when the number of childrens is 0.
- If t matches a child node c_i of a max node $\text{⓪}(c_1, ..., c_n)$ with the best possible score, all child nodes except c_i are removed.

For example, applying `scoreTerm` with the expression shown in Figure 4.6 and the term `-Iterable` yields the score 0.9 and modifies the tree as illustrated in Figure 4.7. Because `-Iterable` is a descendant of the ⓪* node, the third rule applies and all other child nodes including their descendants are removed.

A naive algorithm `scoreFingerprint(e, f)` to evaluate a complete fingerprint `f` would be to sequentially apply the fingerprint terms to `scoreTerm` and accumulating the resulting scores. Unfortunately, this approach does usually not yield the best possible score and is dependent on the order in which the individual terms are processed.

A better approach is to apply the fingerprint terms ordered by their maximal achievable score. Hence, a term that may contribute more to the total score is applied before terms with less potential. To get the potential score of each term, we first apply each term $\texttt{t}_\texttt{n}$ to `scoreTerm(e,` $\texttt{t}_\texttt{n}$`)` with the original expression tree. This potential score then determines the ordering of the fingerprint terms in `f` before applying `scoreFingerprint(e, f)`.

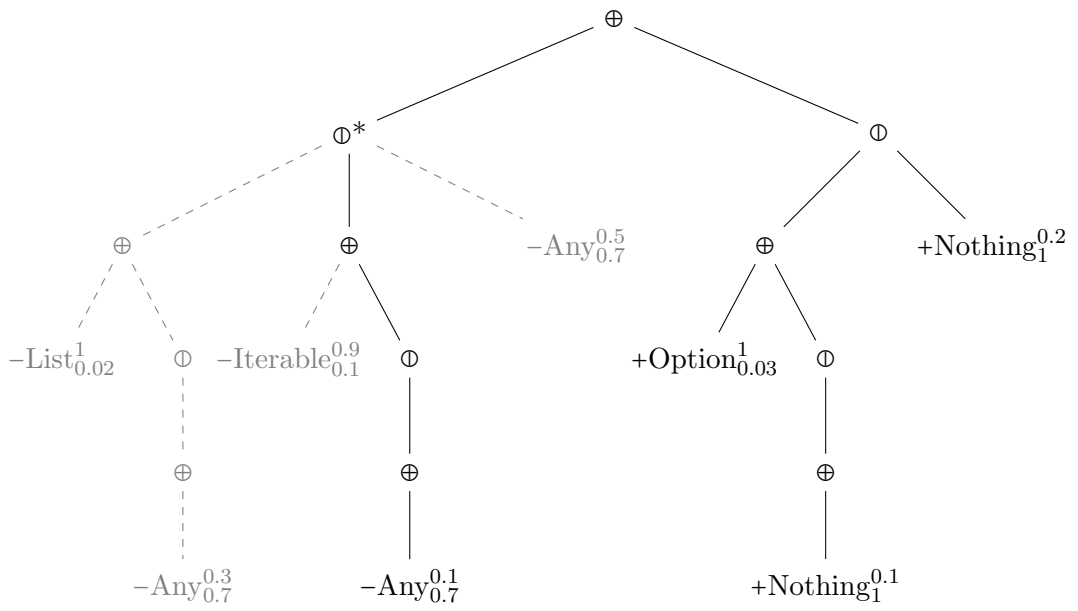

Figure 4.7.: Expression tree as shown in Figure 4.6 after applying `-Iterable`; pruned nodes are shown grayed out and with dashed edges to parent nodes

This heuristic proved to be sufficient to evaluate expressions as produced by our query analyzer. The reason that this approach yields the optimal result in most cases is that the query analyzer does not create arbitrary expression trees. Instead, the weights associated with each leaf are generated with a certain pattern:

> Given an expression $\odot(\oplus(c_{1,1}, ..., c_{1,m}), ..., \oplus(c_{n,1}, ..., c_{n,m}), d)$, for each sub expression $\oplus(c_{k,1}, ..., c_{k,m})$ there is a node $c_{k,l}$ with a higher weight than the weight associated with d.

As long as this property holds for every sub tree of the processed expression, the algorithm will prune d if there is an alternative $\oplus(c_{k,1}, ..., c_{k,m})$ that achieves a total higher score than d. This property can also be observed in the example tree in Figure 4.6. The alternative $-\text{Any}^{0.5}_{0.7}$ will be pruned if the processed fingerprint contains `-List` or `-Iterable`. If the property does not hold and $-\text{Any}^{0.5}_{0.7}$ has a weight of 0.95, a fingerprint `-List, -Any, +Nothing` would be processed with the order `-Any, -List, +Nothing` which would yield a non-optimal score of 0.95 + 0.2 = 1.15.

4.7. Query Analysis

The transformation from a user's query to a complete type query expression that can be processed by the fingerprint evaluation algorithm requires several intermediate processing steps. A complete overview on these steps is given in Figure 4.8 and the implementation, including the intermediate data structures, can be found in `scaps.searchEngine.`

`queries`. Note, that this section only describes the processing of the type part of the query. Textual keywords are passed without any transformation to Lucene's tokenizer.

The query analysis process starts with parsing the query string. The `QueryParser` separates textual keywords and the type signature. The type signature is additionally transformed to a type tree where each node has a type name and zero or more type arguments. Syntactic shortcuts for function and tuple types, like `A => B` and `(A, B)`, are also parsed accordingly.

The parser implementation should give users flexibility on how a query is formulated but should also be able to report syntax errors in query types. Hence, a query like `read file utf8` or `>:>` should be recognized as a sequence of keywords, `Int => String` on the other hand is a type query and `try parse: String => Int` contains both keywords and a type. Furthermore, queries with invalid type name like `List[A` or `(A => B` should result in a parsing error. The query `max` can represent both a type or a keyword. In this case, we first treat it as a type and try to process it accordingly. If the query analysis yields a resolution error, the query is processed again as a keyword query.

The raw type query yielded by the parser is then passed to the `QueryAnalyzer`. The first processing step is to resolve type names with the type definitions provided by the `TypeIndex`. Incomplete type identifiers like `Int` are resolved to their full-qualified name (`scala.Int`). If the name is ambiguous or there is no according type definition, an error is returned. To reduce ambiguity errors, we prioritize type definitions in the root namespace `scala` of the Scala standard library. Furthermore, the resolution step checks if the defined number of type parameters matches the number of type arguments in the query. Hence, `Map[Int]` results in an error because `Map` is defined with two type parameters. If the query omits the type argument list, wildcard types are inserted accordingly. For example, `Map` will be interpreted as `Map[_, _]`.

The next step is to transform the resolved query to a valid type reference (an instance of `TypeRef`). This requires that each type in the query is also associated with its variance at the according position. For example, `Map[Int, String] => Int` will become `+Function1[-Map[/Int, -String], +String]` if `Map` is invariant over the first type parameter and covariant over the second type parameter.

The type reference is then normalized with the same function as the type of value definitions are normalized before added to the index. For example, function types are transformed to their curried representation: `(A, B) => C` becomes `A => B => C`.

This normalized type is then expanded to a unweighted query expression. For each partial type, the type hierarchy is searched for sub or super types according to its variance (see subsection 3.2.5). The alternative types are provided by the `ViewIndex`. During the expansions, the various weighting factors like depth, distance and type frequencies are retained in the unweighted query expression.

In order to reduce the complexity of the expression and decrease the runtime of the evaluation algorithm, the expression tree is minimized in the next step. The minification function performs the two following optimizations:

- Elimination of unnecessary nodes. Sub expressions like $\oplus(\odot(x))$ are rewritten to x.

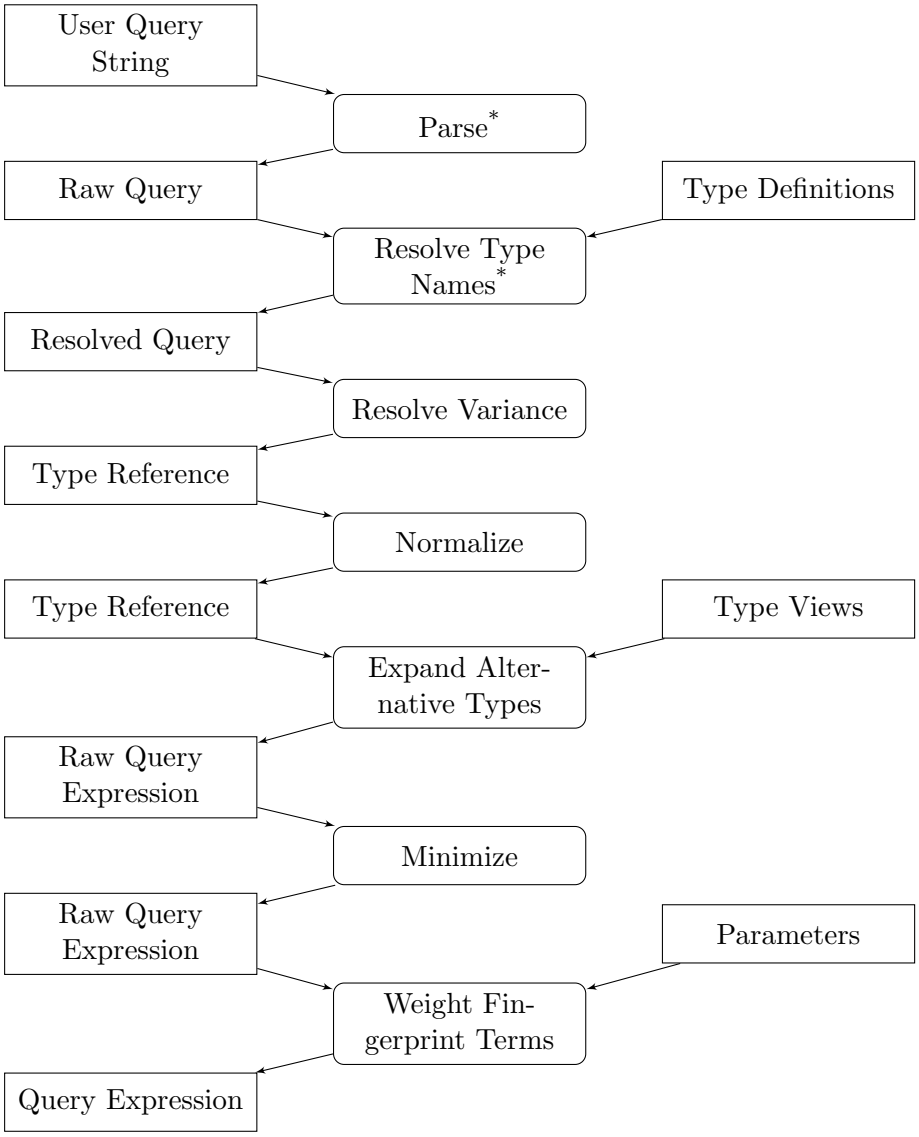

Figure 4.8.: Overview on the query analysis process from a textual user query to a fully resolved and expanded query expression; processing steps are illustrated with rounded corners; rectangular shapes represent input and output data; processing steps that may also return an error are annotated with *

- Out-factorization of common sub expressions. Because query expressions are distributive over $\oplus$, expressions like $\otimes(\oplus(a, c), \oplus(b, c))$ can be rewritten to $\oplus(\otimes(a, b), c)$.

The final step is to calculate the weight associated with each leaf node in the expression tree. As discussed in section 3.2.6, this depends on the parametrized weights of each individual factor.

4.8. Web Service

The Scaps web service uses the core library to manage an instance of the search engine. The search engine is exposed over an HTTP API and by a simple user interface that supports user queries to the search engine. The web service uses the Spray toolkit [Typb] to integrate the search engine in an HTTP server. Spray is a collection of lightweight libraries including an HTTP server implementation and a domain specific language (DSL) for request routing.

4.8.1. HTTP API

The HTTP API provided by the web service exposes the Scaps User and Control APIs over HTTP. User and Control API are Scala traits defined in the `scaps.webapi` package.

The user API defines a `search` method accepting an unparsed query string and a set of module filters and an `assessPositively` method that accepts a positive user feedback about a specific search result. The control API defines a single `index` method that accepts several index jobs with modules to be indexed. Invoking `index` will start the indexing process. After the process completes, the index used by the search engine will only consists of the modules provided in the arguments to `index`. Hence, `index` replaces previously indexed modules. Additionally, both APIs define a `getStatus` method that returns information about what modules are indexed and what jobs are currently processed.

The HTTP API is implemented by using the Autowire [Haoa] library for Scala. Autowire provides macros that enables remote procedure calls (RPC) between Scala systems without depending on runtime reflection. Furthermore, while Autowire is agnostic to the underlying transport protocol, it does not abstract away the fact that calls to the API are made over an unreliable network connection with potentially high latency. The reason we decided to use an RPC based API instead of a RESTful web service is that Autowire requires no boilerplate code to map HTTP requests to method calls. This ensures that changes to the API and the underlying data model can be quickly applied.

4.8.2. Web Client

The web client provides a minimalistic user interface for querying the index. As shown in Figure 4.9, the UI is composed of a text field for entering a query, a module filter for restricting the search to certain modules and the result set. Each result consists of

the definitions signature, its defining module, its fully qualified identifier and a link that allows user to confirm that this result is relevant to their information need.

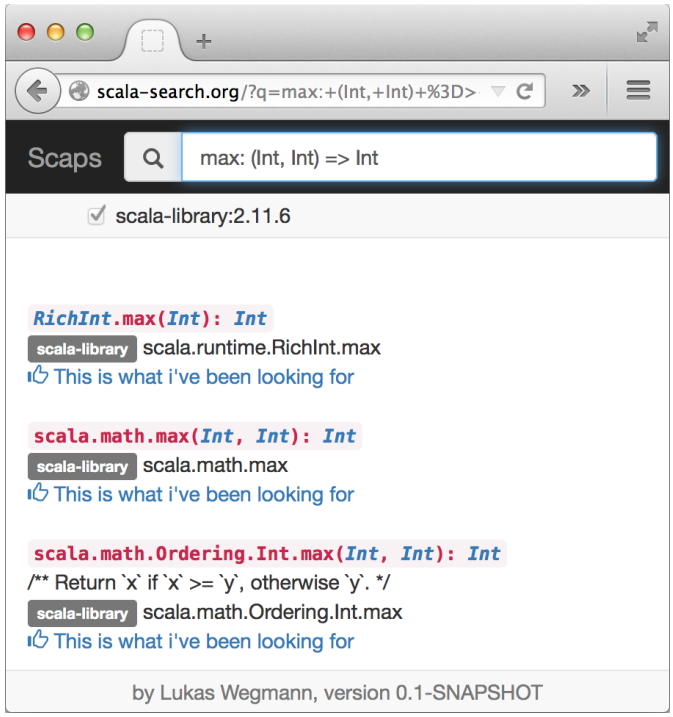

Figure 4.9.: Screenshot of the Scaps web client

The web client is written using ScalaTags [Haob] and Scala.js [Doe]. ScalaTags is an internal Scala DSL for defining HTML templates. Instead of using a dedicated templating DSL with additional syntax rules, ScalaTags allows to use the familiar Scala language constructs. Scala.js on the other hand, is a plug-in for the Scala compiler that adds another backend emitting JavaScript instead of Java byte code. By using Scala.js, the code executed in the browser can be written almost exclusively in Scala. Also many third-party Scala libraries are available for Scala.js, including Autowire and ScalaTags. Compared to other JavaScript cross compilation projects, Scala.js is no framework abstracting over the browser's DOM API and allows to directly call native JavaScript functions.

The following reasons lead to the decision to use a Scala-only architecture: First, we wanted to gain some personal experience with these tools as they promise some interesting approaches to web development. And second, it seemed that this architecture is highly suitable for fast development. Because all layers of the software are written in Scala, the IDE is capable to type check all parts of the project and automated refactorings are also applied on the client code. Furthermore, it reduced the required amount of code, because domain logic can easily be shared between client and server.

In retrospect, ScalaTags and Scala.js turned out to be good choices for this project.

Especially the functional nature of ScalaTags helped to simplify the complex transformations from result sets to HTML nodes. Because the final version of the prototype included less features requiring dynamic code on the client than initially anticipated, Scala.js lead to a slightly over-engineered solution that does not yet leverage the full potential of the cross-compilation step. Nevertheless, using Scala.js involved surprisingly few complications, despite its relatively early development state.

4.8.3. Concurrency

To achieve a good response time when multiple users simultaneously access the Scaps service and to support queries while the index is being rebuilt, a good approach to concurrency is inevitable. As Spray already makes use of the Akka actor library [Typa], we decided to also leverage the actor model [AH85] to handle concurrent calls to the search engine.

The resulting actor system is fairly simple and consists of the following components:

Searcher
: An actor processing one search query.

Index Worker
: An actor processing an index job.

Director
: An actor orchestrating searchers and index workers and manages instances of the search engine facade (see section 4.5).

Scaps API Implementation
: Converts calls to the Control and User APIs to messages that are passed to the search engine actor.

The core of the actor system is the director that receives messages from the API implementation. When it receives a `Search` message, it creates a new searcher with an instance of `SearchEngine` and relays the message to the new actor. The searcher processes the query by using the blocking `find` method on the instance of the search engine. As soon `find` returns the search result is sent to the API implementation which returns it to the caller of the API. Finally, the searcher is terminated by sending a `Kill` message to itself. This message flow is also illustrated in Figure 4.10.

A similar message flow is implemented for `Index` messages that consists of modules to be indexed: In this case, the director creates a new index worker with a new instance of `SearchEngine` and relays the message to this worker. The director confirms that the index job has been started by replying to the API implementation with an according message. Additionally, the director transits to the `Indexing` state. In this state, all further `Index` messages are rejected. When the index worker successfully completed the index job, it sends an `Indexed` message to the director including a reference to the `SearchEngine` on which the index has been built. The director can now transit back to

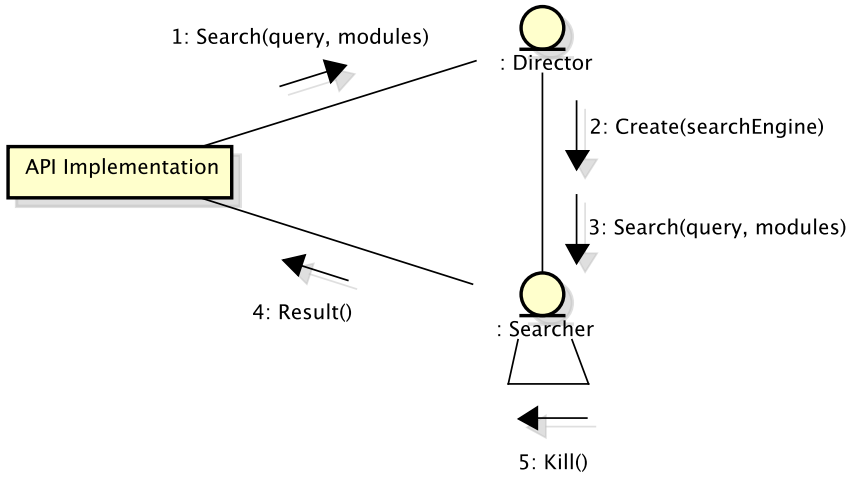

Figure 4.10.: Message flow after `search` has been invoked on the Scaps API

the `Ready` state and dispose the old search engine. From this point in time, all search queries are answered with the updated search engine.

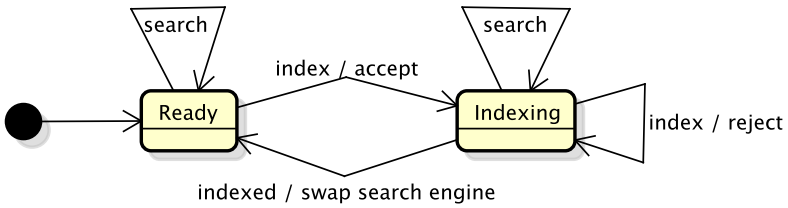

Figure 4.11.: States of the director actor

Figure 4.11 illustrates the two possible states of the director. As shown, search queries can be answered in both states. In the `Ready` state, there exists only one instance of `SearchEngine` that is used to answer all queries. As mentioned in section 4.5, concurrent reads on the indexes are not an issue. In the `Indexing` state, the existing search engine is used to answer queries and the index is being built with a new instance.

4.9. SBT Plug-in

The Scaps SBT plug-in integrates a client to the Scaps Control API into the Scala Build Tool (SBT). This allows the use of SBT to resolve and fetch dependencies of libraries that should be indexed. Furthermore, SBT provides the correct class path required by the presentation compiler to extract entities from the source files.

The SBT plug-in provides the following tasks that can be executed in the SBT console:

scaps
: Queries the Scaps service with `<query>` and prints the top three matching type signatures.

scapsStatus
: Requests & Displays the current status of the service.

scapsModules
: Yields the modules that will be indexed based on the `libraryDependencies` defined in the build settings.

scapsIndex
: Requests the service to index the modules yielded by `scapsModules`.

Additionally, each task uses one of the following settings that should be defined in the build settings:

scapsHost
: Host name of the Scaps User API, defaults to `localhost:8080`.

scapsControlHost
: Host name of the Scaps Control API, defaults to `localhost:8081`.

Because the plug-in includes the local file system paths in the index jobs sent to the Scaps service, `scapsIndex` has to be executed on the same machine that also hosts the service.

4.10. Architectural Insights

As already mentioned, we mainly focused on fast development and easy experimentation with the Scala presentation compiler and our retrieval model while implementing the Scaps service. This resulted in some pragmatic design decisions that may complicate future extensions.

First, integrating feature extraction into the core library and delegating the extraction process to the Scaps web service is to some extension a violation of the separation of concerns principle. An improved design would assign the extraction process to the clients of the Control API. For example, the SBT plug-in extracts all available definitions and sends the list of value, type and view definitions to the Scaps service with the index request. Because the SBT plug-in knows all compiler settings like class-path, enabled compiler plug-ins and flags, it should be responsible to interpret the source files. This design would also simplify incremental indexing of modified source files. This is an important requirement for integrating Scaps into IDEs. Another advantage would be that the SBT plug-in no longer needs to be executed on the server that hosts the web service. API entities could simply be transfered over the network.

The reason we did not use this design is mainly the fact that SBT is currently being built against Scala 2.10. This complicates the integration of the Scala 2.11 presentation compiler into the plug-in to some extent.

And second, the implementation of the retrieval model is not completely isolated and contains Scala specific logic. Though, a more language agnostic implementation would simplify migrations to future versions of Scala and to other programming languages. A clean separation of the retrieval model should theoretically be possible. The core model can be reused between programming languages. The language specific aspects that needs to be factored out are:

1. How can values, types and views be extracted from source files?
2. How can types be normalized?
3. What is the syntax of the queries?
4. How should search results be presented?

If the first aspect would be delegated to the Control API clients as mentioned above, it would already be separated. Because type normalization is also used during the query generation, the language specific normalization rules must also be available by the core library. Furthermore, language specific query parsers and presentation logic may be convenient but should not require major changes to the architecture.

4.11. Evaluation Tools

Finally, `scaps.evaluation` provides the tools used to evaluate the Scaps core library. The package implements two main classes `Benchmark` and `Evaluation`.

The former calculates the evaluation scores for a fixed configuration. This class is mainly used to perform regression tests on the continuous integration (CI) server. For all commits, `Benchmark` is automatically executed and the achieved scores are reported to the CI output. This allows to quickly detect changes that accidentally damage the effectiveness of the search engine. Also, changes that are supposed to improve the evaluation scores can easily be verified. This setup proved to be highly helpful. It allowed, together with the unit test suit, to quickly apply changes and precisely track the consequences (see section E.3).

The second main class `Evaluation` implements the evaluation of our retrieval model and the comparison between various instantiations of the system. This class has also been used to search for a good parametrization of the search engine by accordingly modify the definitions of the various test runs. The evaluation of the retrieval model with the compared systems and the collected measurements is discussed in detail in the next chapter.

5. Evaluation and Parametrization

In chapter 3, we presented various techniques, like depth and distance boosts, that should improve the performance of our retrieval model. Unfortunately, there is no formal proof that supports the effectiveness of these techniques or the effectiveness of the basic ideas of the fingerprint evaluation model. Therefore, we need empirical data to decide which techniques are beneficial and how to configure the parameters of the model.

The evaluation of our retrieval model should mainly give answers to the following questions:

- How effective is the API retrieval model to answer information needs?
- Can the system be improved with the various modifications proposed in section 3.2?
- What is the optimal parameter configuration?

This chapter introduces the measurements and test setup provided to answer these questions and presents and discusses our results.

5.1. Measures of Effectiveness

Because we address API retrieval like an information retrieval problem, we can use the same measurements as traditionally used in IR to evaluate the effectiveness of our solution.

Two simple measures of the effectiveness of an IR system with unranked result sets are precision and recall. In our case of a ranked result set, we have to use some more sophisticated measures that are extensions of these two basic notions of effectiveness.

The precision of an IR system for a certain query is the ratio between the number of retrieved documents that are relevant to the query (true positives) and the total number of retrieved documents:

$$\text{Precision} = \frac{|\text{relevant retrieved}|}{|\text{retrieved}|}$$

Additionally, recall is the fraction of relevant documents that have been retrieved with a query:

$$\text{Recall} = \frac{|\text{relevant retrieved}|}{|\text{relevant}|}$$

Hence, if we identified a total of 5 documents that are relevant to a query and the IR system yielded 4 of those documents, the resulting recall is 4/5 = 0.8. This implies

that a system returning all documents in the collection will always achieve a recall of 1. Hence, a high recall alone will not suffice to state that an IR system is effective.

Retrieval systems are usually ranked. Hence, a search engine yields a list of results ordered by the assumed relevancy to the user's query. The result set contains a vast number of documents that are only browsed by users until no more relevant results show up or the information need has been answered. If we apply the precision and recall measures to such a system, recall will usually tend towards 1 and precision towards 0.

5.1.1. Mean Average Precision (MAP)

A useful tool to reason about the effectiveness of a ranked retrieval system is a precision/recall graph [MRS08, p. 145]. This graph plots the precision and recall values for each rank of the result set. Hence, the n-th data point is precision and recall of the n first results yielded for a certain query.

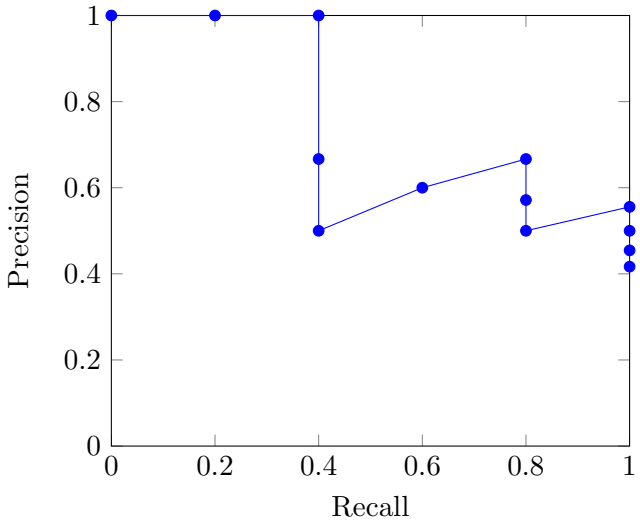

Figure 5.1.: Precision/recall graph for the query `Ordering[Char]`; The first data point marks the precision/recall at rank 0

Figure 5.1 shows a typical precision/recall curve of our retrieval system for the query `Ordering[Char]`. This query should yield available implementations of the `Ordering` type class for `Char`. We identified a total of 5 relevant entities in the Scala standard library for this query and our implementation yielded these relevant results at the ranks 1, 2, 5, 6 and 9.

A good indicator for the accuracy of the query results is the area under the precision/recall curve. An ideal system would draw a precision/recall curve going straight from $(0, 1)$ to $(1, 1)$ with an area of 1. The worst possible system, that does not retrieve the relevant results at all, would draw a precision/recall curve with an area of 0, all points would be clustered at $(0, 0)$. This indicator is also referred to as *average precision* and is defined as

$$\mathrm{AP} = \int_0^1 p(r)\,\mathrm{d}r$$

Hence, the integral of the precision p at the recall level r from 0 to 1. A more practical, but equivalent, interpretation is

$$\mathrm{AP} = \frac{\sum_{k=1}^{n} P(k)\mathrm{rel}(k)}{|\mathrm{relevant}|}$$

whereas n is the number of retrieved documents, $P(k)$ is the precision at rank k and $\mathrm{rel}(k)$ is 1 if the k-th result is relevant or 0 otherwise [TS06].

The *mean average precision* (MAP) refers to the arithmetic mean of the average precision of all queries Q in a test collection:

$$\mathrm{MAP} = \frac{\sum_{q \in Q} \mathrm{AP}_q}{|Q|}$$

Altogether, MAP provides a single metric across the complete result set that incorporates both recall and precision. Thus, optimizing the system for either a high recall or precision value will not suffice to achieve a high MAP value but a high MAP indicates a relatively high precision across all recall levels.

To evaluate our API retrieval system, we use the MAP value calculated for the top 100 results for each query. Note, that including only the 100 best results would not be considered good practice for evaluating traditional IR systems but is sufficient in our case. API retrieval operates typically on smaller document collections (hundreds of thousands instead of millions) and the number of relevant results for an information need is also smaller. Our test queries have been associated with between 1 and 6 documents that we identified as relevant.

5.1.2. Recall at 10

While MAP is suitable for optimizing over the whole result set, the metric does not particularly well represent a user's perception of a good search engine. For instance, if a system answers a query with two relevant results by ranking one result at rank 1 and the other at rank 100, the resulting MAP will be higher than for a system that yields the relevant results at ranks 4 and 5. Assuming that the user will be interested in both results, the later system is probably considered to be more useful independent of the higher MAP of the former.

Furthermore, MAP is not an intuitive concept. Stating that a system has a MAP value of 0.7 does not provide much information except that it probably performs better than a system with a MAP of 0.65.

This is why we collect with *recall at 10* (R_{10}) an additional metric that is more user-centric and easier to communicate. R_{10} is simply the fraction of relevant documents in the top 10 results:

$$R_{10} = \frac{|\text{relevant retrieved}|}{\min\{|relevant|, 10\}}$$

Hence, $R_{10} = 0.75$ indicates that three fourth of the relevant results show up in the top 10 retrieved results, or in our case, on the first result page. Analogous to MAP, we use the arithmetic mean of R_{10} over all queries to get a single score for the performance of the retrieval system.

5.2. Test Collections

For the evaluation of our retrieval model, we extract API entities from the following open source Scala libraries:

- Scala Standard Library
- Scalaz
- Scala Refactoring

The Scala standard library is an obvious choice as it is well-known by all Scala developers and provides a big corpus of various functionality. Additionally, Q&A web platforms provide many questions concerning the standard library that have been used to derive realistic information needs.

Scalaz is a popular Scala library designed for a purer form of functional programming with strong abstractions in the style of Haskell's standard library. Scalaz makes heavy use of advanced Scala language features like higher-kinded types and the type class pattern. Thus, some of the functionality is not yet discoverable by the current state of our implementation (see subsection 3.3.3). Nevertheless, we decided to include Scalaz into the test collection. The reason was to ensure that the additional entities do not interfere with the information needs derived from the other libraries. Hence, we mainly use Scalaz to add "noise" to the retrieval process and to lift the numbers of indexed entities to over 100'000. Furthermore, Scalaz will definitively become an important part of the evaluation of future versions of our implementation with better support for implicit parameters.

The last library in our test collection is Scala Refactoring [Sto10]. This project provides infrastructure for and implementations of automated refactorings for Scala and is used in Scala IDE. We include Scala Refactoring as an example of a domain specific project. Hence, most functionality targets the analysis and transformations of abstract syntax tress. Furthermore, the library defines some extensions to the classes of the standard library. Such extensions are of particular interest for API retrieval as they implement general functionality not easily discoverable without a deep knowledge about the providing library.

5.3. Information Needs

We identified a total of 52 information needs for our test collections and formulated queries accordingly that have been associated with identifiers relevant to the query. Thus, a result is only relevant if it answers the information need and not if it just has the identical type signature as the query. The relevance judgment is binary such that an entity is either relevant or not. We do not use a relevance scale that allows judgments like "A is highly relevant and B is somewhat relevant to Q".

We ensured that the information needs cover various types of queries such that they represent different usage patterns. We use the following dimensions to informally classify usage patterns:

Navigational vs. Informational Queries

According to [Bro02], the purpose of a navigational query is to find a particular document that the user has in mind. The user has a relative clear notion of the document he is looking for, because he used it before or he assumes that such a document must exists. For example, sorting an array is a functionality provided by almost any programming language's standard library. Thus, a user will assume that there should be an according function and uses a query like `sort: Array => Unit`. The information he is looking for is mainly the identifier of the `sort` function.

On the other hand, the purpose of an informational query is to discover new information. A user may have a value of type `Person` and supposes that a person should have some relation to zero or more employees. Though, he does not know how employees are represented in the system or how this relation is provided. He uses a query of the form `employees: Person => List`. In this case, the information of concern includes more than just an identifier. The user is also interested in the type representing an employee and the details of the relation (e.g., only current employees or a history of employees).

Generic vs. Concrete Query Types

A user may not always be able to formulate his information need with the same genericity as provided by the indexed library. Usually, functionality from the standard library tends to be defined in a relatively generic way. But other functionality defined in more domain specific libraries may be defined less generic than possible. For example, a `sum` function on containers may have a concrete signature like `List[Int] => Int` or a more generic one like `Traversable[A] => Monoid[A] => A` which makes use of the `Monoid` type class. The same applies to queries formulated by search engine users. Because we assume that library designer tend to spend more effort to figure out the most generic implementation than search engine users, we want to also support queries with more concrete types as the according relevant entities.

Textual vs. Type-directed Queries

The same information need may be formulated by only using textual keywords,

	No. Configurations	Views	Fractions	Length Norm	Distance Boost	Depth Boost	Type Frequency	Name Boost	Doc Boost	Frequency Cutoff
Baseline	50	-	-	[0,0.5]	-	-	-	0.1	[0,0.5]	0.8
Baseline+All	500	-	on	[0,0.5]	-	[0,2]	[0,2]	0.1	0.05	0.8
FEM	50	on	-	[0,0.5]	-	-	-	0.1	0.05	0.8
FEM+Di	50	on	-	[0,0.5]	[0,2]	-	-	0.1	0.05	0.8
FEM+De	50	on	-	[0,0.5]	-	[0,2]	-	0.1	0.05	0.8
FEM+TF	50	on	-	[0,0.5]	-	-	[0,2]	0.1	0.05	0.8
FEM+Fr	50	on	on	[0,0.5]	-	-	-	0.1	0.05	0.8
FEM-Di	500	on	on	[0,0.5]	-	[0,2]	[0,2]	0.1	0.05	0.8
FEM-De	500	on	on	[0,0.5]	[0,2]	-	[0,2]	0.1	0.05	0.8
FEM-TF	500	on	on	[0,0.5]	[0,2]	[0,2]	-	0.1	0.05	0.8
FEM-Fr	500	on	-	[0,0.5]	[0,2]	[0,2]	[0,2]	0.1	0.05	0.8
FEM+All	500	on	on	[0,0.5]	[0,2]	[0,2]	[0,2]	0.1	0.05	0.8

Table 5.1.: Evaluated instantiations of the prototype with the according configurations; numbers in brackets denote the range in which the according parameters are generated

types or a combination of both.

Unfortunately, we have currently no data on the actual usage patterns of real user interactions with the search engine. This is why we decided to mainly focus on queries more interesting from an API retrieval point of view. Thus, the test queries are biased towards type-directed queries such that we can evaluate the fingerprint evaluation model. The complete test collection is provided in Appendix D.

5.4. Test Setup

To evaluate our retrieval model we instantiate the Scaps core library with various configurations that enable specific features. For each of these instantiations we search for a good parametrization by randomly generating parameter configurations for the enabled features and calculating the according MAP and R_{10} values by applying the system to the test collection. The configuration that achieves the highest MAP value is then compared to the other instantiations.

Table 5.1 lists the exact setup of the individual instantiations. Each instantiation uses some fixed parameters and some parameters that are randomly chosen with a uniform

distribution from a given range. “No. Configurations” indicates how many runs with randomly generated configurations have been evaluated for the according instantiation.

The first instantiation “Baseline” is used to compare our solution to simpler tools for searching functionality in APIs. This instantiation disables type views such that subtype relations are no longer included during the retrieval process. Such a system roughly corresponds to Eclipse JDT’s “Java Search” feature that allows type patterns in queries or to a sophisticated regular expression applied on ScalaDocs.

A more sophisticated textual solution may additionally use some heuristics to weight terms based on its frequency in the document collection or its depth in the query. This is reflected in the “Baseline+All” instantiation that additionally enables “Depth Boost”, “Type Frequencies” and “Fractions”. Enabling “Distance Boost” is not reasonable in this setup because it does not incorporate type views and the distance will therefore always be zero.

The next instantiation “FEM” is the raw fingerprint evaluation model without any of the extensions discussed in section 3.2 enabled but including type views like subtyping and implicit conversions.

The instantiations “FEM+Di” to “FEM+Fr” each enable an extension separately to analyze the improvement of the distinct features. Accordingly, “FEM-Di” to “FEM-Fr” each disable the extension in question. And finally, “FEM+All” instantiates FEM with all extensions enabled.

The parameter ranges have been chosen accordingly to our experience and to achieve feasible results without generating too many configurations with unreasonable parametrization. For example, we observed a length normalization weight of less than 0.3 generally results in a better score. With increasing values for this weight the performance of the system usually dropped rapidly. Hence, limit “Length norm” to values below 0.5 seemed reasonable. For the other parameters, we observed that the absolute value is of less influence than the rate between them. Thus, the lower limit of zero and a uniform distribution of the random number generator is more crucial than the upper limit of 2.

“Name Boost” is fixed for all instantiations to 0.1. Thus, we search good a parametrization relative to a given “Name Boost”. Additionally, “Doc Boost” is fixed to 0.05 for all instantiations expect for the “Baseline”. This is based on the observation that a ratio between “Name Boost” and “Doc Boost” of 0.5 gives sufficiently good results and that this ratio is usually of little impact to the overall score. Furthermore, the test collection mainly focuses on type queries because the support for keywords based searches is not yet as well developed.

Finally, the “Frequency Cutoff” parameter (see subsection 4.6.4) is also fixed to 0.8 for all instantiations. This parameter is only used to limit the number of matched documents and therefore improve the runtime of a query. The value has been chosen sufficiently high to not affect the quality of the result sets and needs to be further adjusted when instantiating the search engine for a productive environment.

Altogether, this test setup allows us to answer the questions initially stated in this chapter. First, we can verify the effectiveness of our approach compared to some existing approaches by comparing “Baseline” and “Baseline+All” with the remaining instantia-

	Views	Fractions	Length Norm	Distance Boost	Depth Boost	Type Frequency	Name Boost	Doc Boost	Frequency Cutoff	**MAP**	R_{10}
Baseline	-	-	0.10	-	-	-	0.1	0.02	0.8	**0.57**	0.69
Baseline+All	-	on	0.01	-	1.58	1.47	0.1	0.05	0.8	**0.62**	0.74
FEM	on	-	0.20	-	-	-	0.1	0.05	0.8	**0.36**	0.48
FEM+Di	on	-	0.14	1.27	-	-	0.1	0.05	0.8	**0.59**	0.72
FEM+De	on	-	0.20	-	0.26	-	0.1	0.05	0.8	**0.31**	0.41
FEM+TF	on	-	0.16	-	-	0.26	0.1	0.05	0.8	**0.51**	0.65
FEM+Fr	on	on	0.20	-	-	-	0.1	0.05	0.8	**0.27**	0.38
FEM-Di	on	on	0.06	-	0.14	0.80	0.1	0.05	0.8	**0.61**	0.81
FEM-De	on	on	0.13	0.31	-	0.83	0.1	0.05	0.8	**0.70**	0.87
FEM-TF	on	on	0.25	1.08	0.10	-	0.1	0.05	0.8	**0.55**	0.66
FEM-Fr	on	-	0.15	1.35	1.38	1.95	0.1	0.05	0.8	**0.68**	0.84
FEM+All	on	on	0.18	0.39	0.09	1.28	0.1	0.05	0.8	**0.70**	0.85

Table 5.2.: Evaluation results with the configuration of the best instantiations (by MAP); all parameters and results have been rounded to two decimal places

tions. Second, we get some insight of the effectiveness of the various extensions. And finally, as a side effect of the evaluation process, we get a good parametrization that can be used in a productive environment.

5.5. Results

Table 5.2 shows the detailed evaluation results with the configurations that achieved the highest MAP value per instantiation. The "Baseline" and "Baseline+All" instantiations achieved a MAP of 0.57 and 0.62 respectively. "FEM+All" resulted in the highest score with 0.7049 closely followed by "FEM-De" with a MAP of 0.6986. A more clearly laid-out comparison between the baseline instantiations and the FEM instantiations with distinct features disabled is given in Figure 5.2.

Additionally, Table 5.3 lists the per query scores of 20 randomly chosen queries for the "FEM+All" and "Baseline+All" instantiations. This comparison shows that both instantiations can answer more than half of the queries with the best possible average precision (AP) of 1. Thus, the n relevant entities for a query are yielded at the ranks 1 to n. Furthermore, some queries have been answered with a higher precision by the baseline instantiation.

	FEM+All		Baseline+All	
	AP	R_{10}	AP	R_{10}
`sort: Array[Float] => _`	0.87	1	0.67	1
`shuffle: List[A] => List[A]`	1	1	1	1
`unit unapply: Trees#Tree => Option`	1	1	1	1
`distance: (Trees#Tree, Position) => _`	1	1	1	1
`step: (Range, Int) => Range`	1	1	1	1
`Ordering[Char]`	0.82	1	0.5	0.5
`List[A] => (List[A], List[A])`	0.63	1	0.81	1
`(List[A], String) => String`	1	1	1	1
`String => Double`	1	1	0	0
`unapply: Trees#Block => Option`	1	1	1	1
`sort: (Array[A], Ordering[A]) => _`	1	1	1	1
`List[A] => (A => Boolean) => List[A]`	0.94	1	0.97	1
`Ordering[A] => (B => A) => Ordering[B]`	1	1	1	1
`Boolean => A => Option[A]`	0.42	1	0.25	0.5
`sort: Array[A] => _`	0.59	1	1	1
`(List[A], String, String, String) => String`	1	1	1	1
`parse: String => Float`	0.17	0.5	0.28	0.5
`ExecutionContext`	0.17	1	0	0
`List[Future[Float]] => Future[List[Float]]`	0	0	0.04	0
`(List[A], Int) => A`	1	1	1	1

Table 5.3.: Per query comparison between "FEM+All" and "Baseline+All" over 20 randomly chosen queries; If a score differs between the two instantiations the higher score is highlighted in green

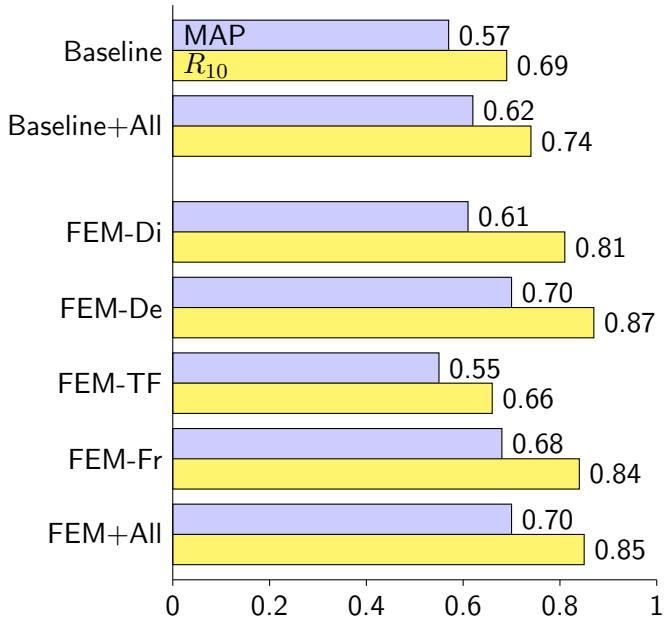

Figure 5.2.: Comparison of MAP and R_{10} between some selected instantiations

5.6. Discussion

The evaluation shows that the fingerprint evaluation model can answer information needs with a generally higher precision and recall than a solution using only textual matching. Additionally, the detailed analysis of the per query scores in Table 5.3 indicates that there are certain queries that can only be answered by incorporating subtyping and implicit conversions and the baseline approach is not sufficient to retrieve all relevant entities.

Our approach achieves generally better scores when the query type differs from the types used in the relevant definitions. For example, searching for implementations of the `Ordering` type class for `Char` with the query `Ordering[Char]` also retrieves the `CharOrdering` object which is of a subtype of `Ordering[Char]`. Also functionality provided by implicit conversions can be retrieved more effectively. But the incorporation of type views comes with a certain impact on the precision of simpler queries that mostly match the type signature of the relevant definitions.

Furthermore, we can conclude that some of the proposed extensions to the fingerprint evaluation model are crucial to the performance of the search engine. Incorporating "Distance Boost" and "Type Frequencies" into the weight of a fingerprint term results in significant improvements.

Disabling "Distance Boosts" decreases the maximum achievable MAP from 0.70 ("FEM+All") to 0.61 ("FEM-Di"). This result conforms more or less with our expectations as weighting alternative types with similar scores as the original type in the query is unlikely to produce feasible results. For example, `Int` in the query `Int =>`

`String` expands to the alternative types `Int`, `AnyVal` and `Any`. With an equal weighting for these alternative types, entities of type `Any => String` are scored equally as entities of type `Int => String`. This effect can also be observed on instantiations that do not incorporate some discrimination between alternative types like "FEM", "FEM+De" and "FEM+Fr".

More surprisingly is the observation that "Type Frequencies" seems to be the most important weighting factor as disabling this feature decreases MAP to 0.55 ("FEM-TF"), below the score of the baseline instantiations. One possible explanation is that type frequencies represents beside the specificity of a type also to some extent the distance to the original query type. Because of the way how query types are expanded and type frequencies calculated, an alternative type will always have a higher type frequency than the according original type.

While the "FEM+All" instantiation with both "Depth Boost" and "Fractions" enabled achieves slightly better scores than "FEM-De" and "FEM-Fr", we can not definitely conclude that these features contribute to the effectiveness of the system. Especially disabling "Depth Boost" has only a marginal impact on MAP and even slightly increased R_{10}. This implies that the assumption that the relevance of an atomic type in a query decreases with its nesting level does not necessarily hold.

5.7. Limitations

There are some flaws in our methodology to evaluate the proposed retrieval model that may be addressed in future work. First, the system used to measure the baseline is still relatively close to our retrieval model. A better approach would be to compare our system to an existing search engine that is also frequently used by developers like Hoogle. But similar tools for Scala 2.11 that would have allowed for a fair comparison are not available.

Second, a crucial flaw in our evaluation is that we use the same test collection to find a good parametrization of the various instantiations and to assess the score of the optimized instantiation. Best practice would suggest to use different test collections for these tasks to ensure that the system is not over-fitted to a particular set of test queries. Due to the relatively big effort required to come up with a good test collection with a feasible number of queries (> 50), we decided to not use different test collections for parameter optimization and evaluation for now. Especially the current lack of actual queries of real users makes it difficult to create a representative collection of queries and relevant entities.

Third, the test collection is biased towards our retrieval model as we have mainly focused on type queries that can be answered by our system. For example, functionality provided through the type class pattern or through higher-kinded types is not yet covered. As long there is no standardized test collection for API retrieval like TREC used in the text retrieval community, a biased test setup is probably inevitable.

Altogether, this evaluation still provides some evidence for the potential of our retrieval model. Especially the insight that we are able to incorporate subtype relations into

queries without drastically decreasing the precision of queries that can be answered without type views supports the usefulness of this approach.

5.8. Parametrization

Based on the insights of the evaluation, we derived a parametrization for a productive deployment of the prototype implementation. Beside good evaluation scores, this configuration should also provide a reasonable response time to user queries. The configuration given in Table 5.4 achieves almost identical test scores but decreases the average runtime of the test queries from 700 ms to 440 ms.

	Views	Fractions	Length Norm	Distance Boost	Depth Boost	Type Frequency	Name Boost	Doc Boost	Frequency Cutoff	**MAP**	R_{10}
Productive	on	on	0.1	0.25	0.1	1	0.1	0.05	0.5	**0.69**	0.85

Table 5.4.: Final parametrization of the productive environment with test scores

6. Conclusion

This final chapter recapitulates the contributions made during this thesis and discusses how our work can be continued in future projects.

6.1. Accomplishments

We have presented a new approach to type-directed API retrieval that is suitable to index libraries written in statically-typed, object-oriented languages. The retrieval model is able to incorporate various concepts common in such programming languages:

- Subtype Polymorphism
- Bounded Parametric Polymorphism
- Definition-site Variance Annotations
- Implicit Conversions (Coercive Subtyping)

Our approach derives an expression from type queries that incorporates both the type's structure and the type hierarchy. This expression is then used to match type signatures of definitions in the API and score them to collect a ranked result set.

Based on the implementation for Scala, we also demonstrated that our approach can effectively answer information needs. We integrated our retrieval model into the Lucene search engine and combined it with the text retrieval model to support mixed queries of textual keywords and types.

The fact, that the model achieved feasible results when applied on the relatively complex Scala Standard Library, indicates that the approach is suitable to be applied on a broad range of libraries. Especially the various collection types provided by the standard library make heavy use of bounded type parameters and multiple inheritance with deep type hierarchies. Additionally, we proposed further extensions for providing better support for libraries that heavily rely on the type class pattern.

6.2. Adaption to Other Languages

As already mentioned, we designed the retrieval model not exclusively with the Scala programming language in mind. This section discusses how the approach can be adapted to other programming languages.

6.2.1. C#

From the mainstream programming languages the type system closest to Scala's is probably used in C#. Beside bounded parametric and subtype polymorphism with similar semantics, C# also uses definition-site variance annotations and allows user-defined implicit conversions. Thus, the concepts incorporated in our prototype implementation should be more or less portable to a implementation for C#. Some language specific features like delegates and events may require additional transformations to synthetic types but should not introduce any major difficulties.

6.2.2. Java

Because interoperability with Java has been a major design goal for Scala, extending our implementation to also support Java libraries should require relatively little effort. In fact, the Scala compiler can also process Java class files which would allow to reuse a major part of our infrastructure to extract entities from Java libraries.

One major difference that may affect the effectiveness of the API retrieval system is that Java exclusively uses use-site variance annotations. One possible approach to overcome this limitation may be to assume every generic type to be invariant over its type parameters. Because Java's collection library defines exclusively mutable data-structures, which are usually invariant, this simplification may be sufficient to still retrieve good results.

6.2.3. Go

Go's type system is an interesting target for our retrieval model as it exclusively uses structural subtyping with named interfaces. Parametric polymorphism is only supported on some built-in types. Thus, Go uses interfaces to describe contracts on types but programmers cannot explicitly state that a certain type implements an interface. Instead, the compiler checks if a type conforms to an interface at use-site. This requires some additional effort during feature extractions as subtype relations are not directly available.

6.2.4. C++

With templates, C++ uses a different approach to provide parametric polymorphism compared to other mainstream programming languages. Put highly simplified, template type parameters are implicitly bounded by structural types derived from how the type parameter is used in the template body. For example, a polymorphic function

```
template<typename T>
void print(T t) {
  std::cout << t.prettify() << std::endl;
}
```

defines a type parameter `T` that is bounded by the requirement that there is a member function `prettify()` defined on `T` which returns a value of a type `X`. Instances of `X` have

to be passed to the right hand side of the << operator on instances of `basic_ostream&` (the type of `std::cout`). Hence, we require that there is a definition of the << operator with a signature like

```
Y operator<<(std::basic_ostream& os, X x);
```

where `Y` is another arbitrary type. Because `Y` is used at the left hand side of the second application of the << operator, we also require that there must be a further definition

```
Z operator<<(Y y, std::basic_ostream&(*fn)(std::basic_ostream&));
```

as the type of `std::endl` is a function accepting a `basic_ostream&` and returning a `basic_ostream&`. This second definition of << returns an arbitrary type `Z` that is not further constrained. Altogether, even simple examples like the `print` method can result in relatively complex constraints on type parameters. Because `T` is not explicitly bounded, we have to assume that it is constrained by the most generic type. A similar function in a language with explicit bounds may just require that `T` defines a member `prettify()` which returns a string. This type is clearly simpler compared to the most generic possible type derived from the requirements of the implementation.

This approach to polymorphism is extremely flexible as it does not unnecessarily constrain type parameters. As long as the compiler is able to substitute `T` with a concrete type and successfully resolve the applications of the << operator, one can pass anything that has a member `prettify()`. This member may also return an `int` or anything that can be written to a `ostream&`.

Because API retrieval model requires that parametric code must have known bounds on type parameters with clear subtype relations to types that conform to these bounds, porting the model to C++ will most likely not be a trivial task. Nevertheless, it may be possible to find a good approximation to this problem. One feasible solution may be to extract type bounds that are less restrictive than actually required by the template implementation. For example, `print` can be indexed with a structural type bound on `T` that requires a member `prettify()` without restricting its return type. As a consequence, every class defining a `prettify()` member would be considered as a subtype of this structural type. Such simplified type bounds are probably sufficient to still achieve good retrieval results.

6.2.5. C++ with Concepts

The upcoming C++17 standard will presumably introduce a new language feature called "Concepts" [SSR13] [ISO15] that may also simplify the adaption of our API retrieval model to C++. A concept is a set of syntactical requirements on template parameters. For example, a simple concept representing the constraints on `T` for the `print` method given in subsection 6.2.4 could be

```
template<typename T> concept bool Pretty =
  requires(T t){
```

```
    t.prettify() -> std::string;
  };
```

The concept `Pretty` accepts every type that defines a member function `prettify()` returning a string. As we can see, concepts allow to describe requirements more flexible than interfaces in other languages. A requirement can be an almost arbitrary expression like `t.prettify()` but also `prettify(t)`, `t + t` or `t()` with an optional requirement on the expression's return type.

Such concepts can now be used to constrain template parameters:

```
template<Pretty T>
void print(T t) {
  std::cout << t.prettify() << std::endl;
}
```

Instantiating the template `print` with a `T` that does not satisfy the `Pretty` concept is now objected by the compiler. Hence, `prettify()` must strictly return a string which is a more specific requirement than given by the `print` implementation with an implicitly constrained type parameter.

Concerning API retrieval, the explicit declaration of type parameter requirements will increase the accuracy of type bounds. While it was not possible to deduce the expected return type of `prettify()` in the implementation of `print` without concepts, `Pretty` now unambiguously defines that it is `std::string`.

6.3. Limitations and Future Work

Although we believe that the provided implementation is already a good starting point, further improvements are needed to precisely answer a broad range of information needs.

The most severe limitation is currently, that the retrieval model focuses on finding definitions that conform to the query type. While this seems feasible from a type theoretical point of view, it sometimes requires users to formulate more generic queries than intuitively reasonable. Especially the rule that only subtypes are matched at covariant positions is overly restrictive. For example, a common habit is to use `List` as a substitute for all types that represent sequences. But querying with `Int => List` yields relatively low scores for definitions of types like `Int => Iterable`.

Concerning the Scala language, the not yet implemented support for implicit parameters excludes some important functionality from being easily retrievable. Thus, a user must know about a type class to retrieve according definitions. While we suggested a possible solution to this issue with the implicit parameter instantiation in subsection 3.3.3, we were not yet able to implement and evaluate this approach.

An interesting continuation of this project would also be the integration of the search engine into an IDE. This would include the design of a user interface for invoking queries directly from the editor and to develop an algorithm that is able to substitute the query expression and its parameters with a definition selected by the user.

And finally, we hope that the Scaps service, publicly accessible at scala-search.org, gains enough traction to collect sufficient usage data to learn more about the actual usage of the search engine. These insights could then be used to improve the test collections and set the direction for further improvements.

6.4. Acknowledgments

I would like to thank Prof. Peter Sommerlad and Mirko Stocker for their continuous feedback during the weekly meetings and their valuable advices. I'm especially thankful for giving me a free hand in realizing this project, but also for challenging my ideas. This helped to stay focused and to not getting lost in details.

A special thanks goes also to Lukas Hofmaier and Frederick Egli. The countless discussions have been great fun and revealed many flaws in my work.

Also many thanks to the Scala community for creating the highly inspiring ecosystem around the Scala language.

A. Bibliography

[AH85] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In S N Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 19–41. Springer Berlin Heidelberg, 1985.

[AHS11] John Altidor, Shan Shan Huang, and Yannis Smaragdakis. Taming the Wildcards: Combining Definition- and Use-site Variance. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 602–613, New York, NY, USA, 2011. ACM.

[Apa] Apache Software Foundation. Apache Lucene. `https://lucene.apache.org/core/`. visited on 2015-08-10.

[Bro02] Andrei Broder. A taxonomy of web search. *ACM SIGIR Forum*, 36(2):3, 2002.

[CJS09] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. SNIFF: A search engine for java using free-form queries. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5503, pages 385–400, 2009.

[Doe] Sébastien Doeraene. Scala.js. `http://www.scala-js.org/`. visited on 2015-08-12.

[Dup] Thibault Duplessis. Scalex. `https://github.com/ornicar/scalex`. visited on 2015-08-10.

[Ecl] Eclipse. Eclipse JDT: Java Search Tab. `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-dialog-java-search.htm`. visited on 2015-08-10.

[ET] EPFL and Typesafe. Scala Standard Library 2.11.5. `http://www.scala-lang.org/api/2.11.5/`. visited on 2015-03-23.

[FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[GKKP13] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*, page 27, 2013.

[GRB+14] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 653–663, 2014.

[Haoa] Li Haoyi. Autowire. `https://github.com/lihaoyi/autowire`. visited on 2015-08-12.

[Haob] Li Haoyi. ScalaTags. `https://github.com/lihaoyi/scalatags`. visited on 2015-08-12.

[HPM+12] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. SIP-14 - Futures and Promises. `http://docs.scala-lang.org/sips/completed/futures-promises.html`, 2012. visited on 2015-06-29.

[ISO15] ISO/IEC JTC1 SC22 WG21. Programming Languages — C++ Extensions for Concepts, Technical Specification N4549. page 57, 2015.

[LSX13] Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: Theory and implementation. *Information and Computation*, 223:18–42, 2013.

[Mit] Neil Mitchell. Hoogle. `https://www.haskell.org/hoogle/`. visited on 2015-06-10.

[Mit11] Neil Mitchell. Hoogle: Finding Functions from Types. `http://community.haskell.org/~{}ndm/downloads/slides-hoogle_finding_functions_from_types-16_may_2011.pdf`, 2011.

[MRS08] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*, volume 1. Cambridge University Press, 2008.

[MXBK05] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. *ACM SIGPLAN Notices*, 2005.

[Ode06] Martin Odersky. Poor Man's Type Classes, 2006.

[Ode14] Martin Odersky. The Scala Language Specification Version 2.9. Technical report, EPFL, 2014.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. *ACM SIGPLAN Notices*, 40(10):41, October 2005.

[Pet12] Tomas Petricek. The theory behind covariance and contravariance in C# 4. http://tomasp.net/blog/variance-explained.aspx/, 2012. visited on 2015-07-22.

[Pie02] Benjamin C Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Rit90] Mikael Rittri. Retrieving Library Identifiers via Equational Matching of Types. *10th International Conference on Automated Deduction*, 1990.

[RWJ+94] S.E. Robertson, S. Walker, S. Jones, M.M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proceedings of 3rd Text REtrieval Conference*, pages 109–126, 1994.

[Sca] Scala IDE for Eclipse. http://scala-ide.org/. visited on 2015-08-10.

[Sca13] Scala. SI-7629 Deprecation Warning for View Bounds under -Xfuture. https://issues.scala-lang.org/browse/SI-7629, 2013. visited on 2015-07-21.

[SSR13] Andrew Sutton, Bjarne Stroustrup, and Gabriel Dos Reis. *Concepts Lite, Technical Report N3701*. ISO/IEC JTC 1, Information Technol- ogy Subcommittee SC 22, Programming Language C++, 2013.

[Sto10] Mirko Stocker. *Scala Refactoring*. Master's Thesis, University of Applied Science Rapperswil, 2010.

[SWY75] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. In *Communications of the ACM*, volume 18, pages 613–620, 1975.

[Tat13] Ross Tate. Mixed-Site Variance. In *FOOL '13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented Languages*, Indianapolis, IN, USA, 2013.

[TS06] Andrew Turpin and Falk Scholer. User Performance versus Precision Measures for Simple Search Tasks. *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 11—-18, 2006.

[Typa] Typesafe. Akka. http://akka.io/. visited on 2015-08-12.

[Typb] Typesafe. Spray. http://spray.io/. visited on 2015-08-12.

[Van79] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition*. Butterworths, 1979.

[WCGH15] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. Building Bing Developer Assistant. Technical report, Microsoft Research; MSR-TR-2015-36, 2015.

[Zha15] ChengXiang Zhai. Text Retrieval and Search Engines. `https://class.coursera.org/textretrieval-001`, 2015. visited on 2015-03-01.

B. Installation Guide

This guide describes how the various executables created during this thesis can be compiled and executed to reproduce our evaluation results and to use the Scaps web service. We assume that SBT with a version of 0.13 or higher is installed on the user's computer. All commands should be executed from the project's root directory if not stated otherwise.

B.1. Run the Evaluation

To benchmark the core library with the default configuration against the test collection defined in `evaluation/src/main/resources/`, run the following command:

```
sbt 'evaluation/run-main scaps.evaluation.Benchmark'
```

The default configuration can be found in `core/src/main/resources/reference.conf`. Single configuration parameters can be overwritten by setting them accordingly in `evaluation/src/main/resources/application.conf`. For example, to adjust the length normalization factor, the following line can be added to the `application.conf` file:

```
scaps.query.length-norm-weight = 0.15
```

The evaluation, as discussed in chapter 5, can be executed by running

```
sbt 'evaluation/run-main scaps.evaluation.Evaluation'
```

This command writes all benchmark scores of the instantiated retrieval system to `evaluation/target/results/evaluation-{date}.csv`. Additionally, a summary of the evaluation results will be written to `evaluation-stats-{date}.txt` in the same directory.

B.2. Run the Web Service

The simplest way to start an instance of the Scaps web service is by using SBT:

```
sbt 'webservice/run'
```

This will bind the User API and the Control API to the interfaces and ports as configured in `webservice/src/main/resources/application.conf`. By default, the User API (including the Web UI) is bound to all interfaces on the port `8080` and the Control API is bound to the `localhost` interface on the port `8081`.

For executing the Service in a productive deployment, using SBT is not advised. Instead, the service should be packaged in a standalone, executable jar:

```
sbt 'webservice/assembly'
```

This task merges all library dependencies in a single jar file and writes it to the `webservice/target/scala-2.11` directory. The jar can now be deployed to the productive environment and executed by running

```
java -jar -Dconfig.file=application.conf
    {jar-dir}/api-search-webservice-assembly-0.1-SNAPSHOT.jar
```

Custom configurations, can be set in the `application.conf` file. This file should at least set the `prod-mode` flag of the web service configuration to true:

```
scaps.webservice.prod-mode = true
```

An example configuration for a productive deployment is given in `webservice/src/main/resources/application-prod.conf`.

B.3. Publish and Install the SBT Plug-in

The SBT plug-in has to be published to a maven repository before it can be used in an SBT project. To publish all artifacts including the plug-in to the local Ivy repository, run:

```
sbt publishLocal
```

The plug-in can now be used by creating a new SBT project on **the same machine that also hosts the web service** and including the following line in the `project/plugins.sbt` file:

```
addSbtPlugin("org.scala-search" % "scaps-sbt" % "0.1-SNAPSHOT")
```

Additionally, the libraries to index have to be included in the project's `build.sbt` file as a library dependency. For example, to index scalaz, add

```
libraryDependencies += "org.scalaz" %% "scalaz-core" % "7.1.1"
```

The Scala Standard Library is a library dependency per default and does not need to be added in order to be indexed.

If the Scaps service is not exposed at the default ports, the correct hostname can bet set by using

```
scapsHost := "localhost:80"
```

to set the location of the User API and

```
scapsControlHost := "localhost:9000"
```

to set the location of the Control API.
Finally, an index job can be started by running

```
sbt scapsIndex
```

An example project, that demonstrates the required project structure is also given in the `demoEnvironment` directory.

C. Motivating Questions from Q&A Platforms

The following list shows some example questions from the Q&A platform *stackoverflow.com*. Such questions have been our initial source of motivation for developing the Scaps API search engine. Furthermore, these questions have been used to create the test collection to evaluate the system. Note, that the answers listed below can differ from the accepted answer if a newer release of the Scala Standard Library provided a better matching function.

1. Title: **Scala: join an iterable of strings**

   Relevant Answer: `scala.collection.Iterable.mkString`

   URL: http://stackoverflow.com/questions/13529512/scala-join-an-iterable-of-strings

2. Title: **Un-optioning an optioned Option**

   Relevant Answer: `scala.Option.flatten`

   URL: http://stackoverflow.com/questions/5968345/un-optioning-an-optioned-option

3. Title: **How in Scala to find unique items in List**

   Relevant Answer: `scala.collection.immutable.List.distinct`

   URL: http://stackoverflow.com/questions/1538598/how-in-scala-to-find-unique-items-in-list

4. Title: **How to transform Scala collection of Option[X] to collection of X**

   Relevant Answer: `scala.collection.immutable.List.flatten`

   URL: http://stackoverflow.com/questions/4730842/how-to-transform-scala-collection-of-optionx-to-collection-of-x

5. Title: **Scala enumeration to int**

   Relevant Answer: `scala.Enumeration.Value.id`

   URL: http://stackoverflow.com/questions/6632855/scala-enumeration-to-int

6. Title: **How to create a list with the same element n-times?**

   Relevant Answer: `scala.collection.immutable.List.fill`

URL: http://stackoverflow.com/questions/12300165/how-to-create-a-list-with-the-same-element-n-times

7. Title: **Hex String to Int,Short and Long in Scala**

   Relevant Answer: `java.lang.Integer.parseInt`

   URL: http://stackoverflow.com/questions/10763730/hex-string-to-int-short-and-long-in-scala

8. Title: **Scala char to int conversion**

   Relevant Answer: `scala.runtime.RichChar.toDigit`

   URL: http://stackoverflow.com/questions/16241923/scala-char-to-int-conversion

9. Title: **How to return an option when reading a vector**

   Relevant Answer: `scala.collection.immutable.Vector.lift`

   URL: http://stackoverflow.com/questions/26617611/how-to-return-an-option-when-reading-a-vector

10. Title: **How to convert List[List[Map[String,String]]] to List[Map[String,String]]**

    Relevant Answer: `scala.collection.immutable.List.flatten`

    URL: http://stackoverflow.com/questions/11690479/how-to-convert-listlistmapstring-string-to-listmapstring-string

11. Title: **What is an idiomatic Scala way to "remove" one element from an immutable List?**

    Relevant Answer: `scala.collection.immutable.List.diff`

    URL: http://stackoverflow.com/questions/5636717/what-is-an-idiomatic-scala-way-to-remove-one-element-from-an-immutable-list

12. Title: **Scala Convert Set to Map**

    Relevant Answer: `scala.collection.Set.zipWithIndex`

    URL: http://stackoverflow.com/questions/4851418/scala-convert-set-to-map

13. Title: **Scala convert Iterable or collection.Seq to collection.immutable.Seq**

    Relevant Answer: `scala.collection.Seq.to`

    URL: http://stackoverflow.com/questions/16939611/scala-convert-iterable-or-collection-seq-to-collection-immutable-seq

14. Title: **Convert String to Byte**

    Relevant Answer: `scala.Char.toByte`

    URL: http://stackoverflow.com/questions/3335821/convert-string-to-byte

15. Title: **Scala create List[Int]**

    Relevant Answer: `scala.collection.immutable.List.range`

    URL: http://stackoverflow.com/questions/2514438/scala-create-listint

D. Test Collection

This section lists the complete test collection used to evaluate our retrieval system. The collection is a map from queries to one or more relevant API entities. Additionally, we provide the exact Maven identifiers of the indexed libraries consisting of a group ID, an artifact ID and a version number.

This listing omits the types of the relevant identifiers which are necessary in some cases to unambiguously identify overloaded definitions. The complete test collection can be found in the according `evaluation/src/main/resources/{library}-test-collection.conf` files.

D.1. Scala Standard Library

Maven ID: `org.scala-lang:scala-library:2.11.5`

```
"String => Int" = [
  scala.collection.immutable.StringOps#toInt
  scala.collection.immutable.WrappedString#toInt
]
"String => Double" = [
  scala.collection.immutable.StringOps#toDouble
  scala.collection.immutable.WrappedString#toDouble
]
"max: Int" = [
  scala.Int.MaxValue
]
"max: (Int, Int) => Int" = [
  scala.math.max
  scala.runtime.RichInt#max
  scala.math.Ordering.Int.max
]
"Ordering[Char]" = [
  scala.math.Numeric.CharIsIntegral
  scala.math.Ordering.Char
  scala.math.Numeric.CharIsIntegral.reverse
  scala.math.Ordering.Char.reverse
]
"Ordering[A] => (B => A) => Ordering[B]" = [
  scala.math.Ordering#on
  scala.math.Ordering.by
]
"Ordering[A] => Ordering[Option[A]]" = [
```

```
  scala.math.Ordering.Option
]
"List => java.util.List" = [
  scala.collection.JavaConversions.seqAsJavaList
  scala.collection.convert.WrapAsJava.seqAsJavaList
]
"java.util.List => collection.Iterable" = [
  scala.collection.JavaConversions.asScalaBuffer
  scala.collection.convert.WrapAsScala.asScalaBuffer
]
"(Int, Int) => Range" = [
  scala.collection.immutable.Range.apply
  scala.runtime.RichInt#until
  scala.collection.immutable.Range.inclusive
  scala.runtime.RichInt#to
]
"exclusive: (Int, Int) => Range" = [
  scala.collection.immutable.Range.apply
]
"step: (Range, Int) => Range" = [
  scala.collection.immutable.Range#by
]
"(List[Char], String) => String" = [
  scala.collection.immutable.List#mkString
]
"(List[A], String) => String" = [
  scala.collection.immutable.List#mkString
]
"(List[Char], String, String, String) => String" = [
  scala.collection.immutable.List#mkString
]
"(List[A], String, String, String) => String" = [
  scala.collection.immutable.List#mkString
]
"(List[A], Int) => A" = [
  scala.collection.immutable.List#apply
]
"List[A] => (A => Boolean) => List[A]" = [
  scala.collection.immutable.List#filterNot
  scala.collection.immutable.List#filter
  scala.collection.immutable.List#withFilter
  scala.collection.immutable.List#dropWhile
  scala.collection.immutable.List#takeWhile
]
"List[A] => Option[A]" = [
  scala.collection.immutable.List#lastOption
  scala.collection.immutable.List#headOption
]
"Option[Option[Char]] => Option[Char]" = [
```

```
  scala.Option#flatten
]
"Option[Option[A]] => Option[A]" = [
  scala.Option#flatten
]
"List[Option[Char]] => List[Char]" = [
  scala.collection.immutable.List#flatten
]
"List[Option[A]] => List[A]" = [
  scala.collection.immutable.List#flatten
]
"List[List[A]] => List[A]" = [
  scala.collection.immutable.List#flatten
]
"List[Future[Float]] => Future[List[Float]]" = [
  scala.concurrent.Future.sequence
]
"List[Future[A]] => Future[List[A]]" = [
  scala.concurrent.Future.sequence
]
"await: Future[T] => _" = [
  scala.concurrent.Await.result
  scala.concurrent.Await.ready
]
"ExecutionContext" = [
  scala.concurrent.ExecutionContext.Implicits.global
]
"sort: Array[Float] => _" = [
  scala.util.Sorting.quickSort
  scala.util.Sorting.quickSort
  scala.util.Sorting.stableSort
]
"sort: (Array[A], Ordering[A]) => _" = [
  scala.util.Sorting.quickSort
  scala.util.Sorting.stableSort
]
"sort: Array[A] => _" = [
  scala.util.Sorting.quickSort
  scala.util.Sorting.stableSort
  scala.util.Sorting.stableSort
]
"List[Char] => (List[Char], List[Char])" = [
  scala.collection.immutable.List#splitAt
  scala.collection.immutable.List#span
  scala.collection.immutable.List#partition
]
"List[A] => (List[A], List[A])" = [
  scala.collection.immutable.List#splitAt
  scala.collection.immutable.List#span
```

```
  scala.collection.immutable.List#partition
]
"List[A] => Int => TraversableOnce[TraversableOnce[A]]" = [
  scala.collection.immutable.List#combinations
  scala.collection.immutable.List#sliding
  scala.collection.immutable.List#grouped
]
"shuffle: List[Float] => List[Float]" = [
  scala.util.Random.shuffle
  scala.util.Random#shuffle
]
"shuffle: List[A] => List[A]" = [
  scala.util.Random.shuffle
  scala.util.Random#shuffle
]
"(collection.Seq[A], collection.Seq[B]) => collection.Seq[(A, B)]" = [
  scala.collection.Seq#zip
  scala.collection.Seq#zipAll
  scala.runtime.Tuple2Zipped.Ops#zipped
  scala.runtime.Tuple2Zipped.Ops#invert
]
"random: Double" = [
  scala.util.Random.nextDouble
  scala.util.Random.nextGaussian
  "scala.math.random: scala.Double"
]
"(List[A], Int) => Option[A]" = [
  scala.collection.immutable.List#lift
]
"List[A] => B => (B => A => B) => B" = [
  scala.collection.immutable.List#foldLeft
  scala.collection.immutable.List#foldRight
  scala.collection.immutable.List#fold
]
```

D.2. Scalaz

Maven ID: `org.scalaz:scalaz-core_2.11:7.1.1`

```
"Boolean => A => Option[A]" = [
  scalaz.Scalaz.option
  scalaz.syntax.std.BooleanOps#option
],
"parse: String => Float" = [
  scalaz.std.string.parseFloat
  scalaz.syntax.std.StringOps#parseFloat
],
"parse: String => Double" = [
```

```
  scalaz.std.string.parseDouble
  scalaz.syntax.std.StringOps#parseDouble
]
```

D.3. Scala Refactoring

Maven ID: `org.scala-refactoring:org.scala-refactoring.library_2.11:0.6.2`

```
"expand: (Selection, Position) => Selection" = [
  scala.tools.refactoring.common.Selections#Selection#expandTo
]
"distance: (Trees#Tree, Position) => _" = [
  scala.tools.refactoring.common.PimpedTrees#TreeMethodsForPositions#distanceTo
]
"children: Trees#Tree => List[Trees#Tree]" = [
  scala.tools.refactoring.common.PimpedTrees#children
]
"unapply: Trees#Apply => Option" = [
  scala.tools.refactoring.common.PimpedTrees#ApplyExtractor.unapply
]
"unapply: Trees#Block => Option" = [
  scala.tools.refactoring.common.PimpedTrees#BlockExtractor.unapply
]
"raw source code: String => Trees#Tree" = [
  scala.tools.refactoring.common.PimpedTrees#PlainText.Raw.<init>
]
"unit unapply: Trees#Tree => Option" = [
  scala.tools.refactoring.common.TreeExtractors#UnitLit.unapply
]
"memoize: (A => B) => (A => B)" = [
  scala.tools.refactoring.util.Memoized.on
]
"replace: List => List => List => List" = [
  scala.tools.refactoring.transformation.TreeTransformations
    #AdditionalListMethods#replaceSequence
]
```

E. Project Management

E.1. Tools

Scala IDE 4.0.0
The development environment

Git For version management

Jenkins
Deployed on the project server and used for continuous builds

Redmine
Deployed on the project server and used for issue and time tracking

Multimarkdown
To create this document using LaTeX as a backend

TikZ and PGFPlots
To draw the majority of the plots and figures

Astah
To draw the UML diagrams

Sublime Text
For writing this report

E.2. Revision Control

The git repository of this project is located at https://git.hsr.ch/git/scalaAPISearch. The commit corresponding to the state of the project used to create the evaluation results as discussed in chapter 5 is tagged with the git tag `evaluation`. To reproduce the evaluation results, the repository can be checked out with the following command:

```
git clone --branch evaluation https://git.hsr.ch/git/scalaAPISearch
```

E.3. Benchmark Progress

As mentioned in section 4.11, we used the `Benchmark` executable to continuously evaluate the current state of the project. The progress of the according test results is illustrated in Figure E.1.

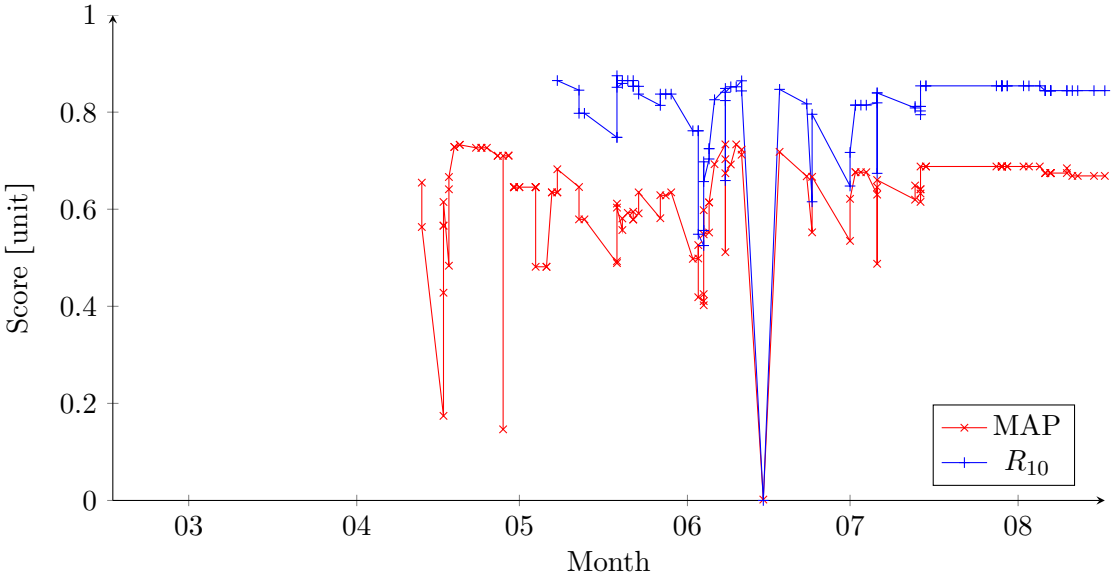

Figure E.1.: Progress of the benchmark results collected by the continuous integration server; each point on the x-axis corresponds to a commit to the git repository

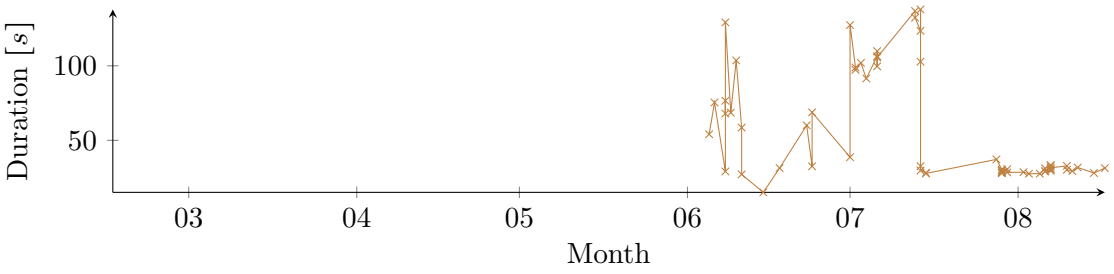

Figure E.2.: Progress of the benchmark runtime

From the project start in the calendar week 8 until week 15, the automatic evaluation of the MAP value was not yet implemented. In week 19, we started to additionally measure the R_{10} value. The project ended with a final commit in week 34.

Because the test collection has been continuously extended, the test scores do not increase with the progress of the project. Typically, additional test queries resulted in a decrease of the test scores and a successive adjustment of the retrieval model and the parametrization in the following commits lead to a recovery of the benchmark results. Other downward spikes are caused by changes that had accidental effects on the test score. These errors were detected easily and fixed with the subsequent commit.

In week 23, we also started to collect data about the duration of evaluating the test collection as shown in Figure E.2. This plot nicely demonstrates how the introduction of the *type frequency cutoff* parameter in mid of July helped to reduce the query time.

E.4. Time Report

The mandatory workload of this project is 27 ECTS which corresponds to a total of 810 hours. The project duration was initially framed to 21 weeks (not including the one week easter vacation) which gives an average workload of 38.6 hours per week. The sum of the actual work hours was 940.

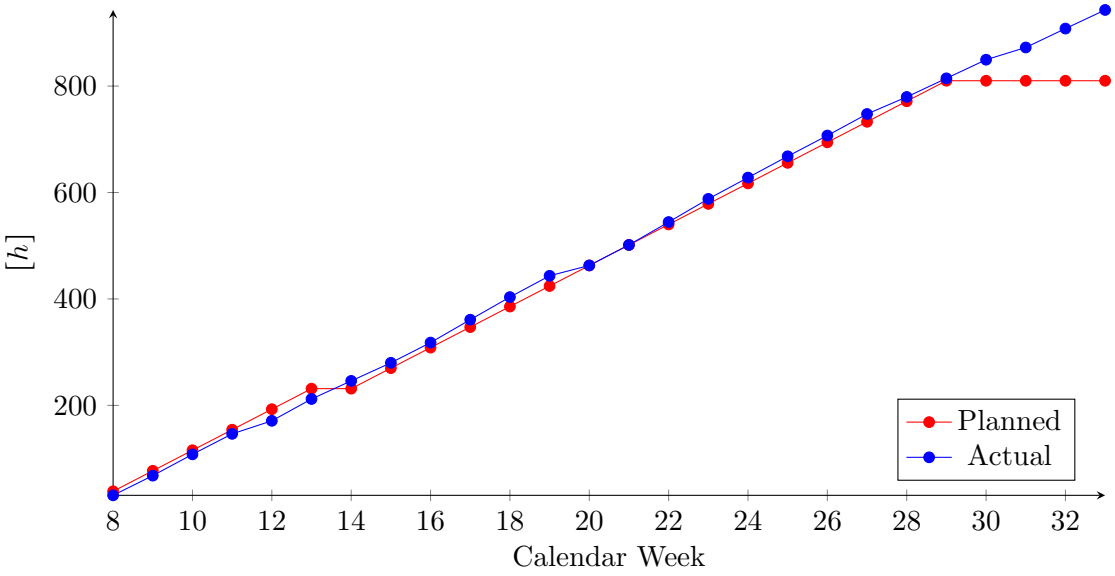

Figure E.3.: Accumulated hours of planned and actual workload per week

E.5. Project Schedule

Figure E.4 shows a coarse-grained overview on the project schedule and compares the planned and actually performed tasks in each week. Note, that we marked only the main tasks performed in a certain week. Some further comments on the schedule:

- The initially scheduled project duration was 21 weeks (W8 to W29). Because writing the documentation and evaluating our solution required more effort than initially anticipated, we extended the project to W33. In W34, we finalized the report.
- In W15, we decided to direct the project's focus towards a more refined retrieval model. This decision also included, that the initially anticipated integration into Scala IDE has been dropped. Instead, we added the new tasks "SBT Plug-in" and "Webservice" to the project schedule.
- Altogether, we initially underestimated the complexity of the API retrieval problem applied to Scala's type system.
- Also, we did not further pursue additional feature extractors for Java byte code and Java source files. Extending our approach to other languages before achieving satisfying results with Scala libraries seemed not reasonable.
- The majority of this report has been written in the last third of the project based on countless notes and drafts collected during the implementation.
- The retrieval model presented in this report has not been designed up-front before we started with the implementation. Instead, we had some initial ideas on how we can address the problem with the Vector Space Model and continuously refined our approach.

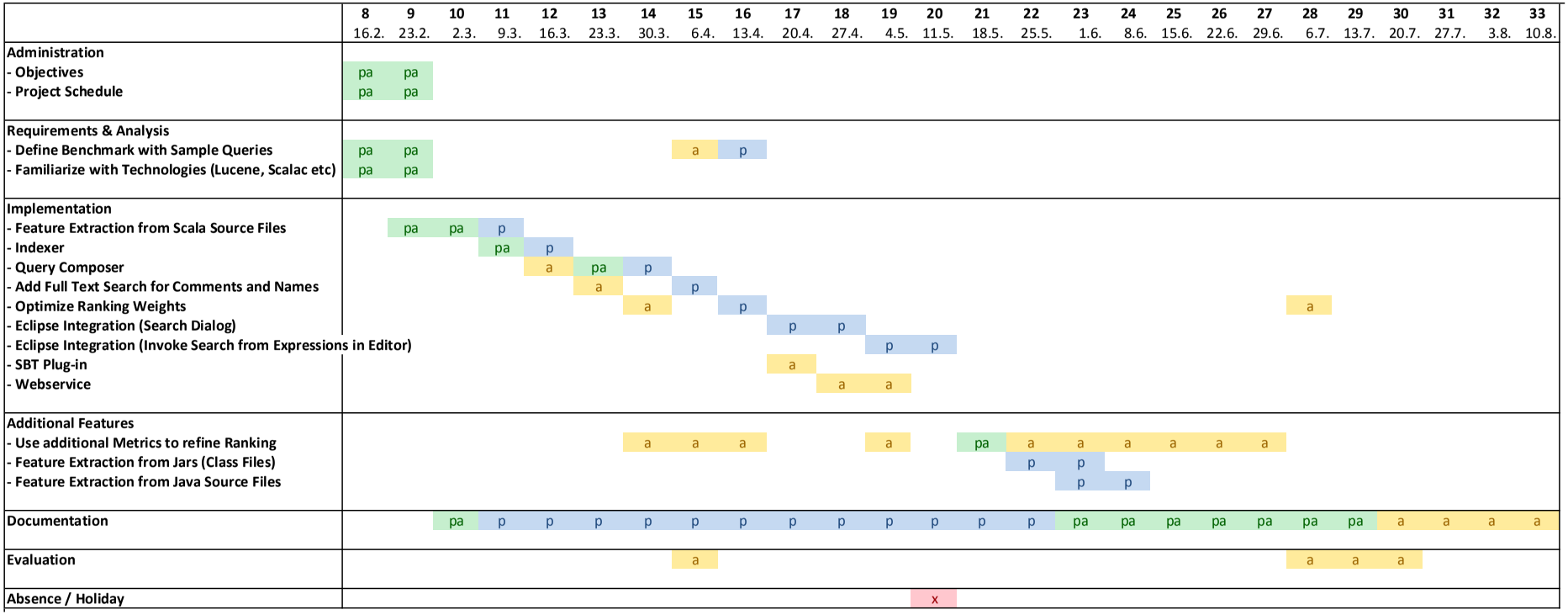

Figure E.4.: Project schedule; the planned tasks per week are marked blue (p); actual work on a task is marked yellow (a); matches of planned and actual work are marked green (pa)