
Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

# Domain–filtered API searching based on type signature in the Java ecosystem

**Scientific Students' Association Report**

Author:

Krisztián Márkus

Advisor:

dr. Péter Ekler

2020

# Contents

# Kivonat

Statikus típusos nyelvekben gyakran elég információ következtethető ki az egyes műveletekről azok típus szignatúrája alapján, hogy alapvető képességeikről spekuláljunk. Az így kinyert információ különösen hasznos lehet rosszul dokumentált könyvtárak vagy modulok esetén, de adott kódbázis ezen alapuló manuális keresése hosszadalmas folyamat. Esetenként megkísérelhető a szükséges műveletek reguláris kifejezések vagy egyszerű szöveges keresések által végzett meghatározása, melyeket azok fejlécein végzünk el, de ezek önmagukban nem kellően kifinomult eszközök a szignatúrák típus rendszer kontextusbeli jelentésének feldolgozására.

Egy lehetséges megoldás egy olyan kereső motor implementációja ami lehetővé teszi ezen információ feldolgozását majd az eredményt felhasználva a megfelelő kompatibilitást ellenőrizve műveletek keresését.

Mivel a cél elsődlegesen a gyakorlatbeli ipari felhasználás és ezen keresztül a szoftver fejlesztési folyamat megkönnyítése és felgyorsítása, így erősen hangsúlyos a tervezett rendszer gyorsasága és helyessége. Egyrészt a keresések időtartama nem szabad hogy jelentősen kizökkentse a felhasználó gondolatmenetét, másrészt — mivel a rendszer céljából adódan gyakran bizonyos ritkán használt operációk megtalálása a cél, — a keresési mechanizmusnak pontosan le kell fednie a felhasználhatóságot és így a találatoknak teljesnek és korrektnek kell lennie.

Egy ilyen eszköz létrehozása gondos tervezési folyamatot igényel hogy biztosítva legyen a mechanizmus pontos illeszkedése a cél domainre, és emellett a szükséges performancia is fenntartható legyen. Dolgozatomban bemutatom a munkám során kidolgozott, az előbbi feltételeket kielégítő modellezési és operációs terveket, valamint az ezekre alapuló megvalósított implementációt az objecktum orientált nyelvek környezetében. Fókuszban a Java nyelv áll, másodlagos célként más JVM nyelvek feldolgozása.

Munkám során nagy inspirációt nyújtottak hasonló létező eszközök, főként a Haskell modulokon operáló Hoogle [6] kereső.

# Abstract

In statically typed languages, often enough information can be inferred about operations through their respective type signatures, as to extract basic knowledge about their capabilities. This information is especially useful for poorly documented libraries and modules; however, searching a given codebase based on this manually is a tedious task. Sometimes attempts are made to find needed operations based on their signature using either regular expressions or plain text search, but such methods are naturally insufficient to properly understand the meaningful structures of the type system.

A viable solution is to implement a search engine that can contextualize the information, and search operations based on proper compatibility.

As the primary goal is real–world production usage and therefore the simplification and expedition of the software development process, the performance and soundness of the designed system is of the utmost importance. On one hand, the duration of searches must not hinder the user, and also — due to the use case of the system often making rarely used operations its target, — the search mechanism must accurately match applicability thus ensuring the correctness and soundness of the results.

The creation of a tool like this requires careful design work to provide an apparatus matching the target domain, and meet the needed performance demands. In this thesis I will present the model and operational designs that pass the aforementioned qualifications, and the implementation based on these for object oriented environments. The focus is initially the Java programming language, with support for other JVM languages being a secondary objective.

This work is largely inspired and influenced by some similar existing products, most notably the Hoogle [6] search engine which operates on the domain of Haskell modules.

# Chapter 1

# Introduction

It is quite a common occurrence in programming to suspect the operation we are in need of is already implemented — either in the standard library, or in an external module —, but we are unable to pinpoint the actual function. In these cases we might have to read through long documentations, try to semi–randomly check on possible candidates, or do some sort of a search. Although often useful, plain text– and regular expression–based searches still have a number of disadvantages. One problem is that even commonplace operations are rarely standardized — even within a single ecosystem —, thus we might have to try many terms to find the correct one. Another problem is that, for various reasons, the vocabulary of programming tends to be small, which results in the common reuse of simple names and many false positive results for our hypothetical search.

The proposed solution is a system that is capable of understanding the contextual meaning of the types of the operations provided by a set of dependencies, and execute searches on these that themselves contain deeper meaning than plain text tokens. The goal is to have the ability to describe our desired operation using types and get back applicable functions.

Such systems do exists, most notable of all is Hoogle [6], which operates on Haskell modules and have been a great source of inspiration for this project. However, there is no similar product operating on object oriented languages in common use that I know of.

## 1.1 Objective

Our goal is to design and implement a system that can find us operations that may be suitable for a given task, based on the types of its parameters. One should be able to write a simple text query that describes one's desired function and the system should look for functions — in the standard library and other dependencies — which can be invoked by these types of arguments. This doesn't mean that the result functions' in– and output types would need to be exactly the same as our query's parameters' types, but they should be compatible so that if one has some parameters that can be applied to a hypothetical function with signature same as the query, than any result function should be applicable as well. The search need to handle generic functions and types, and should also manage generic types as query parameters.

### Example use case: max with comparator

As an example let's imagine a scenario where we want to find a maximum of a list. We have a `List<Dog>` object, where the type hierarchies are as follows (not the same as in the standard library):

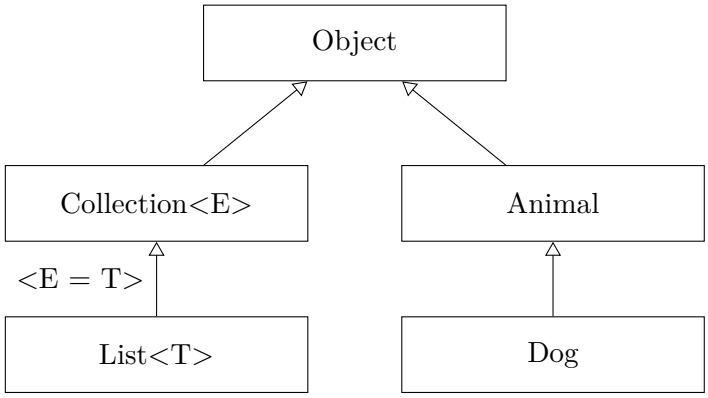


**Figure 1.1:** *Types hierarchy of example codebase*

We also have an animal comparator object `Comparator<Animal>` that we wish to use to find the maximum with.

Java's standard library (JCL) provides us with just the function that we need; in the `java.util.Collections` static utility class we find the following functions among others:

```
public final class Collections {

      static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp) { ... }

      static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp) { ... }

      // Rest of the class

}
```

`Comparators.java`

Now if we want to search for our desired operation, we should only have to write a simple query based on it's parameters:

```
  List<Dog>, Comparator<Animal> -> Dog
```

*Simple query for max*

This query is intuitively understandable to anyone, simply describes its desired operation's inputs and output, and should result in the proper `max` function being listed.

If you observe the also included `min` function, you will see that its signature is exactly the same as its counterpart's. This is no coincidence of course, they are practically mirrorings of each other. This means for us, that (unless we add text name filtering to parameters) similar functions — in fact like here, *closely related* functions — may be present in our result set.

## 1.2 Obstacles to overcome

Naturally, function applicability testing is a problem with existing solutions (if anything, take any compiler for example), and thus searching based on compability may seem a solved field, but there are many challenges to be found.

For starters, type–safe testing is usually deeply embedded into a given compiler, and thus is coupled tightly to it's model. To be able to process external, untrusted and optionally language independent packages in a safe way, we must be able to build a model that focuses on the information we need and not depend on existing language tools.

As mentioned earlier, performance considerations are highly important as well, as — opposed to, say, compilation — searching will invoke many, possibly hundreds of thousands of individual checks. The model thus needs to provide the information needed in an easy to access manner.

Another challenge is the presence of highly generic functions in the codebase. Operations that operate on the root type of our type system (`Object` in Java's case) will only be limited by their arity. While not including fitting functions would cause unwanted behavior, this problem is solvable by the ordering of the results based on various factors (see later in Chapter 7).

## 1.3 Overview

The following chapters will examine the research and design process that was necessary to architect the system capable of meeting the arisen challenges and fulfil the user requirements.

Firstly, Chapter 3 (*Language model*) will define the abstract target language model and type system which I have designed. While based on Java's own, this model will provide a much cleaner and simpler archetype to use and allow later on for adding secondary language support for different environments.

Chapter 4 (*Queries*) will then explore the concept and semantics of the queries the users may use to express their intents.

The core model, one of the most important aspects of the design groundwork will be detailed in Chapter 5 (*Data model*), with special reagards to the types and operations defined.

Chapter 6 (*Managing Java specific behavior*) describes the way the target Java language may be incorporated into the described abstract model using idiosyncratic mappings and other special solutions.

The searching mechinism is detailed in Chapter 7 (*Query fitting*).

Chapter 8 (*Validation of results*) describes the actual process of building the concrete program based on the research work's blueprint results.

Finally, Chapter 9 (*Result and conclusion*) concludes the research and proposes further directions for possible future work.

# Chapter 2

# Related works

The earliest appearance of type–based component searching that I have encountered are in *Retrieving re–usable software components by polymorphic type* [9] and *Using types as search keys in function libraries* [8]. Both of these papers deal with Hindley–Milner type systems and do not deal with object oriented type hierarchies.

As discussed before, *Hoogle* [6] was the biggest inspiration for my work, as it is a production grade project that I have been using for years now. It has influenced heavily the query syntax used by the project, even though its core logic is fairly different due to differences in the language models used.

Ecosystem–wise, *Scaps* [10] is the closest project I came to know, and for sure the closest procuction–grade tool. As it works on Scala modules, it is close to the project's target since it is also — at least partially — an object oriented programming language with primary focus on the JVM as compilation target. Due to its focus, it of course lacks Java–specific functions, and is built on a completely different architecture both in terms of its data mining functionality and its query evaluation method.

As the authors of Scaps note [10]:

> "*While there exist some effective approaches to address type–directed API search for functional languages, we observed that none of these have been successfully adapted for use with statically–typed, object-oriented languages*"

This sentiment strongly matches my own, and can be supported by the fact that while functional programming environments have multiple working examples of type–based search engines (some with even real–world usage), their object oriented counterparts lack similar tools. If we take into consideration that object orinted languages enjoy (much) higher rates of usage (See: [2][4]) over functional ones, the asymmetry becomes even more pronounced.

*A Novel Type–based API Search Engine for Open Source Elm Packages* [7] is a more recent approach of building a similar system for the Elm programming language. The article describes a *type unification algorithm* that is much similar to my implementation, although also much simpler due to the fact that Elm's type system (one heavily influenced by Haskell) lacks type hierarchies.

*Sourcerer* [1] is a search engine for open source code, that supports structured–based searches, but deals with algorithms rather than type signatures. This method could be combined in the future with type compability checks to increase search accuracy.

# Chapter 3

# Language model

As a natural precursor to the design work, a model must be specified to describe the type system that we wish to work on. Obviously such model will have to be based on the Java language's own type system, but should target a sufficient abstraction level as to allow us to adopt it later on for similar languages.

This can be achieved by taking the relatively influential class–based object oriented core while doing away with the idiosyncratic one–off features of the language — whether those are a result of performance considerations, historic baggage, or implementation limits. In our case this concerns language traits such as the distinct concept of primitives and arrays, permitted raw use of generic types, unorthodox function–types, etc. These aspects will have to be handled through decoupled, language–specific modules so as to keep the core as language agnostic as possible.

This chapter will describe the model I have created that contains all information deemed necessary by the research groundwork to allow for searching.

## 3.1 Object oriented paradigm

The concept of type–based searching is usually applied to functional programming languages, with working examples written for Haskell [6] and Clean [5]; although Scala [10] is also targeted. I believe the reason for this is the larger importance that functions enjoy in those kind of systems. They are usually the core building blocks, and higher–order manipulations of them are basic features of their languages.

An object oriented system — and especially one rather strict, like Java — do come with complications that obstruct the usefulness of type–based searching, but there may be mitigating circumstances that I believe make the effort of creating a search engine worth it.

The core principle of this endeavor is the assumption that individual operations (i.e. functions) provide meaningful information to us in isolation, through their type signatures. While this is a fairly trivial statement in a functional environment, that may very well not be the case in a strict object oriented framework — a rather strongly typed and static type system is a prerequisite in both cases. The problem in the OO case is that operations are usually tied to an object and its hidden inner state, and thus they can depend on implicit parameters that their signature does not really convey to us.

Take for example the so called builder pattern, a popular design scheme in the object oriented word, here examined through a simplified example that handles dates:

```
public class ExampleDate {

      public static class Builder {
          private boolean commonEra = true;
          private int year = 0;
          private Month month = Month.JANUARY;
          private int day = 1;

          void setIsCommonEra(boolean commonEra) {
              this.commonEra = commonEra;
          }

          // Rest of the setters and getters

          ExampleDate build() {
              if (!verify()) {
                  // throw exception or return null
              }

              return new ExampleDate(
                  commonEra, year, month, day
              );
          }

      }

      private final boolean commonEra;
      private final int year;
      private final Month month;
      private final int day;

      private ExampleDate(
          boolean commonEra,
          int year,
          Month month,
          int day
      ) {
          this.commonEra = commonEra;
          this.year = year;
          this.month = month;
          this.day = day;
      }

      // Rest of the class

}
```

*Date builder example*

While obviously this is an extremely naïve and simplistic example, it will do just fine for us to illustrate some of the more problematic aspects of working with OO programs. When observing the nested `Builder` class, and more importantly its functions, it's noticeable that they rely on the internal member variables of the class as opposed to external parameters. The first thing we may do to 'correct' this is to consider the implicit '`this`' parameter — which can represent all of the internal state — a *real* function parameter, possibly in a flexible manner to optionally handle it if needed. Another idea that will be experimented with later is to allow searching for multiple operations as if they were composed into a single function. This should allow searching for composite results of functions that are called sequentially. Although this is a powerful concept that can be helpful in many cases, it also heavily complicates searches and effectively explodes the search–space of our system.

## 3.2 Type hierarchy

Class–based systems are the most popular version of the object oriented paradigm, powering many popular languages, inducing the Java language — the main target of this project. Thus naturally our language model will reflect the semantics of a class–based type system.

It is most important to keep in mind the desired objective: *to provide the ability of searching for behavior*, and doing so using type signatures. This means, that we do not care about application state, including the inner member variables of objects. This principle will allow us to create a more uniform model than that of Java's, because it lets us do away with the distinct categories of *classes* and *interfaces* and create the general concept of *types*.

For our intents and purposes, a *type* will refer to anything that describes possible behavior we might look for. The two main referential building blocks of Java — interfaces and classes — will be joined together in this definition, along with some other, more special elements. Using this concept, every value in a Java program can be modeled with a type that may have additional *supertypes* (some special cases will be addressed later on).

With that, some OOP principles became fairly insignificant, as they deal with state. Yet a highly influential foundation of OOP is elevated in importance. Behavioral subtyping (also known as the Liskov substitution principle) is what will primarily determine types' relations with each other during applicability testing. To core rule is quite simple — whenever a given type T is applicable (e.g. as a parameter to a function), all of its subtypes need to be applicable as well.

## 3.3 Generic types

If simple types were all to be had, searching would be fairly trivial to implement. Simple checks on whether a type is a supertype or subtype (depending on parameter kind) would enable query matching. However, since version 5, Java supports generic types and functions, i.e. parametric polymorphism [3, Chapter 4.5].

In our model, types can be generic, i.e. they can have type parameters. These are type level variables which can be bound to other types and thus influence matching. These symbolic placeholders can be referenced in the methods of the type and in its supertypes.

```
public interface Collection<E> {

    E any();

    boolean contains(final E elem);

    Iterator<E> iterator();

    ...

}

public class LinkedList<E> implements Collections<E> {

    ...

}

public interface Map<K, V> extends Collections<Pair<K, V>> {

    Collection<K> keys();

    Collection<V> values();

    ...

}
```

*Generic types*

The above example show how type parameters of types can be linked by the in– or output parameters of their methods directly (e.g. `Collection::any`, `Collection::contains`) or indirectly (e.g. `Collection::iterator`, `Map::keys`, `Map::values`). It is also clear how super type declarations can use this linking very similarly directly (`LinkedList` → `Collection`) or indirectly (`Map` → `Collection`).

In a sense generic types can be though of as type constructors rather than types, as they need to be 'evaluated' by actual type arguments to receive usable types. This distinction may be useful to keep in mind, and will be influential when building the data model (Chapter 5).

While the powerful concept of generic types can greatly improve type safety and reduce code duplication, it also causes a great leap in the complexity of matching operations. Types are no longer necessarily easily checked for compatibility — beside sub– and super-type relations, type parameters have to be considered as well.

### 3.3.1 Variance and wildcards

Following Java's model [3, Chapter 4.5.1], these generic type parameters are treated as invariants of the type. This means that given single parameter generic type `T<a>`, and type `X` with super type `S` and subtype `C`; a function parameter of type `T<X>` can only be substituted by `T<X>`, and not by `T<S>` or `T<C>`. To allow flexible but still type–safe function application, *wildcard* type parameters are introduced following Java's example. Wildcard type arguments may be bounded or unbounded.

The single unbounded wildcard (denoted by Java's `'?'`) allows a given `T<a>` generic type (applied now as `T<?>`) to be matched with any other type with a compatible base,

regardless of the argument's parameter. For example `List<?>` will match `List<String>`, `List<Integer>`, etc.

Bounded wildcards may be upper or lower bounded (= their direction) and they reference another type argument as their limit. Their direction describes their variance in regards to their argument limit. Upper bounded wildcards' (e.g. `T<? extends X>`) type arguments are treated as covariants, while lower bounded wildcards' (e.g. `T<? super X>`) type arguments are treated as contravariants. This means in practice that (in the context of the previous types) `T<? extends X>` can be applied `T<X>` and `T<C>`, while `T<? super X>` can be applied `T<X>` and `T<S>`.

## 3.4 Functions

Functions are the individual operations that we wish to search among. Fitting a math–orinted definition, a function is an operation that takes a certain (fix) amount of input parameters and produces a single output. This representation encompasses both static functions and methods. Just like types, functions can also be generic, which means that they have type parameters and their arguments can reference these parameters. The order of a function's input parameters is not important, only their types.

In contrast to Java, the language model has firt–class support for function types, essentially meaning that functions can be passed around as parameters.

# Chapter 4

# Queries

Queries define the way in which suitable operations are searchable. The goal is to create a simple, intuitive language in which the desired functions are describable. In this task more than anywhere else in the design work, I was influenced by Hoogle and its queries, yet significant changes were necessary to accommodate the language to a class–based object oriented environment, and one that is familiar to (Java) programmers.

The base structure of a query is an enumeration of types representing the desired functions' parameters, with an arrow symbol ('->' or '=>') separating the (singular) output from the inputs.

```
Map<String, Integer>, String -> Integer
```

*Simple query for* `Map::get` *(among others)*

This is a slightly modified notation than that used by Hoogle (comma separated inputs instead of arrows, generic arguments between '<' and '>'), but is still fairly close to Haskell's type descriptors that it itself is based on. The changes are there to make the signature more fit for Java–like function description (e.g. -> for 'input' separation doesn't really make sense without currying), and makes the syntax easily understandable for all programmers.

Some other examples are also included here with their Java equivalent signatures. Fully qualified names are also usable, but for simplicity's sake in these examples mostly simple type names will be used.

```
long -> ()
void wait(long timeout);

() -> Properties
Properties getProperties();

() -> ()
void run();
```

*Simple query examples with Java signature aliases*

## 4.1 Generic queries

Most of the times when looking for a specific operation we have concrete values that we wish to work on, thus we want to enter their most specific types as parameters in order to get any function that is applicable. However, sometimes we wish to express in our queries the need for some universal behavior. In these cases we want to use generic queries, whose substance is a bit different than generic types, despite their similar syntactic constructs.

```
Collection<a>, Comparator<a> -> a
```

*Simple generic query fitting* `max` *using a comparator*

I'd wager the query itself is again intuitively understandable — we have a collection of a given type, a way to sort elements of that very type and need a single item. Of course an infinite number of functions *could* match this signature from fairly relevant (e.g. `smallest`, `secondSmallest`, `thirdSmallest`, etc.) to very generic ones, but first let's discuss what the query really means.

When writing a condition utilizing `List<a>` in a query, we mean 'for all possible types of `a`, `List<a>`', i.e. Rank–1 polymorphism.

Type bounds on these free types are also possible:

```
<a : Comparable<a>> Collection<a> -> a
```

*Simple generic query fitting* `max` *on a comparable type*

In this case a restriction is applied to the possible values of the type of `a`, namely that they have to be a subtype of `Comparable` of themselves. When this explicit type bound is not present — similarly to normal type declarations — the root type (e.g. `Object`) is used as an upper limit.

**Type virtualization** To then compare these generic queries to generic functions would be a challenging task, but a simple solution can be utilized. Free type variables (e.g. `a` in the previous example) can be substituted by ad hoc types created in the appropriate position in the type hierarchy. This fact results from the observation that because the origin goal of the polymorphic type was to mean 'all possible types within given restrictions', we can create a new 'virtual' type with exactly those restrictions and use it as our test subject. Since no additional properties are present on these type, if it can be used as a parameter for a given function, than all types compatible with the original type variable are usable; if it cannot be applied than it serves as an exception type, and thus the whole 'forall' clause fails.

Type virtualization allows us to specialize generic queries without losing information and only deal with non–generic queries later on.

## 4.2 Function types

The query syntax provides a self–similar way to easily handle higher order functions independently from language representation. These parameters (in– and output) can be written similar to nested queries in parentheses.

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

(a -> b), (b -> c) -> (a -> c)

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

*Higher order function query fitting* `map`*,* `compose`*/*`andThen`*, and* `const` *respectively*

This representation is language agnostic and its resolution is up to environment specific modules.

# Chapter 5

# Data model

Previously when describing the language model, I have concluded that since my work is not concerned in operation implementation and state, interfaces and classes may be unified as a single category which I referred to as 'type'. In this chapter I will describe the more accurate concepts and definitions used in the actual design of the system.

*Please note that the definitions I will be using are generally distinct from those used in type theory.*

## 5.1 Types and Semitypes

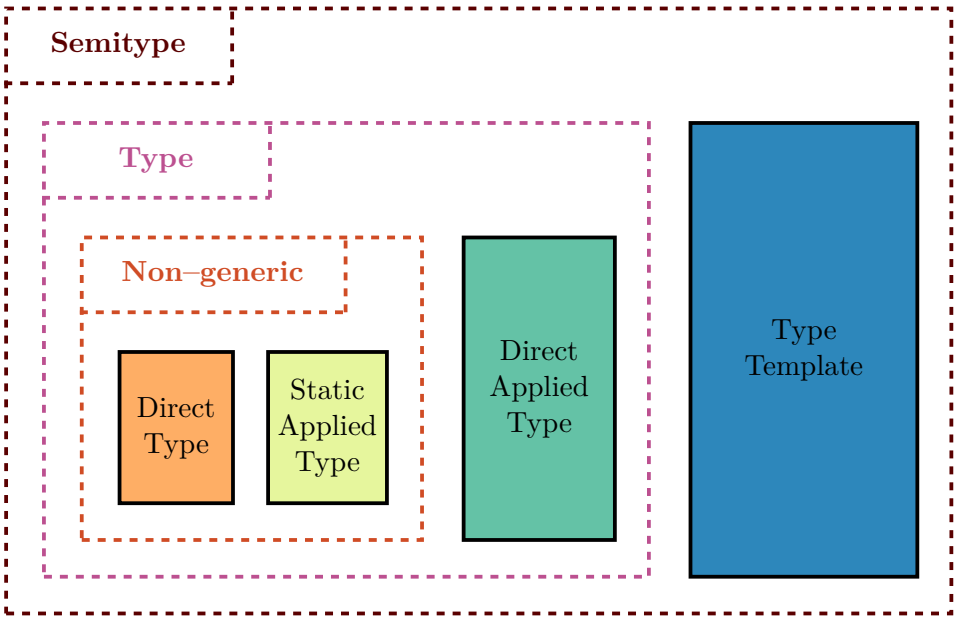


**Figure 5.1:** *Data model*
*(Solid borderd kinds are concrete)*

Semitypes are the general building blocks of the type hierarchy, and encompass Java's classes and interfaces. They provide meta information about themselves based on which they can be identified and compared, and contain information about their supertypes.

Types are a subset of semitypes that represent any value that can be a parameter of a function signature. This excludes type templates which are the equivalent of Java's generic (referential) types, as in this system templates need to be applied type arguments to be usable as parameters. These type parameters may be dynamic, i.e. dependent on some external type variable.

All defined referential types in Java (classes and interfaces) fall into two categories: they are either direct types, or they are type templates.

### 5.1.1 Direct types

Direct types are types that contain no type variables. In Java they are sometimes referred to as non–generic types, but here that has a broader definition. Not having any type variables has the relieving consequence that all of their supertypes must also not have type variables. This makes compatibility checking a fairly simple task on them.

### 5.1.2 Type templates

Type templates — or templates for short — are semitypes that have free type variables. These are upper bounded parameters that need some form of substitution in order to make a type, through a process called application.

In Java's terminology these are generic types, as Java uses type erasure to handle generics instead of compile–time substitution, but from a behavioral perspective this is simply implementation detail that need not be distinguished.

Being semitypes, templates define their supertypes as well, which just as with all semitypes, must be types themselves.

### 5.1.3 Type application

Type application is the process through which a type template's or dynamic type's free type variables are substituted with application arguments. These arguments include other types, (bounded) wildcards, external type variables, and some special elements. During type application an argument must be supplied to all type parameters. There are two main categories of type application: static and dynamic which result in different outcomes.

#### 5.1.3.1 Static type substitution

Static type application removes all external free type variables from a type, thus making a 'Static Applied Type' (SAT) which behaves very similarly to direct types. In fact the two sets (direct types and SATs) make up the category of non–generic types.

Non–generic types are considered to be fully evaluated, meaning that their structure is independent of any outside parameters. Just like direct types, non–generic types are easy to match for compatibility, as their whole type tree may be readily evaluated. In the case of SATs, their application arguments can either be other non–generic types or wildcard arguments.

#### 5.1.3.2 Dynamic type substitution

Dynamic type substitution is the process in which *some* type parameters may become fixed, while others become linked to external type parameters. It is important to note that — just like in the case of static application — all type parameters must be assigned an argument. The difference comes in the kind of arguments: while static substitution requires that all values are static, dynamic substitution allows using dynamic values, such as type parameter references.

Dynamic application works on 'Applicable'-s, i.e. Type templates and 'Dynamic Applied Type'-s (DAT), and applies a set of parameters to them to produce a DAT.

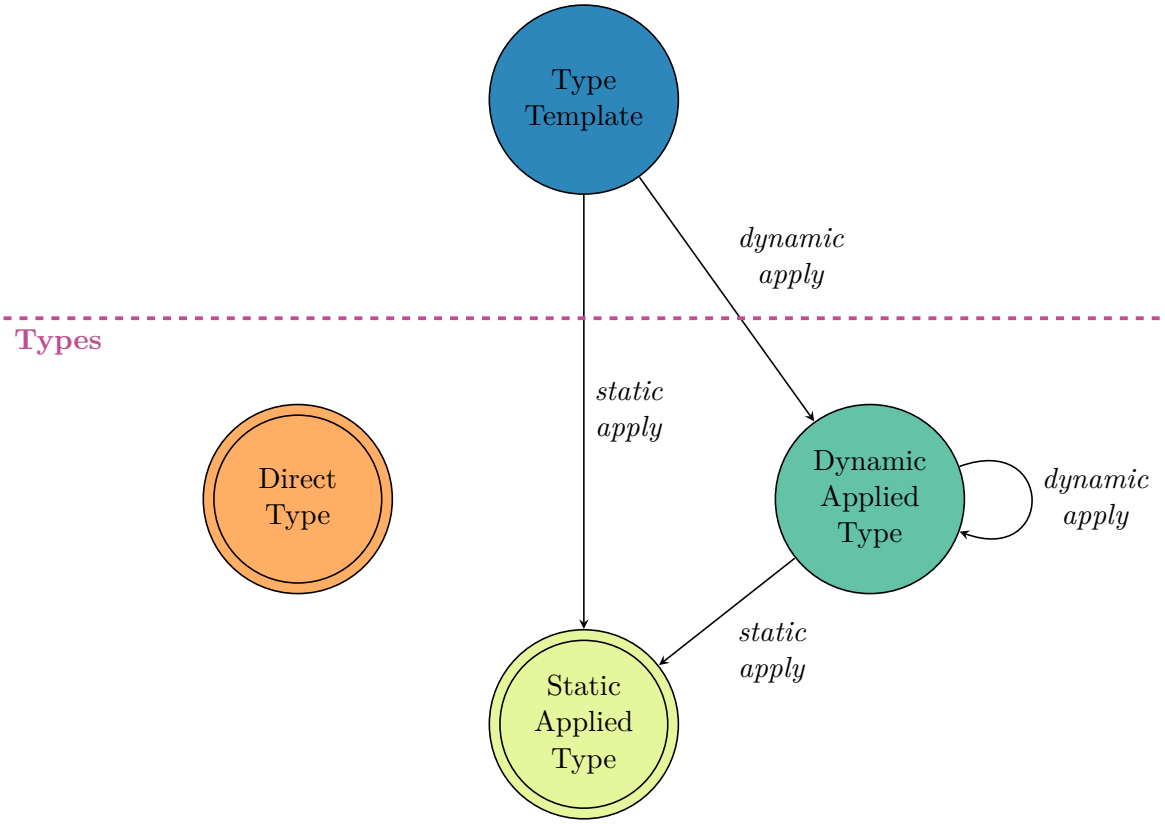


**Figure 5.2:** *Type applicastion graph (Non–generic types with double border)*

### 5.1.4 Static– and Dynamic Applied Types

Both SATs and DATs have a reference type template, of which they are based on, but while SATs have only non–generic types as arguments to that template, DATs contain a mapping that connects outside type parameters as well as non–generic types as parameters.

Consider the following simplified and incomplete codebase:

```
public final class Request {

      // Member definitions

}

public interface Collection<T> {

      // Member definitions

}

public final class Pair<F, S> {

      // Member definitions

}

public interface Map<K, V> implements Collection<Pair<K, V>> {

      // Member definitions

}

public class RequestMap<E> implements Map<Request, E> {

      // Member definitions

}

public class TtlMap extends RequestMap<Integer> {

      // Member definitions

}

public static List<String> sortAlphabetically(Collection<String> values) { ... }

public static <T> List<T> take(List<T> values, Integer size) { ... }
```

*Data model example codebase*

**Classes and interfaces** In this example `Request` and `TtlMap` are both direct types — as they have no type parameters —, while all other classes and interfaces (`Collection`, `Pair`, `Map`, `RequestMap`) are type templates with either one or two type parameters.

The interesting aspect is how these (semi)types reference their supertypes and with them build up a type hierarchy. The first three items are quite simple, as they have no explicit supertypes (the root type set implicitly is a simple direct type), but the bottom three are worth discussing in detail.

`Map` has a single supertype, a derivation of `Collection`. As that is a type template — a semitype, but not a type —, it has to be applied to get a type usable as a supertype. In this case it visibly depends on `Map`'s own type parameters ('`K`', '`V`'), thus it is a DAT obtained by dynamic application. To satisfy `Collection`'s parameters, a single argument must be passed, in the code represented by '`Pair<K, V>`'. This argument is itself a DAT, as it is a dynamic type that depends on external (`Map`'s) type variables. This nested application's base template is `Pair`, and the used arguments are two type parameter references, pointing to the first and second type parameters of `Map`. It is very important to realize that just because a parameter seems identical to a template (in this case the nested `Pair` DAT and

the base Pair template), perhaps even using identical type variable names, only (applied) types can be used as supertypes or type arguments.

Similarly RequestMap has a single supertype, in this case it's a dynamic application of Map with one static substitution (Request) and one dynamic parameter substitution (RequestMap's only type parameter) as arguments.

In the last case, TtlMap being a non–generic types presupposes that all supertypes must be non–generic, and this is of course the case: a SAT super is created by the static application of the RequestMap template with a single direct argument (Integer).

**Functions** Now let's take a look at the two included functions. sortAlphabetically is a non–generic function, i.e. one that has no type variables. All of its parameters (in–, and outputs as well) are non–generic types, in this case specifically, SATs. The single input is a SAT based on the Collection template and the output is based on the List template, both parameterized with a direct type substitution of String.

In contrast, take is a generic function — with a single type variable T — that can utilize DAT parameters as well as non–generic parameters. In this concrete example we see both: values and the output are DATs based on List referencing 'T', while size is a simple direct type (left as the hypothetical 'Integer' to avoid confusion about primitives for now).

#### 5.1.4.1 Type tree

A crucial condition required by compatibility checking during application is that all supertypes of an applicable (DAT or template) must also be applied (when suitable). This ensures that a proper type tree is always present.

In practice this is done in a lazily evaluated fashion to avoid infinite type structures. While the model does require non–cyclic supertype relations (just like in Java's language model), type arguments actually make it possible to create infinitely large types when evaluated unlimitedly. An example code that would result in infinite types are presented here:

```
public interface A<T> { ... }

public class Box<T> implements A<Box<Box<T>>> { ... }

final Box<String> foo = ...;
```

*Infinite recursive type structure*

As mentioned before, the supertypes of this (or any) type build a finite directed acyclic graph, which cannot contain infinite structures. However, observe what happens when evaluating the type of the foo variable. It is a simple Box<String> SAT, with a supertype of A<Box<Box<String>>> SAT (and above that probably Object). However, if we were to carelessly define all types in that SAT when evaluating, we would have to specify the type parameter Box<Box<String>>. The static application of that would require us to evaluate its supertype A<Box<Box<Box<String>>>>. It's easy to see how this results in a recursive infinite structure where all SAT levels have a supertype (a SAT derivative of A) that has a parameter that is equal to itself wrapped in a new Box enclosure.

This can be prevented by only lazily evaluating type parameters of applied types (either simple lazy evaluation or more advanced caching) — because when matching, these infinitely deep types will get compared only to a finite level —, or by loose coupling of types. (See Section 8.2)

### 5.1.5 Type bounds

Whether by the implicit limit of the root type (e.g. `Object`), or a single or multiple explicitly set limit(s), the possible values of a type parameter are always capped. These limits are always required supertypes of the parameter, i.e. upper limits (if one images the type tree with the root at the top) and help operations coerce specific required properties of the parameters. When checking for compatibility, these bounds need to be confirmed as satisfied.

An important aspect here is that these bounds may reference other type parameters, but never form a circle of dependency (they form DAGs). This enables us to always build type parameters from an/the independent one and then use all previously created parameters as parameter substitution in each subsequent type parameter's bounds (creating DATs when referring to other type parameters).

An optional route to handle wildcards is to basically treat them as anonymous type parameters, with specific bounds that need not be exactly determined when matching. This was an early idea, but later on they were handled as a separate kind of type argument for multiple reasons, including to lessen the number of type parameter checks and to allow having SATs containing wildcards.

## 5.2 Signatures and functions

Signatures are basically a list of type parameters and a set of parameters, marked with direction (in/out; essentially variance for matching). Signatures also contain metadata in the form of the parameter names, and implicit '`this`' parameter info. While operation signatures can be generic, query signatures are always non–generic, thanks to type virtualization (Section 4.1).

### 5.2.1 Function types

To satisfy the language model's concept of *function type*s, in the data model signatures may be assigned to semitypes. These signatures describe the given type's behavior as a function. This is closely related to Java's (and other languages') concept of SAM types, and while theoretically any signature can be attached to a (semi)type, currently only SAM types are defined as function types (as their single abstract method).

# Chapter 6

# Managing Java specific behavior

Throughout the design work, an important objective was to ensure the model can be adapted to any programming language with a sufficiently 'Java–like' type–system (e.g Kotlin, C#). This includes all common components being designed from the ground up to not depend on language specific behavior. However, for various reasons including high prevalence and simplicity, Java was chosen to be the main implementation goal. Thus the unique characteristic features of the language have to be accounted for through specialization. These aspects may be a result of any number of historic design decisions caused by the likes of development time–pressure, performance considerations, or various design goals.

## 6.1 Arrays

Arrays are the fundamental collection types supported by C–like languages, that can be built upon to create many different data structures. Because they provide otherwise unobtainable characteristics ($O(1)$ read–write memory access), they are provided as a first–class language feature. While their implementation is different compared to normal classes, their behavioral interface can be modeled as a simple generic type — in fact that is the route that Kotlin takes when interacting with native JVM arrays.

While this approach provides a relatively easy solution, arrays do have another specific quirk about them — inherit covariance. Due to them being created before generics were introduced to the language, they could not depend on wildcard–based use–site variance declarations to enable type–safe producer/consumer semantics. This means that if a function were to take an array of a specific type as parameter (e.g. `Animal[]`), than no array of sub/supertypes (e.g. `Dog[]`) could be applied, regardless of intent. To help avoid type casting, the language designers opted to make arrays covariant. This enables programmers to write functions that use arrays as producers of a given type (e.g `addAll`, `forEach`), or not deal with types at all (e.g. `shuffle`, `arrayEquals`), but also breaks the types safety of arrays (due to the fact that their mutability makes them available to use as consumers).

Thus to process arrays, we have to mark them as implicitly covariant, this ties in closely with the use–site variance declaration of other languages (Scala, C#, Kotlin, etc.) that helps achieve similar goals, albeit in a more type–safe and flexible manner.

## 6.2 Primitives

Similar to arrays, primitives are also a largely historic artifact of the language. Being the only non–referential values in the language, they cause problems due to them not being really part of the type hierarchy, and also their inability to be used as generic type parameters.

These limitations themselves don't affect the search, but Java's solution to them do require workarounds from our part. The basic way primitives are integrated to generic features of the language is (un)boxing, the implicit process through which each primitive type is reversibly, silently converted to its respective 'boxed' type — a proper class that holds the primitive as value. These boxed types (`Integer`, `Float`, `Boolean`, etc.) can then be used in generic type parameters.

This arrangement creates a sort of transparency between the primitive–boxed pairs, where in certain cases they can be considered synonymous.

## 6.3 Function types

Up until version 8, Java did not really have any support for functions as parameters, any kind of higher order function handling had to be handled through wrapper classes (usually anonymous inner classes). However, Java 8 brought with it the introduction of lambda expressions and method references. Implementation–wise these features may look fairly similar to anonymous wrappers, but from a conceptual standpoint they convey behavior much more clearly.

'Single Abstract Methods' — interfaces with exactly one abstract method, and optionally marked with the `@FunctionalInterface` annotation — parameters generally represent higher ordered functions. They may be instantiated using lambda expressions or method references with a compatible type signature as the SAM's abstract method. Common function types have ready–made interfaces in the `java.util.function` package, plus any other type can be easily defined.

Although this provides easy handling from the programmers' perspective, functions still only are a kind of 'second–class citizens' on the Java platform. When processing queries containing function types (e.g. `"List<String>, (String → Integer) → List<Integer>"`), all of these types must be able to match any SAM interface with an appropriate signature function — e.g. `'(String → Integer)'` must match `Function<String, Integer>` and possibly `ToIntFunction<String>` among others.

# Chapter 7

# Query fitting

Having built a suitable data model capable of containing the desired types and a parsing system to fill it with data from software artifacts, a mechanism is required to actually match the functions to the user queries. In this chapter I will overview a relatively simple algorithm for this task. For simplicity's sake I will do away with any potential optimalization and focus on the underlying logic.

## 7.1 Matching criterion

The goal, initially, is to tell if a given function is usable for the given types as represented by a query object. This, in its simplest form, is a binary predicate.

```
fits :: FunctionObject, QueryObject -> boolean
```

*Simplified query fitting as a pseudo signature*

To test the query–function matching, the individual parameters of the functions and queries will need to be matched individually in pairs. Since an operation's parameter order is not significant in any way, these parameters will be paired up in all possible permutations. A single '*pairing*' will thus contain a list of parameter pairs — alongside their *variance* (in/output → co/contravariance). If and only if any of such pairing produces a matching, the function matches the query.

```
Pairing := [{funParam, queryParam, variance}]

fitsPairing :: Pairing -> boolean
```

*Query–function pairing and match*

As described before in Chapter 4, query signatures are always specialized into non–generic variants (using virtual types). This means that their parameters are always non–generic types. In contrast, generic functions can and do contain dynamic types in among their parameters, and also have unbound type parameters. When matching a specific pairing,

the routine will attempt to assign non–generic values to these wild type parameters while simultaniously turning their dependent parameters into non–generic types themselves. This procedure is generally referred to as type–fitting and is very similar to what a modern compiler must do to resolve inferred types.

The method of parameter matching is self–similar in nature, and is demonstrated by the following pseudocode using non–generic types:

```
fitsParam(funParam, queryParam, variance) {
    if (funParam.baseInfo == queryParam.baseInfo) {
        zip(funParam.typeArgs, queryParam.typeArgs).all { funTypeArg, queryTypeArg ->
            fitsParam(funTypeArg, queryTypeArg, INVARIANCE)
        }
    } else {
        switch (variance) {
            INVARIANCE -> false
            COVARIANCE -> {
                queryParam.superTypes().any { querySuper ->
                    fitsParam(funParam, querySuper, variance)
                }
            }
            CONTRAVARIANCE -> {
                funParam.superTypes().any { funSuper ->
                    fitsParam(funSuper, queryParam, variance)
                }
            }
        }
    }
}
```

*Non–generic parameter matching pseudocode*

First the algorithm finds the proper 'common base–type' of the parameters if one exists based on the supplied variance. This is done by iterating on one of the arguments' (as selected by variance) super types until the the types are from the same base, e.g. they are the same direct types, or are instantized from the same template. Then provided a match is found, their type arguments (if any is present) are paired up and tested much the same way invariantly.

This describes the simplest method a matching that indeed works only on non–generic types, but can be built upon to reach complete coverage of out domain.

### 7.1.1 Wildcard parameters

Wildcard parameters (Section 3.3.1) can be handled relatively easily. The unbounded wildcard may be 'matched' to any query parameter regardless of what it is, while bounded wildcards can be matched based on their direction. All that needs to be done, is that when matching type arguments we come across a bounded upper limited wildcard with a type argument of type `T`, then we can resume matching with `T` as usual, but with covariance instead of invariance.

As an example scenario, let's use the following data:

```
// Types:
type Person
type Boss : Person
type List<a>

// Function to test:
addPerson :: List<? extends Person>, Person -> ()

// Query:
List<Boss>, Person -> ()
```

*Example wildcard scenario*

The second input– and the output parameter pairs are naturally matching, but after accepting the two `List` types we would match their arguments. Without the wildcard, this would fail of course, since `Person` is different from `Boss`, but when matching `? extends Person` to `Boss` we check `Person` to `Boss` with *covariance* instead of *invariance*, we treat them as normal input parameters. Now due to the fact that `Boss` is a subtype of `Person`, the arguments will match and so will the whole query.

### 7.1.2 Generic parameters

When dealing with generic functions and thus generic function parameters, matching becomes a bit more difficult. The result of matching two parameters is no longer a binary decision, instead it can result in one of four results:

- **Fit** The two fit just like in the previous method.
- **Unfit** The two cannot be matched, either like before, or due to a type parameter constraint.
- **TypeArgumentUpdate** In order to continue fitting, a free type variable must be bound to a given (non–generic) type.
- **Uncertain** It is impossible to say at this point whether the pair fit.

The first two options are mostly trivil, but the latter two justify a little bit of in–depth revision.

**Type argument updates** are resulted when a given type variable becomes bound due to the necessity of fitting. Think of trying to fit `List<T>` to `List<String>`. It can be done, but only if `T` becomes fixed as `String`. After this result the function type has to be reevaluated with the given type argument now in place and matching has to start again. Keep in mind that since this result may originate from any argument 'depth', it does not mean that after the type argument application the pairing will fit. Let's visit the following example:

```
// Function to test:
replaceByMap :: <T> Map<T, T>, List<T> -> List<T>

// Query:
Map<String, Person>, List<String> -> List<Person>
```

*Example type argument update*

When fitting, the first pairing (`Map<T, T>` $\longleftrightarrow$ `Map<String, Person>`) will result in a type argument update which tells us to bind `T` to `String`. After we proceed with the update our function's signature changes as such:

```
// Function after type argument binding:
replaceByMap' :: <> Map<String, String>, List<String> -> List<String>

// Query:
Map<String, Person>, List<String> -> List<Person>
```

*Example type argument update*

It is clear that the next round of fitting will result in failure due to an incompability of the `String` $\longleftrightarrow$ `Person` pair.

**Uncertain** matching results arise when — as the name suggests — a pairing is impossible to evaluate at the moment. This can happen if a bounded wildcard parameter is matched (e.g. `List<? extends Person>` $\longleftrightarrow$ `List<a>`), or if a type parameter's bound references another type parameter (e.g. `<T, P extends List<T>>` `P` $\longleftrightarrow$ `String`).

When encountering uncertain sub results, they are simply skipped for the time being, but matching must be run until no skips happen.

## 7.2 Qualified matching result

A useful idea that is only explored in theory for now is introducing (scalar) qualification to the matching result. This way if a query fits a certain function, the result also contains a dimensionless value that indicates the *closeness* of the fit. This can then be used to order fitting functions to emphasize more favorable results.

The need for result sorting arises from the fact that overly general functions may be returned for specific queries, and overwhelm the user. Although it is an inherently heuritic idea, it can greatly enhance the user experience. The values for specific results would have to be calculated by individual pairing match results based on how close the parameters are in the type–tree.

# Chapter 8

# Validation of results

In this chapter I will describe the technical specifics of the project and some design decisions undertaken. The prototype system implemented is mostly a real–word counterpart and realization of the previously described data model (Chapter 5) and query fitting methods (Chapter 7).

## 8.1 Environment

While the main target of the project continues to be the Java programming language, the code itself is written in Kotlin. This is mainly the result of personal preference, and the choice of starting out working with Java is based on a couple of factors.

- Java is one of the most commonly known and used programming language [2][4], a sort of lingua franca of object oriented programming — if not programming in general.
- Its type system is fairly simple and shares a lot of common attributes with those of many other languages.
- It is the 'first–class citizen' language of the JVM, which is mostly built around it.

The system is designed to be a modular application built using Apache Maven.

## 8.2 Type coupling

As with most data heavy applications, the model we store our data forms the foundation upon which we can build buisness logic. While the data model has previously been discussed (in Chapter 5), the crucial design step of connecting individual types have not yet been detailed. The main decision lies with choosing between a loosely coupled solution and a tightly coupled one.

### 8.2.1 Tight coupling

Tight coupling in our case referres to the use of strong linking between types. This essentially means ordanary references when pointing to a type's supertypes or type arguments. This have been the first prototype's implementation and served well for simple cases.

The main benefit of this approach is simplicity and performance *while using the data.* The complete type hierarchy is always present to us for use in a hassle–free tree. Lookups are as fast as they can be in our system, 'not found' kind errors are impossible to come across.

However, this design also brings problems, one of which is harder parsing. Since we are building an actual type tree, parsing has to be done in an orderly fashion, circular– and self–references (with type arguments these are very much possible) have to be handled carefully. These problems can largely be sidestepped by abandoning immutability in our model, but that is a high price to pay as it opens up doors for later misuse of the model. Nonetheless, these problems can be and were solved by various methods, including the use of lazy references.

The real problem of tight coupling is that it makes artifact level handling of sets of types virtually impossible. The JVM and its artifacts are build with dynamic linking in the focus, which makes individual modules (usually JARs) separate, distinct entities. It is our goal to simulate this behaviour and allow the loading, and unloading specific artifacts, and defining custom domain spaces for searching.

Handling different versions of artifacts is also extremely hard to achieve using tight coupling, as demonstrated by the following example. Let's say that artifact `'A'` depends on artifact `'D'` with a major version `1`, i.e. `1.0.0 <= version < 2.0.0`. Thus if there exists a `'D'` artifact with versions `1.0.0`, `1.1.0`, `1.2.0`, `...`, `1.15.0`, then a type `T` in `'A'` may reference 16 unique (but API–wise the same) types from each versioned `'D'` modules. Due to tight coupling, however, `T` can only truly reference a single type, whichever `'D'` version it's in. To solve this problem either only a single version of each artifact should be allowed, or there should be multiple versions of artifacts with all their dependencies' possible permutations. Neither of these solutions are practical or acceptable.

### 8.2.2 Loose coupling

To solve the aforementioned problems, the tightly coupled data model was replaced by a loosely coupled one, in which types only *describe* their referred dependencies by their information, i.e. info and optinally info of type arguments.

This way handling modules is made almost trivial and type parsing bacomes much easier, but performace and usage complexity suffers. Every supertype or type argument lookup will result in lookups with a type resolver that actually finds types based on their descriptors. This is not only slower, but also introduces 'not found' error possibilities — althouh to be fair, the same problems are mirrored by the actual usage of these artifacts.

### 8.2.3 Hybrid coupling

'Hybrid coupling' is a combined way of handling type linkage. In this final aproach, a trasparent proxy system is used, with which types may reference others either directly or indirectly. This method retains most of the complexities of both previuous attempts (and even adds some of its own), but also allows for quick lookups *and* separate artifact handling when used properly.

The core idea is to link individual modules tightly internally, and loosely to the outside world. This way common intra–module connections will remain to be fast, while inter–module connections remain to be practical and flexible. Indirect linking is also used within modules to replace some of the less idiomatic workarounds of the original, tightly coupled version such as those used for SAM types (Section 6.3) or certain type arguments.

## 8.3 Type– and operation parsing

Type– and operation parsing is the mechanism of processing artifacts and turning them into suitable data as outlined by the language model. It is designed to be able to accommodate multiple formats and sources, including program sources, compiled binaries, or documentation.

A concrete module to process compiled JAR artifacts was created, this has the benefit of not needing sources and may work on non–java code. This is a two phase procedure that consists of extracting information and building the model. The two parts can, too, be substituted with other implementations.

**Info extraction** Firstly, we need to obtain the basic information of types and their connections as minimal parsed data — i.e. simple decoupled types referencing each others' names. This part is implemented using the ASM code manipulation library and an ANTLR parser.

A `ClassVisitor` visits all types and examines their internal signature — the normalized descriptor of Java's bytecode. This is done using an ANTLR parser generated from a grammar descriptor by the ANTLR maven plugin. This allows the type–safe processing of the intermediate structure of the signature.

**Nested classes** A big challenge was the handling of nested classes, especially non–static classes nested in generic outer classes. These types are little more than syntactic sugar constructs in Java, but their signature is indistinguishable (at least using ASM) from static nested classes. This creates a problem due to the fact that they are dependent on the type variables of their wrappers, but this fact is not visible from the descriptors. In the end, a workaround solution was realized where during inspection the visitor class keeps track of the compiler generated synthetic '`$this#`' references to wrapper type objects to determine if the visited class was in fact non–static nested.

The result of this initial phase is a collection of all semitypes (generic–, and non–generic Java types) represented by their basic data and information on their supertypes. This data is represented internally as a trie (or prefixtree) on the package names.

**Type model building** The second phase is the actual turning this parsed data into values. This is done in an iterative fashion where all 'ready' types — types whose all dependencies are already built — are continuously built and removed from the set of waiting types.

This process always finishes due to the model's requirement of non–cyclic super–sub relations and the fact that type arguments are lazily evaluated and thus need not be ready to complete a type.

## 8.4 Query parsing

Similar to type parsing, query parsing is mainly done using a generated ANTLR parser created by a custom grammar descriptor. This language defines valid queries which are then made into signature types using type lookups on defined type repositories.

# Chapter 9

# Result and conclusion

Throughout my research I have designed the system and implemented the basic core functionality. As of now, the framework is capable of parsing compiled Java codebases including the standard library (JCL) and other third party libraries in various formats. These components can build upon each other (e.g. reference types from other modules), yet can be treated more or less as independent items.

Maven artifact resolution has also been added. This feature allows the loading of artifacts from maven repositories including loading their dependencies as well.

Type– and function parsing are close to being feature complete and can process the tested artifacts (including JCL) with only a handful of skipped edge cases. Right now a 'cold start' with loading all JCL items and some other libraries (from apache commons) loads ~20,000 semitypes and ~130,000 functions in a matter of seconds on a normal laptop. This is well within the performace requirements of real–world usage.

Query parsing is considered feature complete as of now, searches are evaluated on the loaded codebase in seconds.

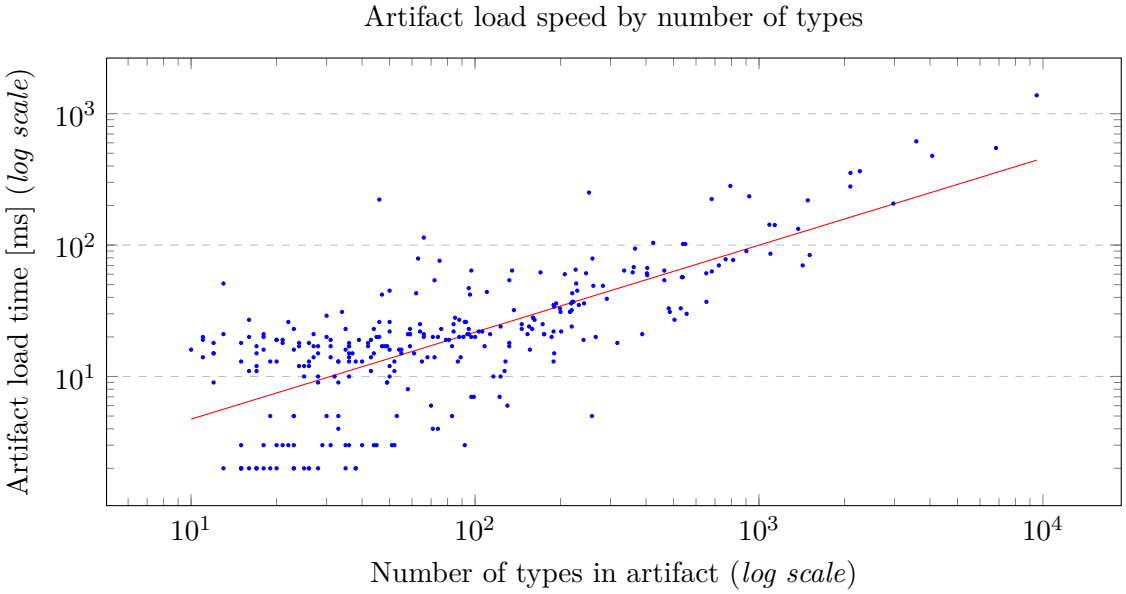


**Figure 9.1:** *Load time by (semi)type count, based on 300 randomly sampled Java artifacts from Maven central (Evaluated on Intel 4720HQ with 8GB RAM)*

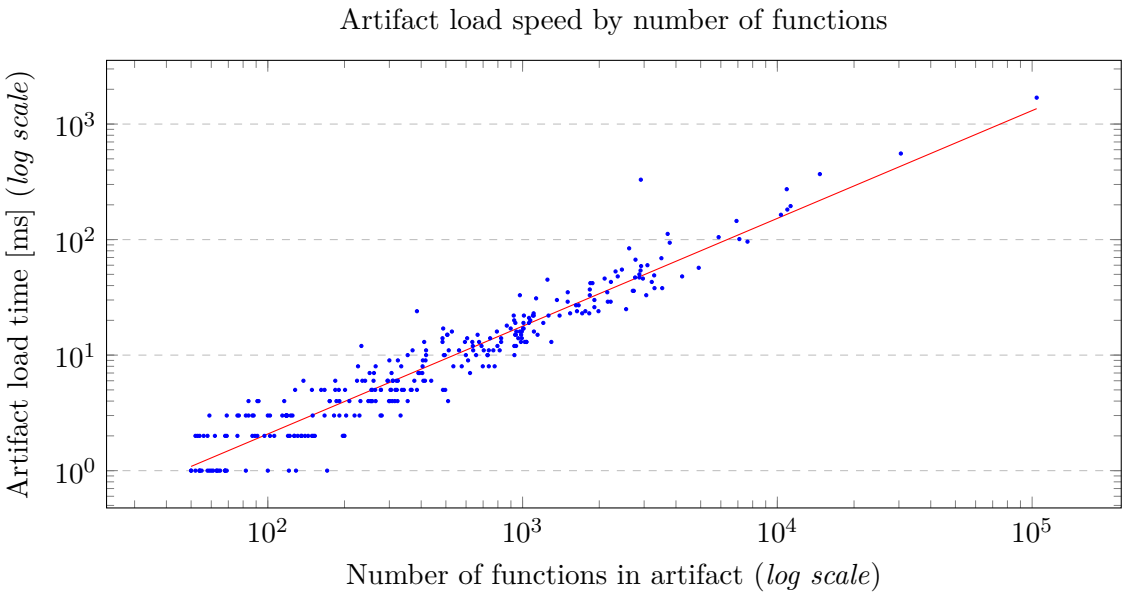


**Figure 9.2:** *Load time by function count, based on 300 randomly sampled Java artifacts from Maven central (Evaluated on Intel 4720HQ with 8GB RAM)*

## 9.1 Core logic

The designed core matching logic seems to be working as intended, with some extra specializations (as mentioned in Managing Java specific behavior) still needed for a more user friendly experience.

Parameter– and function name based filtering is also to be added later on, but these features will depend on much more commonplace features — e.g. plain text search.

## 9.2 Optimization

Currently, while performance seems to be encouraging, I believe it can nonetheless still be improved by several magnitudes. Narrowing the search–space, improving object reuse and caching will make a significant upgrade in speed and memory consumption.

## 9.3 Future work

There remains a great number of features and capabilities that will be added to the system.

**Composite matching** Composite function matching — i.e. fitting a combination of multiple functions — remains a challenge that will probably necessitate large optimizations before being productively viable.

**Tuple types** Similairly to function types, tuple types cound be added to the system as a way to represent generic value classes. This way simple data holders could be referenced

without their proper names. These types would — similairly to SAMs — have to be attached their tuple descriptor while parsing.

```
// Map interface
public interface Map<K, V> {

    static interface Entry<K, V> {

        // getters and setters

    }

    Set<Map.Entry<K, V>> entries();

    // ...

}

// Query to match Map::entries without tuple types
Map<String, Person> -> Collection<Map.Entry<String, Person>>

// Query to match Map::entries with tuple types
Map<String, Person> -> Collection<(String, Person)>
```

*Tuple type example*

**Language support** As Java processing is becoming feature complete, other languages can be dealt with, Kotlin being the first choice. Later on additional JVM languages can be targeted, then even different ecosystems (e.g. CLR through C#) may be experimented with.

**Web interface** The last component of the system is a simple web interface that will allow users to query common dependencies online without installing anything. A basic version of this appears to be simple enough to make.

## 9.4 About the project

The concrete implementation is hosted on Github under the working title of 'Function-Search' for now. It is accessible at `https://github.com/mktiti/FunctionSearch`.

# Acknowledgements

I would like to thank my advisor, dr. Péter Ekler, for his encouragement and help in researching, implementing, and documenting the project.

# Bibliography


[1] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 681–682, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593491X. DOI: 10.1145/1176617.1176671. URL https://doi.org/10.1145/1176617.1176671.

[2] TIOBE Software BV. Tiobe index for september 2020, 2020. URL https://web.archive.org/web/20200929231011/https://www.tiobe.com/tiobe-index//.

[3] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification: Java SE 8 Edition*. Java Series. Addison-Wesley, Upper Saddle River, NJ, 5 edition, 2014. ISBN 978-0-13-390069-9. URL http://docs.oracle.com/javase/specs/.

[4] Stack Exchange Inc. Stackoverflow 2020 developer survey - most popular technologies, 2020. URL https://insights.stackoverflow.com/survey/2020/#technology.

[5] Mart Lubbers and Camil Staps. Cloogle, 2016. URL https://cloogle.org/.

[6] Neil Mitchell. Hoogle: Finding functions from types, May 2011. URL https://ndmitchell.com/downloads/slides-hoogle_finding_functions_from_types-16_may_2011.pdf. Presentation from TFP 2011.

[7] Junpeng Ouyang and Yan Liu. A novel type-based api search engine for open source elm packages. In *Proceedings of the 2019 3rd International Conference on Computer Science and Artificial Intelligence*, CSAI201, page 294–298, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376273. DOI: 10.1145/3374587.3374633. URL https://doi.org/10.1145/3374587.3374633.

[8] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991. DOI: 10.1017/S095679680000006X.

[9] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 166–173, 1989.

[10] Lukas Wegmann. Scaps: Scala api search, 2015. URL http://about.scala-search.org/.