TypeSearch: Type-Directed API Search For All

Master's Thesis

Department of Computer Science
Eastern Switzerland University of Applied Sciences

Spring Term 2023 – Fall Term 2023

Author:	Marc Etter
Advisor:	Prof. Dr. Farhad D. Mehta
Project Partner:	IFS (Institute for Software, OST)
External Co-Examiner:	Joachim Breitner

Abstract

Software developers spend a lot of their time finding and composing pre-existing functions from various libraries. Almost all developers today use general-purpose search engines for this search. Specialized search engines such as *Hoogle* additionally use type information to improve this search, and have been successful for some typed functional programming languages. The options currently available for type-directed search for mainstream object-oriented languages is limited. Existing approaches for these languages do not have first-class support for subtyping or parametric polymorphism. The splitting and composition of a desired functionality into and from a number of pre-existing functions is also a task that needs to be done manually. In this Master's Thesis we present a proof-search-based approach to type-directed search with first-class support for subtyping, parametric polymorphism, splitting, and composition. The approach is language agnostic, and can be specialized to simultaneously support multiple typed object-oriented languages. Given that most mainstream languages fall under this category, this approach would extend the benefits of type-directed search to the majority of programmers. As a proof of concept, we provide a running implementation of the core language-agnostic approach and extend it to support the Java programming language. Further extensions would allow the tool to simultaneously support multiple programming languages using the same query syntax.

Contents

1 Introduction

Nowadays, software development consists largely of combining and integrating functionality from third-party libraries and frameworks. As such, expertise in a programming language often means having deep knowledge about a wide array of the language's ecosystem. To successfully find the appropriate library functions for the task at hand, developers consult search engines, such as *Google*[1], *StackOverflow*[2], and more recently, *ChatGPT*[3].

The need for online resources to search for library functions clearly exists. In certain programming language ecosystems this need has inspired the creation of dedicated tools.

This Master's Thesis presents two chief contributions:

1. A proof-search-based approach, using the Curry-Howard correspondence [Pie02, §9.4], for finding API functions based on a user-supplied approximate type signature, with full support for parametric polymorphism and subtype polymorphism. We name this approach *TSaPS* (**T**ype **S**earch **a**s **P**roof **S**earch).

2. *TypeSearch*, a proof-of-concept implementation of *TSaPS*. *TypeSearch* is designed with a modular plugin system so that support for new programming languages and different storage backends can be added easily. *TypeSearch* ships with a plugin implementation for Java, and a simple in-memory storage backend. *TypeSearch* is publicly accessible at `https://typesearch.dev`.

Chapter 2 presents a programming language agnostic data model that is used as an intermediate representation of API functions. This data model enables the core of *TSaPS* and *TypeSearch* to be implemented independently of the target programming language. Chapter 3 then shows how *TypeSearch* parses API functions and extracts the necessary information into this data model.

Chapter 4 describes the query syntax used by *TypeSearch*. This query syntax is designed to be intuitive for developers used to object-oriented programming languages. It is also designed to be language-agnostic, so that developers can reuse the same query regardless of the programming language they are searching in.

Chapter 5 first gives a brief introduction into the Curry-Howard correspondence and proof theory, and then details the theoretical foundations of *TSaPS*. This chapter also discusses where *TSaPS* differs from conventional proof theory to accommodate the challenges of type-directed API search.

Chapter 6 goes into further detail regarding *TypeSearch* and how we efficiently search for valid proofs. In the same chapter we also discuss how proofs are ranked to provide a useful search engine experience. Performance measurements of *TypeSearch*, both in terms of speed and result quality, are shown in chapter 7.

Chapter 8 details notable examples of existing work in this field. This chapter also describes the various underlying approaches that have been used, with their respective advantages and disadvantages. All of these tools are dedicated to a specific programming language, which was part of the motivation behind *TSaPS*.

Chapter 9 discusses additional ideas to improve the effectiveness of the *TypeSearch* implementation, as well as shortcomings of the current implementation. Finally, chapter 10 summarizes the results of the previous chapters.

[1] `https://google.com/`
[2] `https://stackoverflow.com/`
[3] `https://chat.openai.com/`

This technical report goes into implementation details for every topic. We also plan to publish a more succinct paper on *TSaPS* and *TypeSearch* that explains the key concepts [EM24].

2 Modelling API Contents

The Abstract Language Data Model (ALDM) is a data model storing information about the contents of APIs. The ALDM attempts to consolidate various type implementations across programming languages into a single data model. However, the ALDM is *not* a rigorous type system. Rather, it stores information *about* the types encountered in a particular language. The goal of the ALDM is not to fully map every language's type system into a common type system that fulfills all the rigorous properties that would be required by a compiler. The goal of the ALDM is to store information about a language's types *accurately enough*, so that an efficient and useful type-directed search can be performed. The ALDM does not make any assumptions about how this information is used during search, so the ALDM tries to capture as much information about the API contents as possible. The ALDM is an important center piece of the *TypeSearch* implementation, but not integral to the concepts of *TSaPS* presented in chapter 5.

Most type concepts of modern object-oriented programming languages can be classified as well-known theoretical type constructs from the system of higher-order bounded quantification, as shown by [Ett20]. This type system is denoted as $F^{\omega}_{<:}$ ("F-omega-sub") [Pie02]. The ALDM uses the formal names from type theory when describing type constructs, even if the ALDM does not completely accurately reflect the theory. These differences are described in later sections.

The ALDM consists of three data structures: type definitions, type references, and type mappings. There is also subtype information being extracted and stored, but the subtype storage implementation depends on the particular subtype system in use. Each of these are described in further detail in the following sections. The illustrations accompanying the descriptions of the data structures make use of the Unified Modelling Language (UML) [BRJ05]. An example of Java source code being mapped into the ALDM is shown in figure 3.2.

2.1 Type Definitions

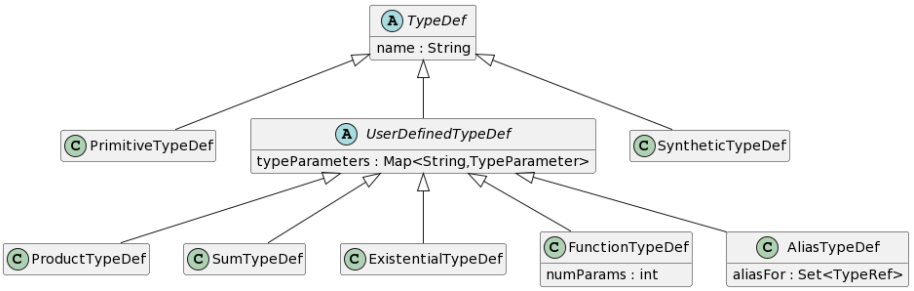

Figure 2.1: Data model for type definitions

A `TypeDef` describes the definition (or declaration, depending on source language) of a type. Type definitions are regarded as unique within a programming language. As such, there is a bijective mapping of every type in a language to a `TypeDef` in the ALDM. Type definitions are further specialized as shown in figure 2.1. These specializations

(subclasses) serve as additional hints during querying and are further described later on.

2.1.1 PrimitiveTypeDef

A `PrimitiveTypeDef` represents the type definition of a primitive type. Usually, primitive types are defined by a programming language specification itself and directly built into the compiler of that language. Since the list of primitive types is well-defined and usually finite, it is feasible to simply hard-code the full list of `PrimitiveTypeDef`s into the appropriate language plugin (see chapter 3). Any type defined through source code (i.e. not built into the compiler) is instead a `UserDefinedTypeDef`, whose further specializations are described in the following sections.

2.1.2 ProductTypeDef

A `ProductTypeDef` roughly represents an algebraic product type. In object-oriented programming languages this concept most closely resembles classes. However, classes usually are the combination of three components: a data structure ("member fields"), methods operating on that data structure ("member functions"), and inheritance information (parent classes and protocols/interfaces). Strictly speaking, an algebraic product type only covers the data structure aspect of a class. For the purposes of type-directed API search, however, it is sufficient to think of OO-classes as product types. The member functions of a class are indexed as individual `FunctionTypeDef`s, as described in section 2.1.5. The inheritance information is tracked in a separate data structure, detailed in section 2.7.

2.1.3 SumTypeDef

A `SumTypeDef` represents an algebraic sum type. In languages like Java or C# these are enumerations. However, for example TypeScript has so-called union types, which are more general sum types.

2.1.4 ExistentialTypeDef

An `ExistentialTypeDef` describes existential types. As explained by Pierce in [Pie02], existential types can be interpreted as packages. This can be further extended to protocols (commonly also known as "interfaces") as discussed in [Ett20]. Therefore, e.g. Java interfaces are stored as `ExistentialTypeDef`s in the ALDM.

2.1.5 FunctionTypeDef

`FunctionTypeDef`s express unique definitions of functions. In the ALDM, functions are uniquely defined by the number of input parameters. Every input parameter type is modeled as a type parameter which is then substituted with a `ConcreteTypeArgument` for actual functions. This concept is illustrated in figure 2.3.

2.1.6 AliasTypeDef

Lastly, `AliasTypeDef`s represent type aliases. These are typically explicit type aliases, if a programming language supports such a feature. However, intersection types are

also represented as “anonymous” type aliases. Such anonymous type aliases arise e.g. in Java or TypeScript when using a construct such as `Mammal & Flying` to denote a type that must satisfy both the `Mammal` and `Flying` requirements. Since an anonymous type alias cannot be disambiguated by its name, it is uniquely identified by the set of types it references. Hence, `Flying & Mammal` and `Mammal & Flying` would both reference the same `AliasTypeDef` in the ALDM.

2.2 Type References

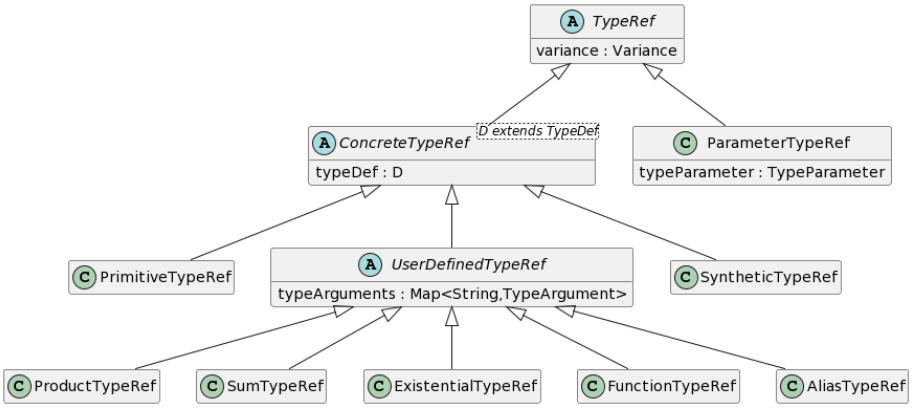

Figure 2.2: Data model for type references

Type references are the counterpart to type definitions. Whenever a type is used (“referenced”), this is modelled using a `TypeRef`. In contrast to type definitions, type references are not necessarily unique. If a type is used in the same way in multiple places, multiple identical `TypeRef`s will exist. Type references therefore capture the information about how a type is used in a particular context. An example of such a context is the declaration of a variable like `List<String> strings`. The variable declaration of `strings` references the type definition for `List` with a concrete type argument `String`.

The `TypeRef` data structure is nearly identical to that of the `TypeDef`s. This is most obvious when looking at the subtree rooted at `ConcreteTypeRef`. This subtree exactly reproduces the structure of the `TypeDef` data model. Every specialization of `TypeDef` has a corresponding `TypeRef` node, which models a reference to that particular kind of type definition. The main difference is the addition of nodes at the root of the tree. This enables the model to differentiate between concrete types being referenced and the usage of type parameters.

2.2.1 FunctionTypeRef

`TypeRef`s only store information about the referenced `TypeDef`, as well as potential type arguments. However, `FunctionTypeRef`s are a bit more nuanced in how they are used and how they differ from `FunctionTypeDef`s.

`FunctionTypeRef`s are slightly unintuitive, as the definition of e.g. a member function is modelled as a function type reference, and not a function type definition as one might expect at first. This is more obvious in a language like TypeScript, where a function

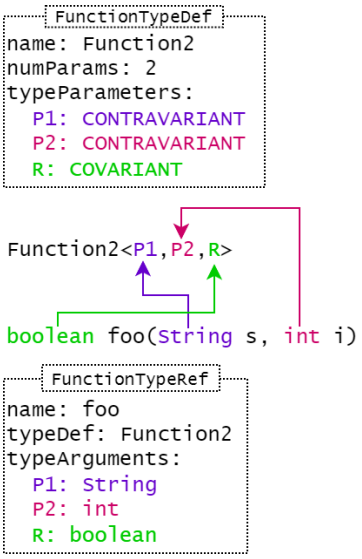

Figure 2.3: FunctionTypeRef referencing FunctionTypeDef with type arguments

can be viewed as a regular variable where the type happens to be that of a function. In that regard, such a variable is no different from a non-function variable, in that the type of such a function variable is simply a reference to a type defined elsewhere (the `FunctionTypeDef`).

In particular, the function type definition referenced is decided solely by the number of input parameters to the function. The concrete types (or possible type parameters) are then passed as type arguments to the type definition of the function which accepts that number of input parameters. This concept is inspired by Java's functional interfaces and especially Scala, where every function is assigned the type `FunctionN`, where `N` is the number of input parameters. For example, `Function2[-T1,-T2,+R]` in Scala has two input parameters, which are denoted with the type parameters `T1` and `T2` (each being contravariant). The return type is similarly represented with the type parameter `R` (covariant). An example of such a `FunctionTypeRef` is shown in figure 2.3.

2.3 Type Parameters

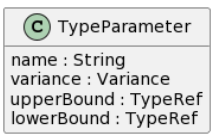

Figure 2.4: Data model for type parameters

Type parameters model the free variables of parametric polymorphic types. These type parameters are often also known as "generics". Type parameters have upper and lower bounds, which default to $\top$ (`TOP`) and $\bot$ (`BOTTOM`), respectively. $\top$ and $\bot$ are artificial types added in the ALDM for the type that is a supertype of all other types ($\top$)

and the type that is a subtype of all other types (⊥). In addition to a type parameter's bounds, it also has a name and a variance. The name is used to correlate which type argument is substituting which type parameter. The variance is used for languages that implement declaration-site variance, such as Scala.

2.4 Type Arguments

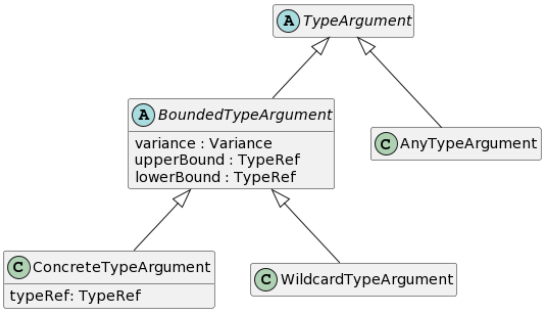

Figure 2.5: Data model for type arguments

While type parameters describe how a type definition can be parameterized, type arguments describe how such a parametrizable type is used. Similar to type parameters, (bounded) type arguments have a variance and upper and lower bounds. These are used for programming languages implementing use-site variance, such as Java.

Type arguments are split into three main kinds: concrete type arguments, wildcard type arguments, and "any" type arguments. Concrete type arguments refer to a specific `TypeRef` being used. Note that a `TypeRef` could be a `ParameterTypeRef`. This means that a type is parameterized based on a type variable declared elsewhere. Wildcard type arguments describe the case where a wildcard token is used as the type argument, as opposed to an actual type (variable). For example in Java, a wildcard type argument (with upper bound) is used like `List<? extends Number>`. This construct means "a `List` where each element is a subtype of `Number`", making the `WildcardTypeArgument` covariant. Lastly, an "any" type argument is a placeholder similar to a wildcard type argument, but without any constraints whatsoever on the type that eventually replaces it. `AnyTypeArgument`s are only introduced through wildcard characters in a query, not through API source extraction. See chapter 4 for more details. The distinction between a `WildcardTypeArgument` and an `AnyTypeArgument` becomes important when dealing with subtyping and other type comparisons.

2.5 Value Definitions

A value definition is similar in concept to a type definition. In the context of the ALDM and type-directed API search, a "value" is best summarized as "any symbol that could potentially be a search result". What exactly this entails depends on the particular language plugin. However, `ValueDef`s usually include constructs such as publicly accessible functions and constants. The most important attributes of a `ValueDef` are the ID, the type reference, and the module. The ID is a unique identifier for this

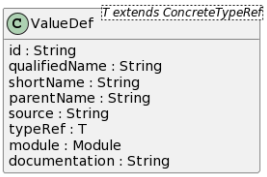

Figure 2.6: Data model for value definitions

`ValueDef` and is usually built from the fully-qualified name of the symbol and information about in which module the symbol is defined.

Figure 2.7 shows the data structure tree for a single `ValueDef` entry. This `ValueDef` represents the method `java.util.List#size`, which returns the number of elements in a list as a primitive integer. The `ValueDef` is red, `TypeRef`s are green, `TypeDef`s are blue, the `TypeArgument` is lavender, and the `TypeParameter` is yellow. The arrows in this diagram represent object references, with the variable name as a label.

2.6 Type Mappings

In many programming languages there exist types that should be considered equivalent for the purposes of searching, even though they are not equivalent or even related through subtyping. Examples of such types in Java are primitives and their boxed counterparts, or various functional interfaces.

To support these additional equivalence relations, the ALDM makes use of so-called type mappings. Type mappings group types into equivalence classes and select one of the types in this group as the primary representation. Then, when comparing two types, each type is first mapped to the primary representation of their respective equivalence class. Thus, if two different types are part of the same equivalence class, the type comparison will be done using the same representation type, resulting in a match. This leads to a partitioning of the type space, where each type within a partition is mapped to a single "key type" within the partition. Types without any provided type mappings are trivially mapped onto themselves.

For example, both `Function<String,Boolean>` and `Predicate<String>` are part of the same equivalence class and ultimately map onto the generalized function representation `String -> Boolean`. This means a query by any of the three expressions would find matches of either type. Figure 2.8 shows an example of such a type comparison using type mappings.

The definition of these equivalence classes, as well as which type to use as the primary representation, is up to the individual language plugins. The specific Type Mappings for Java are described in section 3.2.3.

2.7 Subtyping Information

Accompanying the ALDM is a subtyping information registry. Depending on the subtyping scheme used by a programming language, the subtyping information might be stored in a different representation. *TypeSearch* provides an interface for all subtype registries, called `SubtypeRegistry`. The search implementations in *TypeSearch* only make

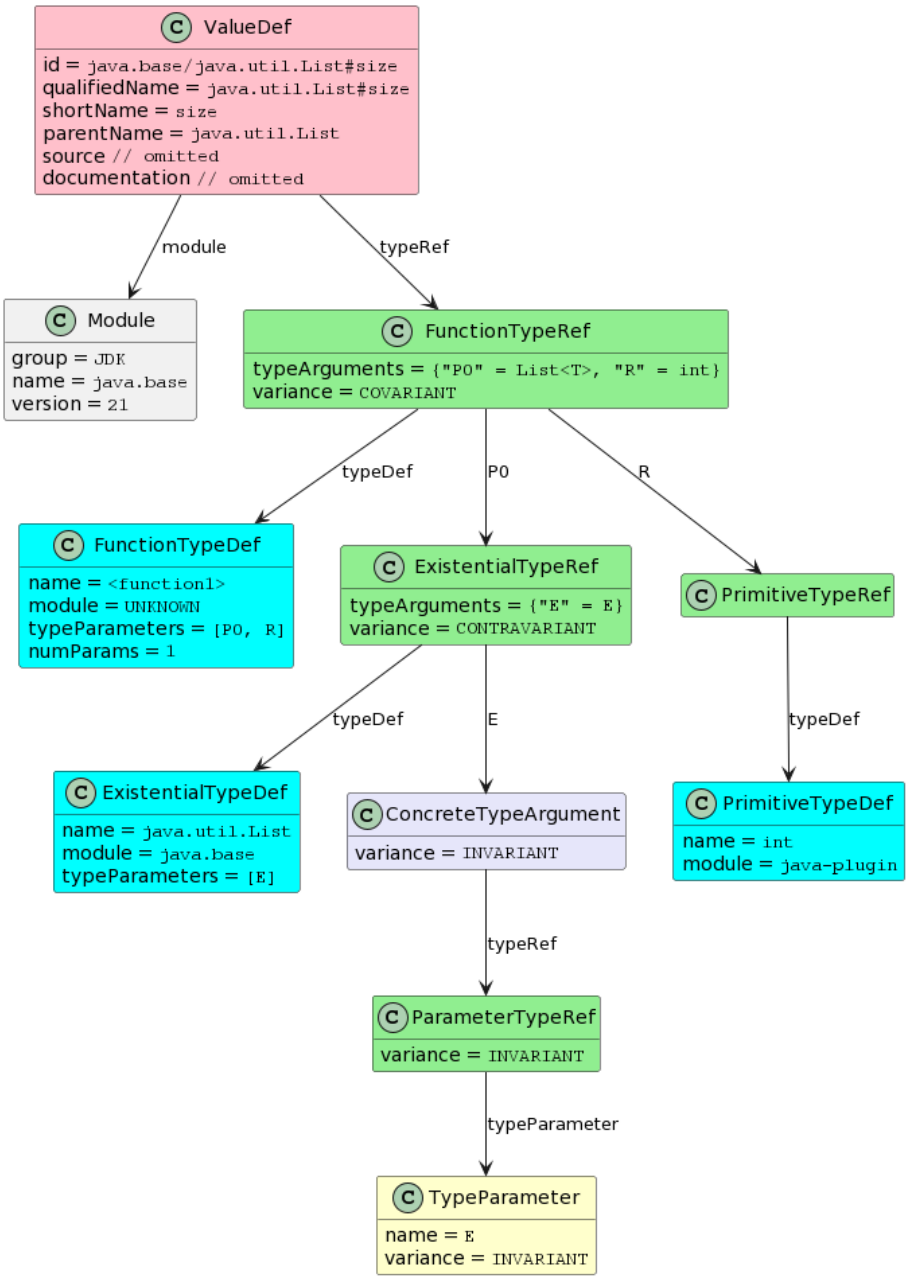

Figure 2.7: Example of a ValueDef data structure

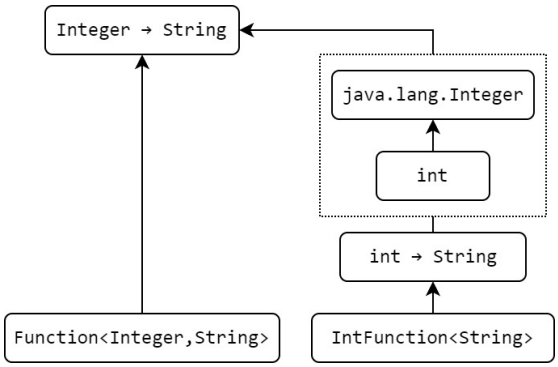

Figure 2.8: Example of type comparison using type mappings

use of this interface, so no assumptions about the underlying implementation are necessary.

2.7.1 Nominal Subtyping

Nominal subtyping is a system where every type explicitly declares for which types it is a subtype [Pie02, §19.3]. This is the most common kind of subtyping in object-oriented programming languages. Nominal subtyping information is most naturally represented by a directed acyclic graph. Every type encountered during indexing is added to the subtyping graph as a vertex. Directed edges run from a subtype to a direct supertype. Following edges from any vertex will always end at $\top$, representing a fictional type that is the supertype of all known types. *TypeSearch* uses the JGraphT[1] library to implement the nominal subtyping graph.

Using this representation, calculating the subtype distance between two types corresponds to calculating the shortest path in the graph between the two respective vertices. If two types are unrelated (i.e. there exists no subtyping relation between them), no path exists along the (directed) edges connecting the two vertices. An example of a subtyping graph is shown in figure 2.9. In that figure Java classes are green, interfaces blue, and the artificial $\top$ type is red. The red arrows show the shortest path from `ArrayList` to `SequencedCollection`. This shortest path traverses two edges, resulting in a subtype distance of 2. This subtype distance can be used when ranking search results, for example subtypings with a shorter subtype distance might be ranked more favorably.

2.7.2 Structural Subtyping

Another common subtyping scheme is structural subtyping. In structural subtyping, a type is considered to be a subtype of another type, if it contains all fields of the supertype, and each of those fields are a subtype of the supertype's field with the same name [Pie02, §19.3]. TypeScript is an example of a language that makes use of structural subtyping. *TypeSearch* does not yet provide a `SubtypeRegistry` implementation for handling structural subtyping, nor does it provide a language plugin for any structurally subtyped language. However, due to the `SubtypeRegistry` already being

[1] `https://jgrapht.org/`

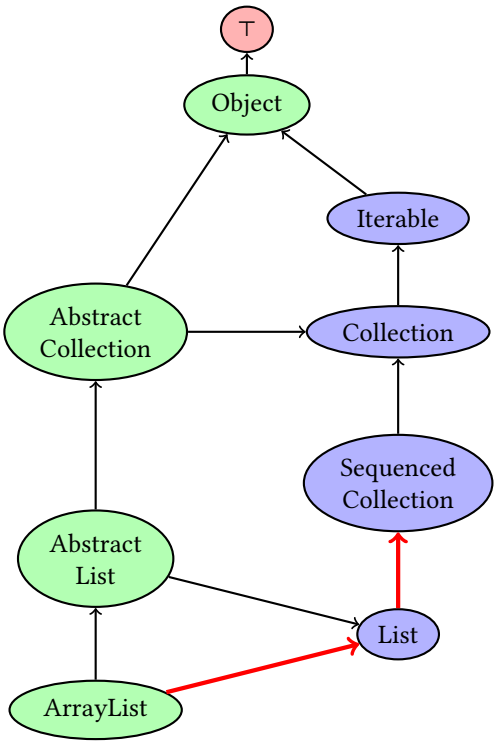

Figure 2.9: Example of a Subtyping Graph with subtype distance

implementation-independent, adding such an implementation should be possible without needing to adjust any of the search code.

3 Mining API Contents

API mining refers to the process of scanning the source code or an artifact of a particular programming language and extracting the type information into the Abstract Language Data Model. Since this process is highly language-specific, a plugin-based approach was chosen. *TypeSearch* offers a plugin API that can be implemented to provide extraction functionality for a programming language. *TypeSearch* ships with a plugin implementation for Java.

3.1 Plugin API

The `plugin-api` module of *TypeSearch* offers interfaces to be implemented by a language plugin, as well as data structures common to all language models. A language plugin must only provide an implementation of the `LanguagePluginDescriptor` interface, and one or more implementations of the `DefinitionSource` interface. The language plugin descriptor is used to retrieve metadata about the plugin, such as the plugin name, author, version, and programming language supported.

A `DefinitionSource` is an implementation that can mine API contents, optionally filtered by modules. Depending on the programming language ecosystem, there might be multiple sources of libraries which each require a distinct implementation to fetch and possibly parse. For example, the Java plugin, as described below, has two such definition sources: one for the JDK and one for Maven artifacts.

For a plugin to be picked up by *TypeSearch*, it is sufficient to register the language plugin descriptor in the Spring context. This is accomplished most easily by using SpringBoot's autoconfiguration mechanism. For developers interested in adding a new language plugin, we recommend looking at the Java plugin implementation as a reference.

3.2 Java Plugin

Figure 3.1: Java type construct mappings into the ALDM

Java	TypeDef	Notes
Class	Product	
Interface	Protocol	
Primitive	Primitive	
Enum	Sum	
Record	Product	
Constructor	Function	name is set to `<init>`
Method (member)	Function	`this` added as first parameter
Method (static)	Function	
Array	Product	`E[]` is represented as `Array<E>`
Multiple Type Bounds	Alias	for example `A & B`

The language plugin for Java adds the capability to extract type information from the Java Development Kit (JDK), as well as libraries hosted on Maven Central[1]. The

[1] `https://central.sonatype.com`

Java plugin makes heavy use of the javaparser[2] library for parsing and analyzing Java source code. Table 3.1 lists how various Java type constructs are mapped into the ALDM. Figure 3.2 shows a simplified example of this. The Java language plugin currently supports extracting from a JDK up to version 21. However, the javaparser library does not yet offer support for certain newer language constructs, such as records[3] and sealed classes[4]. Source files containing such constructs are unfortunately skipped for now.

Libraries often contain functions and types only meant for internal use within the libraries. The Java plugin filters API functions and types so that only those functions and types are extracted which are actually relevant. In general, a type or function is considered relevant, if it is possible to use that type or function within custom code that depends on the library. This is the case if the type or function is declared as `public` or `protected`. In the case of `protected`, an additional check is done to ensure that the declaring type is not `final`, meaning a developer could create a custom subtype and then gain access to the `protected` member.

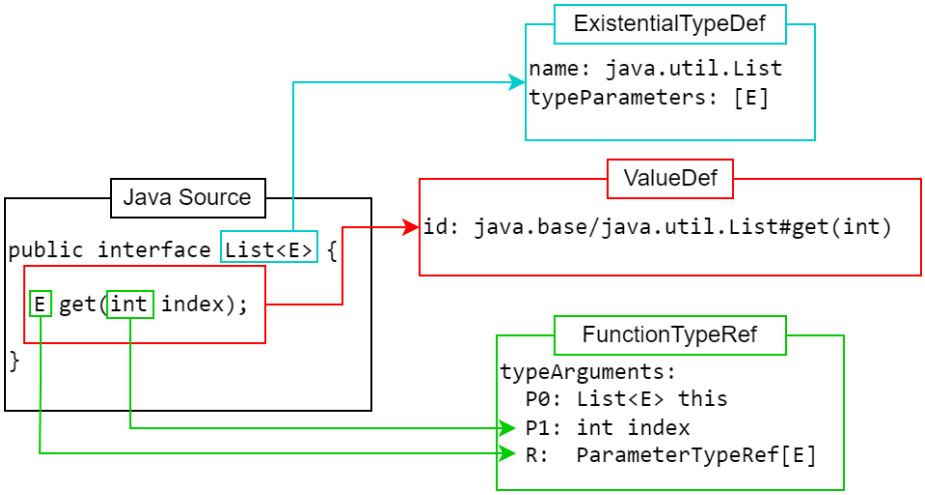

Figure 3.2: Java language extraction example

3.2.1 JDK Extraction

Most JDK implementations include Java source files in their distribution. These source files are usually used by IDEs to show JavaDoc or the actual source code. They are found in a `src.zip` archive in the JDK's installation directory. *TypeSearch* extracts this ZIP-archive and indexes the JDK by parsing the source files within.

Although reflection could be used to extract most of the information from the JDK, Java's reflection system does not have access to some helpful information. For example, the reflection system does not retain comments or method parameter names. Since both of these pieces of information are highly valuable for a user in deciding whether a particular function is what they are looking for, we opted to use the source-parsing approach instead.

[2] https://github.com/javaparser/javaparser
[3] https://github.com/javaparser/javaparser/issues/2446
[4] https://github.com/javaparser/javaparser/issues/2888

3.2.2 Maven Central Extraction

While indexing the JDK provides a solid basis, very few modern Java applications are based solely on the JDK's functionality. The most popular dependency management tools for Java projects are Maven[5] and Gradle[6]. Both of these tools use the same dependency resolution method under the hood: They download dependencies (by default) from the so-called "Maven Central" repository. Maven Central can store various artifacts, but most notably JAR files. However, JAR files which are used as dependencies in an application usually contain compiled `.class` files. Similarly to using reflection, such compiled class files have omitted useful information.

Fortunately, most popular libraries also offer an alternative JAR alongside, a so-called "sources JAR". This JAR file contains all the source code of the library, intended to be consumed by e.g. IDEs and other tooling. This is exactly what *TypeSearch* makes use of as well. *TypeSearch* downloads both the regular JAR and the sources JAR of libraries. It then uses the sources JAR to parse all the source files within and adds them to the ALDM. The compiled class-JARs are used by the javaparser tool for symbol resolution during parsing.

Which Maven artifacts *TypeSearch* will download and extract on application startup is configured through Spring application properties. *TypeSearch* does not yet have the ability to resolve transitive dependencies by itself, so such dependencies must be listed explicitly for the time being. See appendix B for a list of all configuration properties for *TypeSearch*.

3.2.3 Java Type Mappings

Type mappings are highly language-specific and must thus be provided by the corresponding language plugin. For Java there exist two kinds of type mappings: boxed types and functional interfaces.

Boxed Types

Java has so-called boxed types (also known as wrapper types) [Gos+23, §5.1.7]. These are fully-fledged classes, each representing a primitive type of the Java language. In most cases, the Java syntax allows for so-called auto-boxing (or auto-unboxing). This means the primitive type and the corresponding reference type are automatically converted into one another for the programmer's convenience. The list of primitive types in Java is defined by the Java Language Specification [Gos+23, §4.2] to be `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. All these boxed types do not have any type parameters. As such, the type mapping between primitive types and boxed types is a trivial one-to-one mapping. The type mappings for boxed types is biased towards the reference type, so the reference type is the type that will be used in the candidate index (see section 6.2.1).

Functional Interfaces

Java 8 introduced lambda expressions together with functional interfaces [Gos+23, §9.8]. A functional interface is a regular Java interface (with specific limitations) representing the type of a lambda expression. *TSaPS*'s function type syntax runs into

[5] `https://maven.apache.org/`
[6] `https://gradle.org/`

the problem that the mapping from a function type expression to a functional interface is ambiguous. For example, the query type `A -> A` could be resolved to `Function<A,A>` or `UnaryOperator<A>`. To solve this, all functional interfaces are mapped to the *TSaPS* syntax representation. This means that two functional interfaces are considered equivalent if they have a compatible type signature, even if the actual interfaces are unrelated in terms of subtyping.

4 Specifying Search Queries

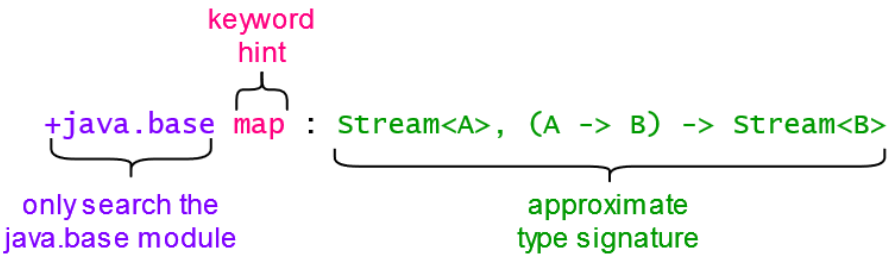

Figure 4.1: Example of a *TypeSearch* query

TypeSearch implements a single language-agnostic query syntax. This is possible thanks to the pre-processing step of mapping a language's type information into the uniform representation of the ALDM. Furthermore, using a single, consistent query syntax removes the required context-switching when changing which language ecosystem a user wants to search through. For example, the same query issued for Java can be used to search through TypeScript libraries.

Designing a language-agnostic query syntax comes with its own challenges. On the one hand, we aim to make the query syntax as intuitive as possible, such that the barrier of entry is as low as possible. On the other hand, a general query syntax is bound to have elements that are unusual for the particular programming language being queried. For example, Java uses a "dash-arrow" (`->`) to denote lambdas, whereas TypeScript uses an "equals-arrow" (`=>`). This particular example was solved by treating both kinds of arrows synonymously. However, there are other areas where we decided in favor of a stricter, simpler syntax instead. In an effort to make *TypeSearch* more accessible to developers used to OOP syntax, we designed the query syntax with common OOP syntax constructs in mind. For example, we decided to use a tuple syntax for function parameters similar to Java lambda expressions, as opposed to arrow syntax that is more common in functional programming languages.

The query syntax is made up of three parts: the module filters, keyword hints, and an approximate type signature. Module filters are optional, but inclusions and exclusions may not be mixed. Keyword hints and approximate type signature may be combined, but at least one of the two is required. Keyword hints are used in a simple text-search manner. The approximate type signature is the main focus of *TypeSearch* and the basis of the type-directed API search approach. Details regarding the type-directed search concept, and our implementation of it, are described in chapters 5 and 6. The formal EBNF specification of the query syntax can be found in appendix A.

4.1 Approximate Type Signature

The approximate type signature is the most important part of the query, at least in terms of type-directed API search. This is where a user specifies the type signature of the function they are looking for. This type signature need only be approximate, as the search implementation attempts to also find results that are close matches.

An approximate type signature consists of a comma-separated list of parameter

types, an arrow, and a return type. If a type is itself a function type, the entire type must be enclosed with parentheses, as shown in figure 4.1.

4.1.1 Type Resolution

When parsing a type query, all type names are resolved into known types. Type names can be fully-qualified (e.g. `java.util.List`) or use the simple type name (e.g `List`). Ambiguous simple type names will lead to an error when parsing the query. Certain namespaces are treated as "favorites" and will take precedence automatically in case of type name ambiguities. Which namespaces these are, is language-specific. *TypeSearch* treats the packages `java.lang.*` and `java.util.*` as favorites for Java. This means that `List` will resolve to `java.util.List` instead of `java.awt.List`, which would need to be qualified explicitly.

4.1.2 Universal Quantifiers

Single-letter type names are considered to be (unbounded) universally quantified type variables. Multiple occurrences of the same type variable in the query restrict search results to functions that also use a single type variable in each of those locations. For example, the query `A -> A` would not be matched by `String#toUppercase` (type signature String → String), but matched by `Function#identity` (type signature $\forall T.\ T \to T$). There is currently no way to specify bounds on a query type variable. In such a case it is often simpler to directly use the intended bound as the type of that parameter. So instead of the (hypothetical) query `A <: Number -> A`, the query `Number -> Number` would likely produce similar results.

4.1.3 Wildcards

Any unparameterized type name can be replaced with a wildcard character (`_`, `*`, or `?`). A wildcard character represents the concept of "any type is allowed here". All wildcard characters can be used interchangeably. Wildcard types are notably different from single-letter type names. Single-letter type variables can only be matched by another type variable, while wildcard types can be matched by any type, including type variables and concrete types.

The example query `_ -> String` would match all following function signatures:

- String → String
- int → String
- $\forall T.\ T \to$ String
- $\forall T <:$ Number. List $T \to$ String

4.1.4 Type Parameters

Type parameters are supplied with angular brackets and separated by commas. Type parameters are optional. Not supplying any type parameters is considered equivalent to supplying a wildcard expression for every type parameter of the type. E.g. `Map` is equivalent to `Map<?,?>`. If type parameters are specified, a value for every type parameter must be supplied.

4.2 Module Filter

In *TypeSearch*, all indexed library functions belong to a module. A module describes the location of a function. Although the exact location-description of a function might differ from language to language, modules in *TypeSearch* are designed to hide this detail. Module filters can either include or exclude one or more modules. If inclusions are specified, only functions from included modules are shown in the results. If exclusions are specified, no functions from the excluded modules are shown. Inclusion and exclusion filters are mutually exclusive. If no module filter is specified, all known functions are included in the search.

A module consists of three components: a group, a name, and a version. How these values are determined, depends on the particular language and might even differ between different extraction sources for the same language. The filter expression syntax is inspired by Maven artifact specifiers: `<group>:<name>:<version>`, where the group and version are optional. Filter expressions have a '+' prefix for inclusion, or '-' prefix for exclusion. While a query may contain multiple module filters, all of them must be placed at either the beginning or the end of the query.

5 Type Search as Proof Search

The *TSaPS* approach is based on the Curry-Howard correspondence [Pie02, §9.4] between the world of programs and the world of proofs. Using the Curry-Howard correspondence, the problem of type-directed API search can be translated into a problem of proof search. Once a valid proof has been found, that proof can be translated back into a search result of an API function. The Curry-Howard correspondence can even give us synthesized code snippets showing how to use the function or function composition, as described in section 6.8.

We first give a brief introduction to aspects of the Curry-Howard correspondence most relevant to our approach. Then, we introduce the constructs we need to build valid proofs. Chapter 6 deals with the implementation of these concepts and how *TypeSearch* efficiently searches for such valid proofs.

5.1 Curry-Howard Correspondence

The Curry-Howard correspondence is integral to understanding *TSaPS*. We begin by highlighting the key concepts of the correspondence. Please refer to Types & Programming Languages by Pierce [Pie02] for a more thorough description of the Curry-Howard correspondence.

The key insight of the Curry-Howard correspondence is that programs (or functions / terms) can be viewed as proofs. The types of those programs can be viewed as the logical propositions proven by said proof. This correspondence is bidirectional and is preserved through transformations on either side. Such a correspondence allows taking a problem in one domain and solving it in another domain, where the established techniques and frame of mind might be more suitable for the job.

The Curry-Howard correspondence is usually introduced using a form of the λ-calculus on the program side. *TSaPS* is also based on the λ-calculus. This means that functions collected from APIs must first be translated into the λ-calculus form before further processing by *TSaPS*. Section 5.3 goes into more detail on how this translation is accomplished.

To get an intuition for how we use the Curry-Howard correspondence, let's consider the following example: We are looking for a function with type signature `String -> Int`. There exists an API function `parse : CharSequence -> Int`, which can be viewed as a hypothesis `CharSequence -> Int` with the label `parse`. Furthermore, we have the subtype relation `CharSequence <: String`. We now want to build a proof, where the hypothesis `parse` is used to prove our query statement `String -> Int`. Finding such a proof corresponds to `parse` being returned as a search result of *TSaPS*. The following sections introduce the necessary syntax and proof rules to formally build this proof. A full proof of an extended example is shown in figure 5.1.

5.2 Proof Syntax

The syntax presented here is heavily inspired by the syntax of Pierce's treatment of λ-calculus with higher-order bounded quantification ($F^{\omega}_{<:}$) [Pie02, §31]. The main omission in *TSaPS*'s syntax is the lack of syntax constructs for values, and kindedness of types. Both of these are not required by *TSaPS*, since we can assume that APIs only contain type correct functions.

5.2.1 Types / Propositions

$\langle type \rangle ::= \langle universaltype \rangle$
$\quad | \quad \langle monotype \rangle$

$\langle universaltype \rangle ::= \forall \langle typevar \rangle <: \langle monotype \rangle . \langle type \rangle$

$\langle monotype \rangle ::= \top$ (maximum type)
$\quad | \quad \langle typeid \rangle$ (concrete type)
$\quad | \quad \langle typevar \rangle$ (type variable)
$\quad | \quad \langle monotype \rangle \rightarrow \langle monotype \rangle$ (function type)
$\quad | \quad \langle monotype \rangle \ \langle monotype \rangle$ (type construction)

$\langle typeid \rangle$ refers to the name or identifier of a type, which must be unique. This usually refers to a fully-qualified name in a programming language. $\langle typevar \rangle$ is a type variable introduced by a quantified type. A type variable may only occur within a quantified type that introduces the type variable. Type variables are usually denoted by a single uppercase letter.

Universally quantified types are defined separately to enforce that they may only occur on the outermost layers of a type. Additionally, if a quantified type has a bound of $<: \top$, the bound is omitted for brevity. Table 5.1 shows examples for the various forms a type can take.

Table 5.1: Type examples

$\langle type \rangle$	**Example**
concrete type	String
type variable	X
function type	String $\rightarrow$ Int
type construction	Optional Int
universal type	$\forall X <:$ Enum. EnumSet X

5.2.2 Terms / Proofs

$\langle term \rangle ::= \langle termid \rangle$ (term name)
$\quad | \quad \langle term \rangle \ \langle term \rangle$ (term application)
$\quad | \quad \langle term \rangle \ \langle type \rangle$ (type application)

Only the signature of an API function is translated. The actual function body remains abstracted and is referenced by a $\langle termid \rangle$. Additionally, there are term and type applications, where the first term is a function and the input argument is either a term or a type, depending on the function term. Table 5.2 shows examples for the different forms a term can take.

Table 5.2: Term examples

$\langle term \rangle$	**Example**
term name	*toString*
term application	*toString s*
type application	*newEnumSet* Month

Notably, function terms themselves are missing from the term syntax. The way *TSaPS* translates API functions and the input query into proof sequents, such terms never arise (or rather are not necessary). To more easily differentiate between type-level and term-level notation, the convention will be used that type identifiers begin with uppercase letters, and term identifiers begin with lowercase letters.

5.2.3 Type Environment / Proof Context

$$\Gamma = \{\gamma \mid \gamma = \langle term \rangle : \langle type \rangle\} \cup \{\delta \mid \delta = \langle type \rangle <: \langle type \rangle\}$$

The type environment (or proof context) is a set consisting of all hypotheses that may be used during a proof. A hypothesis is a tuple of a term and an associated type, denoted as $\langle term \rangle : \langle type \rangle$.

Additionally, the type environment contains all the subtype relations as tuples in the form "SubType <: SuperType". Subtype relations are transitive and impose a partial order on all types in the type environment. All types are a subtype of the top type $\top$.

During type search, the proof context contains a hypothesis for every API function that should be considered, as well as additional hypotheses that are introduced by proof rules during proof search. In the example shown in figure 5.1, the contents of the proof context are shown explicitly for every step of the proof.

5.3 Term Translation

After mining API contents is complete, the information about the API contents is stored in the Abstract Language Data Model. For proof search, we require this information in λ-calculus form. This means we need to apply another translation from the ALDM into the proof syntax. API mining could of course directly extract the information in proof syntax form. Due to legacy reasons, the ALDM is still used as an intermediate data model. This also carries the benefit that other search implementations (for example the *Hoogle5* fingerprint approach) can be implemented in a modular fashion, similar to the language plugin model.

To translate a `ValueDef` from the ALDM into a term of the proof syntax, we begin by constructing a function term that has the same type signature as the `ValueDef`'s `TypeRef` (usually a `FunctionTypeRef`). Any `UserDefinedTypeRef`s are translated into type constructions. We then collect all type variable occurrences in that term and wrap the term with a universal quantifier for each type variable, adding bounds as required.

Type mappings (section 2.6) and subtyping information (section 2.7) can be used as is, without the need to be translated.

5.4 Proof Rules

The following sections describe the various proof rules that *TSaPS* uses.

5.4.1 Axioms

$$\frac{}{\Gamma, \{\, m : \mathrm{G} \,\} \vdash g : \mathrm{G}}\ \mathrm{hyp}_m$$

The hyp rule states that if a hypothesis m has the exact same type as the goal g, then the proof is finished. The hypothesis m that matches the desired goal is then included as a search result.

5.4.2 Function Simplification

$$\frac{\Gamma, \{ g_1 : \mathsf{G}_1 \} \vdash g\ g_1 : \mathsf{G}_2}{\Gamma \vdash g : \mathsf{G}_1 \to \mathsf{G}_2} \to^{\text{goal}}_{g_1}$$

The $\to^{\text{goal}}$ rule allows simplifying goals with a function type. Since the goal term g has a functional type, we know that g must be of the form $\lambda g_1 : \mathsf{G}_1.\ g\ g_1$. By applying $\to^{\text{goal}}$, we transfer this input parameter g_1 from the goal into the set of hypotheses, and thus eliminate one $\to$ from the goal type. This is repeated until the goal only consists of the return type. The new hypothesis created by $\to^{\text{goal}}$ is assigned a fresh name, i.e. a term identifier that does not appear in Γ.

5.4.3 Bounded Universal Quantification

$$\frac{\Gamma, \left\{ \begin{array}{l} m : \forall X <: \mathsf{T}.\ \mathsf{M}\ X \\ m\ \mathsf{S} : \mathsf{M}\ \mathsf{S} \end{array} \right\}, \{ \mathsf{S} <: \mathsf{T} \} \vdash g : \mathsf{G}}{\Gamma, \{ m : \forall X <: \mathsf{T}.\ \mathsf{M}\ X \}, \{ \mathsf{S} <: \mathsf{T} \} \vdash g : \mathsf{G}} \forall^{\text{hyp}}_{m\ \mathsf{S}}$$

This rule allows creating a new hypothesis m S from a universally quantified hypothesis m by instantiating the type variable X with a concrete type S. This instantiation is then added to the set of hypotheses. Such an instantiation is only permitted if S satisfies the bound of X, namely $\mathsf{S} <: \mathsf{T}$.

If the goal contains universally quantified types, we want to eliminate them to bring the goal closer to normal form (see section 6.1). The following rule accomplishes this by eliminating one universal quantifier at a time:

$$\frac{\Gamma, \{ \mathsf{F} <: \mathsf{T} \} \vdash g\ \mathsf{F} : \mathsf{G}\ \mathsf{F}}{\Gamma \vdash g : \forall X <: \mathsf{T}.\ \mathsf{G}\ X} \forall^{\text{goal}}_{X \mapsto \mathsf{F}}$$

Similar to $\to^{\text{goal}}$, we know that g must have the form $\lambda X <: \mathsf{T}.\ g\ X$ based on its type. Same as in $\forall^{\text{hyp}}$, the universal type variable X is instantiated with a concrete type. However, this time F is a fresh type, following the same idea of $\to^{\text{goal}}$, i.e. a type that does not exist in Γ. Additionally, this fresh type F is bounded by the same bound present on X.

5.4.4 Resolution

$$\frac{\Gamma, \left\{ \begin{array}{l} n : \mathsf{T} \\ m : \mathsf{T} \to \mathsf{M} \\ m\ n : \mathsf{M} \end{array} \right\} \vdash g : \mathsf{G}}{\Gamma, \left\{ \begin{array}{l} n : \mathsf{T} \\ m : \mathsf{T} \to \mathsf{M} \end{array} \right\} \vdash g : \mathsf{G}} \text{resolution}_{m\ n}$$

The resolution rule performs incremental simplification by proving one parameter of a function m at a time. This resolution is analogous to function application: applying resolution introduces a new hypothesis where the function m is applied to the term

n. Using the resolution rule allows complex hypotheses to be deconstructed into a composition of simpler hypotheses.

5.4.5 Subtyping

$$\frac{\Gamma, \left\{ \begin{array}{c} m : \mathsf{S} \\ m^{<:} : \mathsf{T} \end{array} \right\}, \{ \mathsf{S} <: \mathsf{T} \} \vdash g : \mathsf{G}}{\Gamma, \{ m : \mathsf{S} \}, \{ \mathsf{S} <: \mathsf{T} \} \vdash g : \mathsf{G}} \ \mathrm{sub}_{m,\mathsf{S}<:\mathsf{T}}$$

The final proof rule is a rule for dealing with subtyping. The rule states that the type of a term m can be replaced with any of its supertypes.

The question might arise, why subtyping is not simply handled through "upcast" functions, converting a subtype to a supertype. Assume we have the function $abs : \forall X <: \mathsf{Number}.\ X \to X$. Here, X may only be instantiated with a subtype of Number, for example Int. The knowledge that Int is a subtype of Number would only be encoded by the fact that an upcast function with type signature $\mathsf{Int} \to \mathsf{Number}$ exists. If we were to treat this upcast function the same as any other function, we would run into problems. Assume a function $parseNumber : \mathsf{String} \to \mathsf{Number}$ exists. The presence of this function suggests that we might be able to instantiate X in abs with String, which would be incorrect. Rather than carefully differentiating upcast functions from regular functions, we decided to keep these concepts separate in *TSaPS*.

5.5 Proof Example

Let's take a look at a complete example of a proof, shown in figure 5.1. Such a proof construction is best understood by reading the proof tree from bottom-to-top.

We start by creating the proof context by adding all known API functions and subtype relations. For this example we limit ourselves to only the single API function *get* and the subtype relation $\forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X$. On the right-hand side we add the query expression `ArrayList X, Int -> X`, translated into λ-syntax.

The first three proof rules perform goal normalization (see section 6.1). First, the type variable X is instantiated with the fresh type F. Then, both input parameters (g_1 and g_2) of g are transfered into the proof context with the $\to^{\mathrm{goal}}$ rule. The goal is now in normal form and has type F.

Looking through all hypotheses in the proof context, we find that *get* is a goal candidate and apply transforming proof rules to arrive at an exact match. See chapter 6 for a detailed description of how we determine which proof rule to apply when. The next rule we need to apply is $\forall^{\mathrm{hyp}}$ to instantiate X on *get* with F. This yields the new hypothesis *get* F.

The new hypothesis is now a function with two input parameters. For each one of these we want to apply the resolution rule. Before we can do so, however, we need to remediate the type mismatch between the first parameter of *get* F of type List F and g_2 of type ArrayList F. Fortunately, we have a subtyping relation for exactly this. We apply the sub rule to obtain the hypothesis $g_1^{<:}$ with type List F.

Now we can apply the resolution rule once for each parameter, ultimately arriving at the hypothesis *get* F $g_1^{<:}\ g_2$ of type F. Having obtained a hypothesis with the exact same type as the goal type, we have found a valid proof and top off the proof with the hyp rule.

$$\frac{}{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ \vdots \\ get\ \mathsf{F}\ g_1^{<:} : \mathsf{Int} \to \mathsf{F} \\ get\ \mathsf{F}\ g_1^{<:}\ g_2 : \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}\ \mathrm{hyp}_{get\ \mathsf{F}\ g_1^{<:}\ g_2}$$

$$\frac{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ get\ \mathsf{F} : \mathsf{List}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F} \\ g_1^{<:} : \mathsf{List}\ \mathsf{F} \\ get\ \mathsf{F}\ g_1^{<:} : \mathsf{Int} \to \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ get\ \mathsf{F} : \mathsf{List}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F} \\ g_1^{<:} : \mathsf{List}\ \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}\ \mathrm{resolution}_{get\ \mathsf{F}\ g_1^{<:}\ g_2}$$

$$\frac{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ get\ \mathsf{F} : \mathsf{List}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F} \\ g_1^{<:} : \mathsf{List}\ \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ get\ \mathsf{F} : \mathsf{List}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}\ \mathrm{resolution}_{get\ \mathsf{F}\ g_1^{<:}}$$

$$\frac{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ get\ \mathsf{F} : \mathsf{List}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ get\ \mathsf{F} : \mathsf{List}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}\ \mathrm{sub}_{g_1,\mathsf{ArrayList}<:\mathsf{List}}$$

$$\frac{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ get\ \mathsf{F} : \mathsf{List}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}\ \forall^{\mathrm{hyp}}_{get\ \mathsf{F}}$$

$$\frac{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ g_2 : \mathsf{Int} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1\ g_2 : \mathsf{F}}{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1 : \mathsf{Int} \to \mathsf{F}}\ \to^{\mathrm{goal}}_{g_2}$$

$$\frac{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ g_1 : \mathsf{ArrayList}\ \mathsf{F} \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F}\ g_1 : \mathsf{Int} \to \mathsf{F}}{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F} : \mathsf{ArrayList}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F}}\ \to^{\mathrm{goal}}_{g_1}$$

$$\frac{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g\ \mathsf{F} : \mathsf{ArrayList}\ \mathsf{F} \to \mathsf{Int} \to \mathsf{F}}{\left\{\begin{array}{r} get : \forall X.\ \mathsf{List}\ X \to \mathsf{Int} \to X \\ \\ \forall X.\ \mathsf{ArrayList}\ X <: \mathsf{List}\ X \end{array}\right\} \vdash g : \forall X.\ \mathsf{ArrayList}\ X \to \mathsf{Int} \to X}\ \forall^{\mathrm{goal}}_{X \mapsto \mathsf{F}}$$

Figure 5.1: Proof tree of "$\forall X.\ \mathsf{ArrayList}\ X \to \mathsf{Int} \to X$" with subtyping

6 Searching for Proofs

In this chapter we discuss how *TypeSearch* implements search for valid proofs within *TSaPS*. Figure 6.1 shows an overview of the entire search procedure. It assumes that any query parsing and translation into the *TSaPS* data model has already been done.

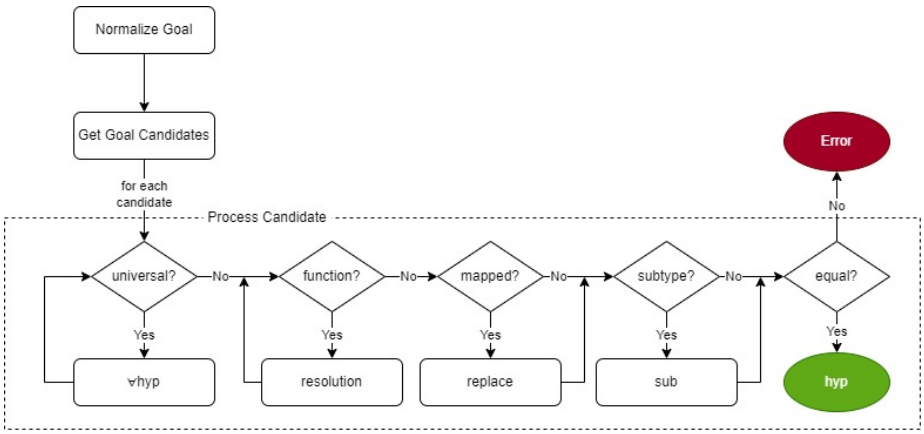

Figure 6.1: Search method procedure

This proof search rarely leads to a single valid proof. Usually, there are multiple possible choices for any proof step. Thus, the output of this search procedure is a proof search tree, where every leaf corresponds to a valid proof. Branches represent different choices to get to a valid proof.

6.1 Goal Normalization

Before we can start searching for proofs, it is convenient to transform the goal (i.e. the "query") into a normal form. A goal is considered to be in normal form, if (1) the goal contains no type parameters, and (2) the goal contains no arrows.

To achieve this, we first apply the proof rule $\forall^{\text{goal}}$ for every type parameter the goal has. Once the goal no longer contains any type parameters, we apply $\rightarrow^{\text{goal}}$ until the goal no longer has any arrows. The goal is now in normal form. The restriction that universally quantified types may only occur on the outermost layers guarantees that this order of operations is sufficient to arrive at the normal form. Furthermore, this goal normalization automatically uncurries any functional return types in the query. For example, the queries `A, B -> (C -> D)` and `A, B, C -> D` are treated identically.

6.2 Goal Candidates

After goal normalization has completed, the goal term now has a single concrete type. We call this type the "goal type". The proof search in general is performed "backwards", starting with this goal type. To start, we collect all goal candidates — terms (proof hypotheses) that could eventually return a type compatible with our goal type. This includes terms that return:

- the same type as the goal type

- a subtype of the goal type
- a type equal to the goal type after applying a type mapping (see section 2.6)
- a subtype of the goal type after applying a type mapping
- a type parameter where the goal type is a subtype of the parameter's bound
- a type parameter where the parameter's bound is a subtype of the goal type

6.2.1 Candidate Index

To efficiently retrieve these goal candidates, a reverse index is built ahead of time during application start up. This data structure indexes every hypothesis by its return type, and all of its supertypes. The types used in the index are always types after applying any type mappings. For parameterized types, only the "raw" type is considered for the index, so the type without any applied type arguments. If the return type of a hypothesis is a type variable, the hypothesis is indexed by the type variable's bound, all subtypes of the bound, and all supertypes of the bound. If a type variable is unbounded ($\forall X <: \top$), it is added to a list of unbounded universals rather than added to the list of every possible type. This means we can fetch all goal candidates with only two lookups in the index: one for the goal type and one to get all unbounded return types. Figure 6.2 illustrates this concept.

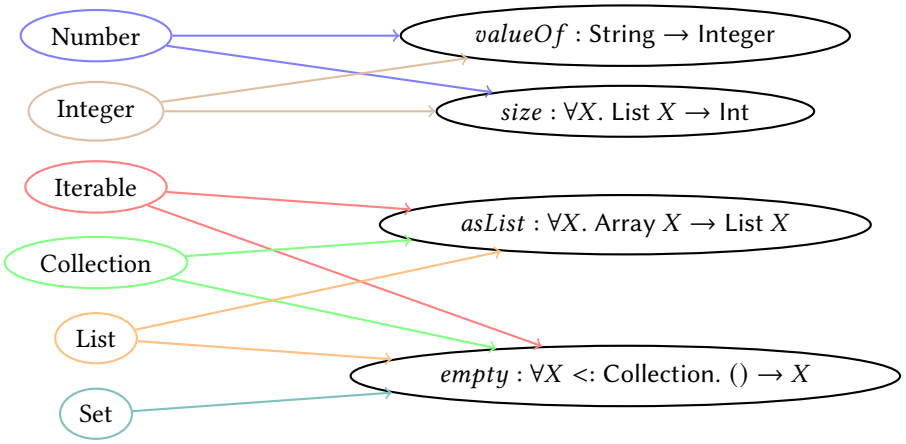

Figure 6.2: Goal Candidate Index

The reverse index uses an over-approximation of the return type. This means that the returned hypotheses are a superset of the actual goal candidates. To narrow down these "index candidates" to the real goal candidates, every returned hypothesis must now be compared in more detail against the goal type of the query. This comparison mainly includes checking for the precise type arguments, which have been omitted from the reverse index. For example, a goal type of List String will find the index candidate with return type List Integer, which is not actually compatible. Thus, this index candidate will be discarded.

6.3 Universals

Once all goal candidates have been determined, we continue to process each candidate individually down the chain depicted in figure 6.1. As the first step, we check whether the hypothesis has any type parameters. If so, we look for an instantiation.

If the candidate's return type is or contains a type parameter, the instantiation is found by instantiating the type parameter so that the return type equals (or is a subtype of) the goal type. This is similar to unification, but considers subtyping as well.

Finding an instantiation is more complicated if the type parameter does not occur in the candidate's return type. Due to function composition, such type parameters could potentially be instantiated with any possible type (within the type parameter's bounds). At this point, *TypeSearch* implements a heuristic to limit the search space for possible instantiations: only types mentioned in the query are considered. Then, an exhaustive search is performed by instantiating the type parameter with all types mentioned in the query. This limitation does not exclude any valid proofs that do not use function composition.

6.4 Functions

This step deals with resolving function parameters and can be considered the most central part of the proof search. Resolution happens one function parameter at a time and then repeats. However, instead of fetching goal candidates, we fetch resolution candidates based on the type of the parameter being resolved. Each resolution candidate is then recursively processed as if it were a goal candidate with the parameter type being treated as the goal type.

Such a recursive resolution quickly leads to exponential explosion of the search space. For this reason, *TypeSearch* has a configurable maximum search depth and maximum function composition depth. Appendix B shows all configuration parameters of *TypeSearch*.

If no resolution candidates can be found for a function parameter, the candidate is rejected. Otherwise, a resolution proof rule is recorded in the proof tree.

6.5 Mapped Types

If a candidate has made it successfully through resolution, we have found a valid proof. We merely need to (potentially) add some proof rules on top of the result to formally finish the proof. The first adjustment that needs to be done is to check for type mappings. For example, the fully resolved hypothesis might have the type "Function Int Bool" for the goal type "Predicate Int". They are related through the common representation "Int → Bool". To achieve equality between the type of the hypothesis and the goal, we need to convert the Function type to a Predicate. Note that there exists no type mapping for this direct relationship. Instead, we can obtain such a type mapping by composing the type mappings "Function → fn1" and the inverse of "Predicate → fn1", as shown in figure 6.3.

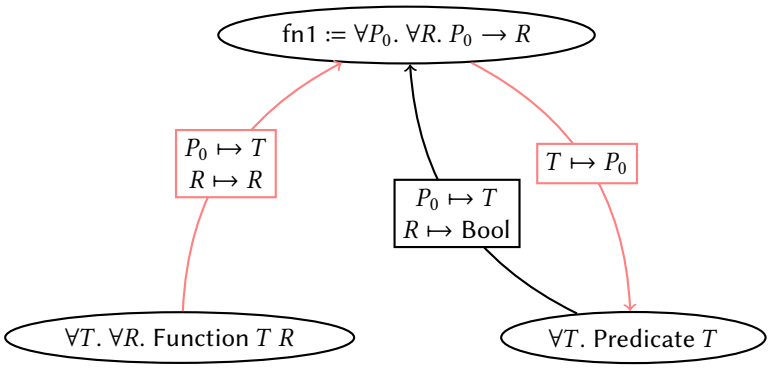

Figure 6.3: Type Mapping Composition

6.6 Subtyping

If the type of the (fully resolved) candidate is a subtype of the goal type, we need to apply a sub rule. This step should never lead to rejection of a candidate, as this subtype compatibility has already been checked during goal candidate retrieval. The application of the sub proof rule is merely a formality. However, the addition of this proof rule can be an important factor when ranking proofs amongst each other.

6.7 Proof Finalization

As the final step, the hyp rule is added to the proof, completing it. Now that all valid proofs have been found, the proofs are ranked according to a relevance metric, such that "better" proofs appear higher up in the search result list.

TypeSearch trivially sorts proofs by proof length, with shorter proofs being considered more relevant. This can be interpreted as shorter proofs needing fewer transformations of the corresponding term(s) to conform to the query. However, this runs into a different problem: the *TSaPS* proof grammar, and proof search in general, does not require that the hypotheses introduced from goal normalization are actually used in the proof. This means the result will include valid proofs using terms with fewer arguments than the query. From a proof-theoretical point of view this is absolutely correct. For our purposes, however, we desire that as many of the parameters specified in the query as possible are actually used in the proof. *TypeSearch* achieves this by applying a configurable penalty to all proofs based on how many query parameters remain unused in the proof. These two simple ranking rules already turn out to produce very useful relevance rankings as shown in chapter 7.

After proof ranking has completed, we are still left with an ordered list of proofs rather than actual search results that could be shown to the user. For the moment, *TypeSearch* simply uses the goal candidate of the proof and selects the corresponding `ValueDef` as the search result. This is a shortcut that does not work properly if a proof contains function composition — the composed functions would be lost. To correctly handle function composition, a more intelligent transformation from proof to search result would be required, such as code synthesis, described in the next section.

6.8 Code Synthesis

Whenever a proof rule is applied, any transformations apply equally to types and terms. This means not only does a valid proof give us an API function (term) that satisfies the given query, the proof also tells us exactly which parameter of that API function should be provided with which value.

The hyp rule references a single term that has the exact same type as the goal type. Often, this referenced term contains function applications in the form of e.g. *m n*. Such applications can be translated back into (pseudo-)code snippets. The result of the example in figure 5.1 of "$get\ \mathsf{F}\ g_1^{<:}\ g_2\ :\ \mathsf{F}$" can be translated back into the following compact psuedo code: `X g = get<X>(g1, g2)`. Here, `g1` and `g2` are the first and second parameters of the query, respectively, and `X` is the type variable of the query, which was instantiated to `F` during proof search. A more verbose example is given in listing 6.1. In the verbose example, the query is reflected in the signature of the function `g`. The implementation of `g` corresponds to the proof result.

```
1  forall X. X g(ArrayList<X> g1, Int g2) {
2      return get<X>(g1, g2);
3  }
```

Listing 6.1: Verbose synthesis for the proof in figure 5.1.

If a proof contains function composition, the synthesized code snippet would reflect this by containing additional function calls.

7 Evaluating TypeSearch

This chapter shows performance measurement results of *TypeSearch*, the proof of concept implementation of *TSaPS*. Due to the absence of a real user-base, performance measurements were done by selecting a random sample population of JDK 21 API functions. For each API function, a query is generated with a type signature equal to that of the function. Then, these queries are executed, and the desired metrics collected in order to give an initial estimate of the feasibility of using this approach in practice.

7.1 Speed

Table 7.1: Response time statistics

Mean	1 212 ms
Sample Std. Dev.	4 027 ms
50% (Median)	239 ms
90%	2 078 ms
99%	24 127 ms

Speed measurements are measured from the point in time a query is issued to the moment the list of search results is returned. Figure 7.1 shows a histogram of response times across a sample population of 1 000 against an index size of 40 000 API functions. The entirety of JDK 21 contains 41 208 "relevant" functions (see section 3.2). The histogram is limited to a range of 0 to 5 seconds response time for legibility. Any response times exceeding 5 seconds are included in the last bar, leading to the visible peak. Table 7.1 shows some key statistics about this distribution.

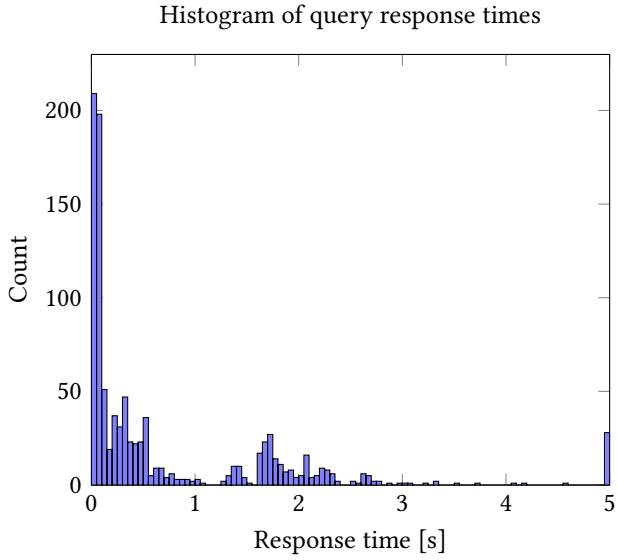

Figure 7.1: Response time histogram

Figure 7.1 shows the distribution of response times for queries against a fixed index of 40 000 API functions. To get a feeling of how well *TSaPS* and *TypeSearch* scale with index size, figure 7.2 shows the mean and median response times in relation to index size. Each of these means and medians were calculated from the same random sample population of 1 000 queries.

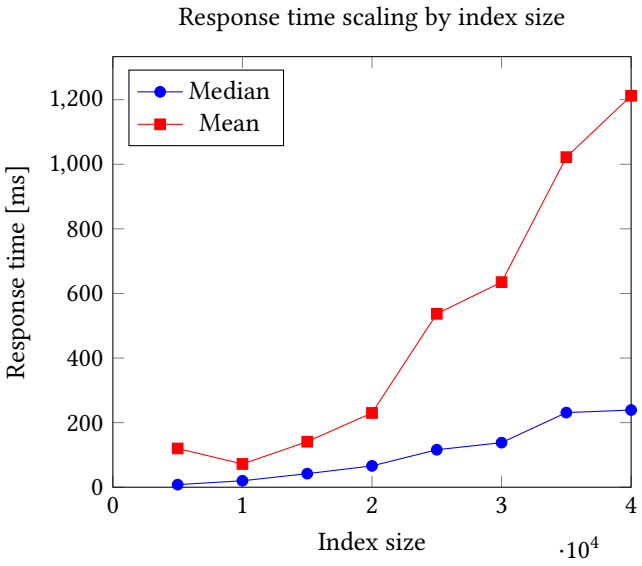

Figure 7.2: Response times by index size

7.2 Quality

The quality of search results is harder to measure meaningfully. We measured the quality of *TypeSearch* using the same approach of taking random samples and generating queries from them. We then record the position of the sample in the search result list. The results of this measurement are shown in figure 7.3.

The small increase in the histogram at the value 100 is due to the fact that we assigned a result position of 100 to every sample where the sample was not found within the first 100 search results.

7.3 Metric Collection

To eventually get real feedback about the usage of *TypeSearch* and the user's satisfactions with the results, *TypeSearch* collects certain metrics about the application's usage. Firstly, all issued queries are logged, together with the response time. Importantly, this also stores syntactically invalid queries. This gives us information about how users think a query should be formulated. We do not store any information about the client that issued the query for data privacy reasons.

Additionally, users have the possibility to give feedback about concrete search results. After issuing a query, every returned search result has buttons to "like" or "dislike" a search result. Figure 7.4 shows a screenshot of the *TypeSearch* user interface

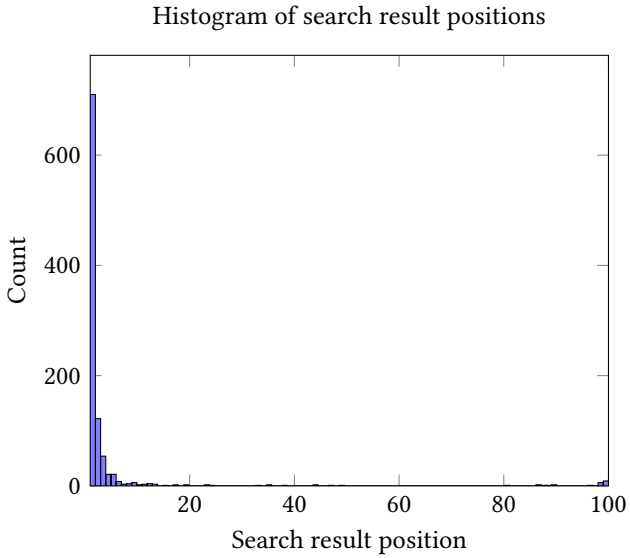

Figure 7.3: Quality histogram

with the feedback buttons on the right side. The user may then supply additional, optional written feedback. This feedback is stored together with the corresponding query. We do not yet have any significant amount of feedback, but we hope that this user-supplied feedback feature will allow us to optimize the rankings moving forward.

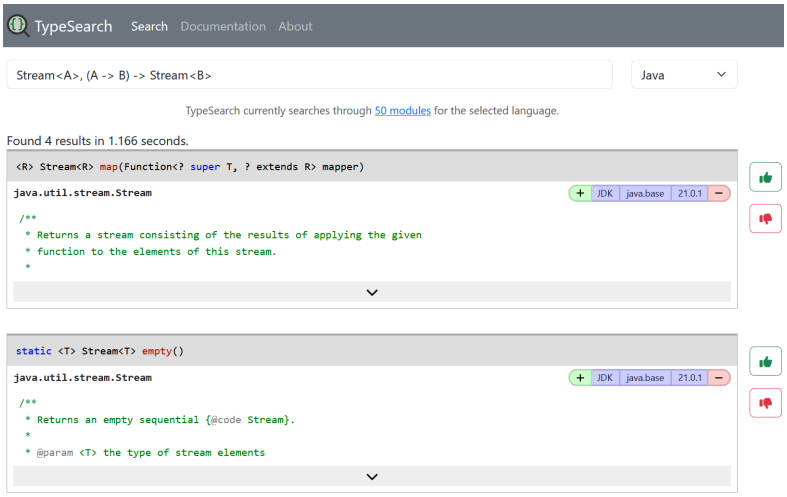

Figure 7.4: TypeSearch User Interface

8 Comparing Related Work

The idea of building specialized software to search for functions based on an approximate type signature is not new. Type-directed search has been used both as the objective and as an intermediate step for code synthesis in various approaches. This chapter goes over some related work in this field, with some notable mentions at the end.

8.1 Hoogle (2005)

Hoogle [Mit05][Mit08] for Haskell is likely one of the best known and most successful type-directed search engines in the world of typed functional programming languages. *Hoogle* being specifically a search engine for Haskell was also the main motivation behind this thesis: building a type-directed search engine with a language-agnostic core. *Hoogle* has undergone multiple iterations, most of them being a complete rewrite with a fundamentally different concept at its center. The newest iteration, *Hoogle 5*, is based on fingerprinting functions based on certain indicators, such as arity and type frequency [Mit20]. This same principle is implemented in a previous version of *TypeSearch*[1]. However, a desire for a more systematic approach with fewer heuristics led us to develop *TSaPS*.

Guo et al. published an improvement they called *Hoogle+* in 2020 [Guo+19][Jam+20]. Their main improvement over *Hoogle* is that *Hoogle+* can synthesize entire terms by composing multiple functions, similar to the jungloid mining approach. However, it pays for the added functionality with a significant performance loss and added complexity regarding the required user input. *Hoogle+* not only uses a type-signature query string, but also properties, as known from property-based testing techniques. These properties are evaluated during the search to discard undesired compositions, improving accuracy, but slowing down the process.

In 2023, Guerra et al. published *Hoogle★*. They improved upon *Hoogle+* by adding the ability to synthesize constants and λ-abstractions. Both *Hoogle+* and *Hoogle★* base their implementation on *TyGAR* [Guo+19]. *TyGAR* is a search method extending *SyPet*'s [Fen+17] Petri nets with the support for parametric polymorphism, but still without any support for subtyping. *SyPet*'s use of Petri nets is discussed in more detail in section 8.3.

8.2 Jungloid Mining (2005)

The tool *Prospector* created by Mandelin et al. was a code assistance feature for the Eclipse IDE released in 2005 [Man+05]. It was one of the first search engines for Java functions capable of returning composite results. It focused on synthesizing code fragments which transformed an input type into an output type, possibly by calling multiple intermediate methods. The authors named such a fragment a "jungloid". Especially for complex tasks (i.e. tasks requiring intermediate steps), *Prospector* performed respectably. The main limitation of their approach, however, is that "jungloids" can only transform a single input type into a desired output type. Oftentimes, a developer requires combining multiple values/types into a single type, which is why *TSaPS* allows for multiple input types.

[1] `https://legacy.typesearch.dev`

8.3 SyPet (2017)

SyPet [Fen+17] is a more recent contribution, offering type-directed search for Java. Feng et al.'s main insight is that modeling the search space as a Petri net instead of hypergraphs improves performance significantly. While *SyPet* claims to support parametric polymorphism, they do so by creating monomorphic instantiations of functions for every possible instantiation type. For a large search space with many types this quickly becomes unfeasible - especially for functions with multiple type parameters. Additionally, *SyPet* has no support for subtyping and specifically targets Java.

8.3.1 Petri Nets

Feng et al. use Petri nets to model types as resources that are "consumed" by functions when used as an input, and "produced" when returned by a function. This representation creates a structure similar to a hypergraph, but with certain advantages (and disadvantages) to a pure hypergraph. One of the main benefits of Petri nets listed by Feng et al. is that Petri nets can account for functions with side effects, while hypergraphs only really work for pure functions.

TypeSearch, as will be discussed in section 8.3.2, uses an approach that is very similar to hypergraphs. As the *SyPet* paper mentions, this allows us to directly obtain synthesized code snippets from a valid proof, rather than separating the approach into a sketch and synthesis phase, as *SyPet* does. We acknowledge that this representation does not support functions with side effects. However, we decided to prioritize speed and ease-of-use of our tool. For example, *TypeSearch* does not make use of test cases to verify synthesized code snippets. We believe it is more important for the user to get quick results. If a user must first write one or more test cases, this introduces a significant effort for the user before they get any search results. Furthermore, we believe that users are able to quickly judge the majority of synthesized code snippets on whether they fulfill their needs or whether they are nonsense.

We believe a search engine should inherently be designed to deal with inaccuracies, as the user by definition has incomplete information over what they need (otherwise they would not need the search engine in the first place). Therefore, certain restrictions of *SyPet*, such as every input parameter *must* be used in the synthesized code, are overly limiting. It is perfectly imaginable that a user might think that they need a certain parameter for the function they want, but the function actually does not need that parameter — or vice-versa.

8.3.2 Hypergraphs

For *TSaPS* and *TypeSearch* we decided to use proof search as the primary mechanism. Unsurprisingly, this approach can be translated into other representations. One such representation uses hypergraphs. In such a hypergraph, nodes correspond to types, and (hyper-)edges correspond to functions.

TSaPS and *TypeSearch* also consider subtyping relations and quantified types, which are not easily added to a hypergraph representation where every node corresponds to a concrete type. Support for these type constructs is done similarly to *TyGAR* [Guo+19], in that each node in the hypergraph analogy corresponds to an abstract type (e.g. a type variable with specific bounds, or a type including all of its subtypes), rather than a concrete type.

To model *TSaPS* in terms of hypergraphs, we proceed as follows: each time a query is executed, we insert two artificial nodes S and T into the hypergraph. Edges are then added from S to every node (type) that occurs in the query's parameter list. Similarly, an edge is inserted from the query's return type to T. Proof search now corresponds to finding a "complete" subgraph that contains the artificial node T. A subgraph is considered "complete" if every node other than S in that subgraph has either an incoming edge from a different node in the same subgraph, or at least one incoming edge without any source vertices. This condition corresponds to resolution having terminated, meaning there are no "left-over" input parameters to be substituted.

Starting at T, we traverse the hypergraph along one incoming edge at a time. After the first step, we arrive at Bool. In the example depicted, the next step selected the incoming edge "contains", which leads us to the String and Char nodes. If a different incoming edge had been selected (not shown in the example), we might find a different valid search result, or the search might not lead to a valid subgraph.

This procedure repeats, until there are no further incoming edges (or incoming edges have no source vertices) to follow. This usually means we arrived at S (no further incoming edges), or there exists a function that produces the required type without needing any inputs (an edge without any source vertices). In other words, the backwards traversal always either ends up at S or at a "dead end", an edge without source vertices. Traversal is also aborted, if the maximum search depth or maximum function composition depth has been reached.

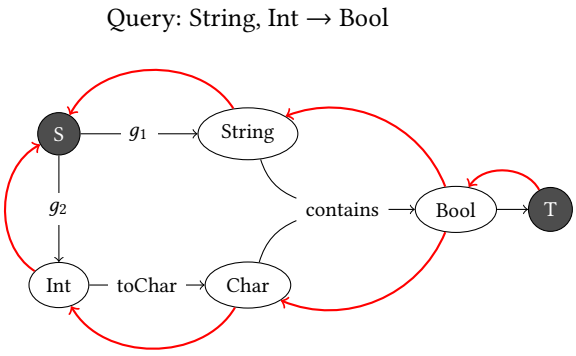

Figure 8.1: Hypergraph backwards traversal analogy

In the example shown in figure 8.1, the query `String, Int -> Bool` is depicted as backwards traversal of a hypergraph. The subgraph found through this backwards traversal can then be synthesized back into the search result `Bool g = contains(g1, toChar(g2))`.

8.4 Others

There exist multiple different type-directed search engines for various programming languages. *Scaps* [Weg+16] is a search engine for the Scala programming language and makes use of "variance fingerprinting". The approach is similar to the fingerprinting idea of *Hoogle 5*, but focuses on the variance polarity of parameters. *elm-search*[2]

[2] `https://klaftertief.github.io/elm-search/`

and *Moogle* [OL20] are search engines for Elm. The former is built on top of custom type-distance heuristics. The latter uses a unification algorithm and AST-based graph models. *Loogle*[3] is a tool for Lean and Mathlib.

[3] `https://loogle.lean-lang.org/`

9 Further Work

There are still many improvements to *TypeSearch* in its future: For one, code synthesis is not yet implemented, and function composition is not yet viable with the current eager enumeration of all possible proofs. An important performance improvement would be to refactor proof search to work lazily and prioritize the proof search based on which branches are most promising. In a similar vein, instantiation of type variables that do not occur in the return type of a function could be improved by deferring instantiation through the use of unification placeholder variables.

The ranking of search results works well in most cases, but could be improved. For example, some API functions use a type variable for the return type but that type variable does not occur anywhere in the parameter list. Such API functions are highly unlikely to be the desired search result and should be ranked lower, because such a function can return any type. One example of such a function is `org.junit.jupiter.api.Assertions#fail(String)` with the type signature $\forall T.\ \mathsf{String} \to T$. Given the query `String -> String`, a function $\forall T.\ T \to T$ should be considered a better match than the `fail` function. Currently, they would be assigned the same ranking.

While *TypeSearch* handles extraneous query parameters well (there is no hard requirement that all query parameters must be used in a proof), it cannot yet find proofs using parameters that were forgotten in the query. One approach to extend proof search with the ability to use missing hypotheses during resolution, is to add a $\forall X.\ X$ hypothesis to the index. Such a hypothesis will be able to resolve any parameter. Of course, this hypothesis must be assigned a large penalty, so that it is only used as a fallback if no other resolutions can be found.

Lastly, how well the language-agnostic core truly performs for additional programming languages will only be proven when one or more additional language plugins are added. Especially the addition of a plugin for a programming language with a structural type system, such as TypeScript, would be useful to validate the language-independent aspects of both proof search itself and the subtyping registry.

10 Conclusion

Type-directed API search engines are a useful tool for many programming languages. Various approaches have been explored and all of them have focused on a specific programming language. Recent work seems to prefer Petri nets for type-directed search and code synthesis. In this Master's Thesis we introduced *TSaPS*, a type-directed search approach based on the Curry-Howard correspondence and proof search. We demonstrated that such a proof-based search can also perform well, even for large indexes with tens of thousands of functions. *TSaPS* supports parametric polymorphism and subtype polymorphism as first-class citizens, which many existing tools do not.

TypeSearch successfully offers a proof-of-concept implementation of *TSaPS* that developers can use immediately. *TypeSearch* ships with a language plugin for Java. Due to being built on top of a programming language-independent core, extending *TypeSearch* with support for additional languages only requires developing a plugin for parsing the language.

All in all, we conclude that a type-directed API search and code synthesis approach based on the Curry-Howard correspondence and proof search is certainly viable and flexible enough to be used for typed mainstream programming languages. With our proof-of-concept implementation, *TypeSearch*, we show that the approach performs well for medium to large sized libraries. We are looking forward to further development of *TypeSearch* and are eager for feedback, which can be given on `https://typesearch.dev`.

References

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. 2nd. Addison-Wesley Professional, 2005. ISBN: 0321267974.

[EM24] Marc Etter and Farhad Mehta. *TypeSearch: Type-Directed API Search For All*. Planned publication. 2024.

[Ett20] Marc Etter. *Type Systems for the OO Programmer*. https://eprints.ost.ch/id/eprint/1110/. 2020.

[Fen+17] Yu Feng et al. "Component-Based Synthesis for Complex APIs". In: *SIGPLAN Not.* 52.1 (Jan. 2017), pp. 599–612. ISSN: 0362-1340. DOI: 10.1145/3093333.3009851. URL: https://doi.org/10.1145/3093333.3009851.

[Fou21] Linux Foundation. *OpenAPI Specification v3.1.0*. https://spec.openapis.org/oas/latest.html. Feb. 2021.

[Gos+23] James Gosling et al. *The Java® Language Specification — Java SE 21 Edition*. https://docs.oracle.com/javase/specs/jls/se21/html/index.html. Aug. 2023.

[Guo+19] Zheng Guo et al. "Program Synthesis by Type-Guided Abstraction Refinement". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371080. URL: https://doi.org/10.1145/3371080.

[Jam+20] Michael B. James et al. "Digging for Fold: Synthesis-Aided API Discovery for Haskell". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428273. URL: https://doi.org/10.1145/3428273.

[Man+05] David Mandelin et al. "Jungloid Mining: Helping to Navigate the API Jungle". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 48–61. ISBN: 1595930566. DOI: 10.1145/1065010.1065018. URL: https://doi.org/10.1145/1065010.1065018.

[Mit05] Neil Mitchell. *Hoogle*. Presentation from PLASMA. Dec. 2005. URL: https://ndmitchell.com/downloads/slides-hoogle-08_dec_2005.pdf.

[Mit08] Neil Mitchell. "Hoogle Overview". In: *The Monad.Reader* 12 (Nov. 2008), pp. 27–35. URL: https://ndmitchell.com/downloads/paper-hoogle_overview-19_nov_2008.pdf.

[Mit20] Neil Mitchell. *Hoogle Searching Overview*. https://neilmitchell.blogspot.com/2020/06/hoogle-searching-overview.html. 2020.

[OL20] 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*. CSAI '19. Normal, IL, USA: Association for Computing Machinery, 2020, pp. 294–298. ISBN: 9781450376273. DOI: 10.1145/3374587.3374633. URL: https://doi.org/10.1145/3374587.3374633.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press. Cambridge, MA, USA: MIT Press, 2002. ISBN: 9780262162098.

[Weg+16] Lukas Wegmann et al. “Scaps: Type-Directed API Search for Scala”. In: *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. SCALA 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 95–104. ISBN: 9781450346481. DOI: 10.1145/2998392.2998405. URL: https://doi.org/10.1145/2998392.2998405.

List of Figures

Listings

List of Tables

A Query Syntax Specification

Listings A.1 and A.2 show the formal query specification using ANTLRv4[1] grammar. This specification is used to generate the parser and Java data model for queries.

```
parser grammar QueryGrammar;

options {
    tokenVocab=QueryLexerGrammar;
}

// starting rule
query: queryString EOF;

queryString
    : modules queryExpr
    | queryExpr modules
    | queryExpr
    ;

queryExpr
    : combinedQuery
    | typeQuery
    | keywordQuery
    ;

// module filters
modules
    : moduleInclusion+
    | moduleExclusion+
    ;

moduleInclusion: ADD module;
moduleExclusion: SUB module;

module: (moduleGroup? MODULE_SEPARATOR)? moduleName (MODULE_SEPARATOR
     moduleVersion)?;
moduleGroup: MODULE_IDENTIFIER;
moduleName: MODULE_IDENTIFIER;
moduleVersion: MODULE_IDENTIFIER;

// queries
keywordQuery: ~(COLON | ARROW | MODULE_IDENTIFIER)*;
typeQuery: parameters ARROW typeExpr;
combinedQuery: keywordQuery COLON typeQuery;

// parameters
parameters
    : noParameters
    | atLeastOneParameter
    ;

noParameters: LPAREN RPAREN;

atLeastOneParameter
    : typeExprList
    | LPAREN typeExprList RPAREN
    ;

// type expressions
typeExprList: typeExpr (COMMA typeExpr)*;

typeExpr
    : simpleType
    | parameterizedType
    | functionType
    ;

simpleType
    : IDENTIFIER
```

[1] https://www.antlr.org/

```
65      | WILDCARD
66      ;
67
68  parameterizedType: IDENTIFIER LT typeExprList GT;
69
70  functionType: LPAREN parameters ARROW typeExpr RPAREN;
```

Listing A.1: Query Syntax ANTLRv4 Grammar

```
1   lexer grammar QueryLexerGrammar;
2
3   fragment IDENTIFIER_CHAR: [a-zA-Z0-9.$#_];
4
5   // DEFAULT_MODE: if the first character is a + or -, switch to
        PRE_MODULE_MODE
6   // otherwise, switch to UNDECIDED_QUERY_MODE
7   SUB: '-' -> mode(PRE_MODULE_MODE);
8   ADD: '+' -> mode(PRE_MODULE_MODE);
9
10  COLON: ':' -> mode(KEYWORD_QUERY_MODE);
11  ARROW: ('->' | '=>') -> mode(KEYWORD_QUERY_MODE);
12
13  LPAREN: '(' -> mode(UNDECIDED_QUERY_WORD_MODE);
14  RPAREN: ')' -> mode(KEYWORD_QUERY_MODE);
15  COMMA: ',' -> mode(KEYWORD_QUERY_MODE);
16  LT: '<' -> mode(KEYWORD_QUERY_MODE);
17  GT: '>' -> mode(KEYWORD_QUERY_MODE);
18
19  WILDCARD: [?_*] -> mode(UNDECIDED_QUERY_WORD_MODE); // appears before
        IDENTIFIER to have higher priority in parsing
20  IDENTIFIER: IDENTIFIER_CHAR+ -> mode(UNDECIDED_QUERY_WORD_MODE);
21  WORD: ~[ \t\r\n]+? -> mode(KEYWORD_QUERY_MODE);
22
23  WS: [ \t\r\n]+ -> skip;
24
25
26  // MODULE_MODE: is not actually used as a lexer mode, but used for common
        token types
27  mode MODULE_MODE;
28  MODULE_IDENTIFIER: ~[ \t\r\n:]+;
29  MODULE_SEPARATOR: COLON;
30
31
32  // PRE_MODULE_MODE: used when lexing modules at the beginning of the query
        input
33  // As soon as we encounter whitespace (i.e. module spec is finished), switch
        to
34  // QUERY_OR_MODULE_MODE.
35  mode PRE_MODULE_MODE;
36  PRE_MODULE_IDENTIFIER: MODULE_IDENTIFIER -> type(MODULE_IDENTIFIER);
37  PRE_MODULE_SEPARATOR: MODULE_SEPARATOR -> type(MODULE_SEPARATOR);
38  PRE_MODULE_TERMINATOR: WS -> skip, mode(QUERY_OR_MODULE_MODE);
39
40
41  // QUERY_OR_MODULE_MODE: We don't know whether to expect another module spec
42  // or a query element next. Depending on the next character, switch to the
43  // corresponding mode.
44  mode QUERY_OR_MODULE_MODE;
45  QUERY_OR_MODULE_SUB: SUB -> type(SUB), mode(PRE_MODULE_MODE);
46  QUERY_OR_MODULE_ADD: ADD -> type(ADD), mode(PRE_MODULE_MODE);
47
48  QUERY_OR_MODULE_LPAREN: '(' -> type(LPAREN), mode(UNDECIDED_QUERY_WORD_MODE);
49
50  QUERY_OR_MODULE_IDENTIFIER: IDENTIFIER_CHAR+ -> type(IDENTIFIER), mode(
        UNDECIDED_QUERY_WORD_MODE);
51  QUERY_OR_MODULE_WORD: WORD -> type(WORD), mode(UNDECIDED_QUERY_MODE);
52
53  QUERY_OR_MODULE_WS: WS -> skip;
54
55
56  // UNDECIDED_QUERY_MODE: We are lexing the actual query tokens.
57  // We do not know yet which query type we are lexing for.
58  // If we encounter a + or -, we switch to POST_MODULE_MODE.
```

```
59  mode UNDECIDED_QUERY_MODE;
60  UNDECIDED_QUERY_COLON: ':' -> type(COLON), mode(TYPE_QUERY_MODE);
61  UNDECIDED_QUERY_ARROW: ('->' | '=>') -> type(ARROW), mode(TYPE_QUERY_MODE);
62
63  UNDECIDED_QUERY_SUB: SUB -> type(SUB), mode(POST_MODULE_MODE);
64  UNDECIDED_QUERY_ADD: ADD -> type(ADD), mode(POST_MODULE_MODE);
65
66  UNDECIDED_QUERY_LPAREN: '(' -> type(LPAREN);
67  UNDECIDED_QUERY_RPAREN: ')' -> type(RPAREN), mode(KEYWORD_QUERY_MODE);
68  UNDECIDED_QUERY_COMMA: ',' -> type(COMMA), mode(KEYWORD_QUERY_MODE);
69  UNDECIDED_QUERY_LT: '<' -> type(LT), mode(KEYWORD_QUERY_MODE);
70  UNDECIDED_QUERY_GT: '>' -> type(GT), mode(KEYWORD_QUERY_MODE);
71
72  UNDECIDED_QUERY_WILDCARD: WILDCARD -> type(WILDCARD), mode(
        UNDECIDED_QUERY_WORD_MODE);
73  UNDECIDED_QUERY_IDENTIFIER: IDENTIFIER_CHAR+ -> type(IDENTIFIER), mode(
        UNDECIDED_QUERY_WORD_MODE);
74  UNDECIDED_QUERY_WORD: WORD -> type(WORD), mode(UNDECIDED_QUERY_WORD_MODE);
75
76
77  // UNDECIDED_QUERY_WORD_MODE: We are in the middle of lexing a word of a
        query.
78  // We do not know yet which query type we are lexing for.
79  mode UNDECIDED_QUERY_WORD_MODE;
80  UNDECIDED_QUERY_WORD_LPAREN: '(' -> type(LPAREN);
81  UNDECIDED_QUERY_WORD_RPAREN: ')' -> type(RPAREN);
82  UNDECIDED_QUERY_WORD_COMMA: ',' -> type(COMMA);
83  UNDECIDED_QUERY_WORD_LT: '<' -> type(LT);
84  UNDECIDED_QUERY_WORD_GT: '>' -> type(GT);
85
86  UNDECIDED_QUERY_WORD_WILDCARD: WILDCARD -> type(WILDCARD);
87  UNDECIDED_QUERY_WORD_IDENTIFIER: IDENTIFIER_CHAR+ -> type(IDENTIFIER);
88  UNDECIDED_QUERY_WORD_WORD: WORD -> type(WORD);
89
90  UNDECIDED_QUERY_WORD_WS: WS -> skip, mode(UNDECIDED_QUERY_MODE);
91
92
93  // KEYWORD_QUERY_MODE: We are lexing keywords.
94  // If we encounter a :, we switch to TYPE_QUERY_MODE.
95  // If we encounter a + or - at the beginning of a word,
96  // we switch to POST_MODULE_MODE.
97  mode KEYWORD_QUERY_MODE;
98  KEYWORD_QUERY_COLON: WS ':' -> type(COLON), mode(TYPE_QUERY_MODE);
99
100 KEYWORD_QUERY_SUB: WS SUB -> type(SUB), mode(POST_MODULE_MODE);
101 KEYWORD_QUERY_ADD: WS ADD -> type(ADD), mode(POST_MODULE_MODE);
102
103 KEYWORD_QUERY_WORD: WORD -> type(WORD);
104
105 KEYWORD_QUERY_WS: WS -> skip;
106
107
108 // TYPE_QUERY_MODE: We are lexing type expressions.
109 // If we encounter a + or -, we switch to POST_MODULE_MODE.
110 mode TYPE_QUERY_MODE;
111 TYPE_QUERY_SUB: SUB -> type(SUB), mode(POST_MODULE_MODE);
112 TYPE_QUERY_ADD: ADD -> type(ADD), mode(POST_MODULE_MODE);
113
114 TYPE_QUERY_ARROW: ('->' | '=>') -> type(ARROW);
115
116 TYPE_QUERY_LPAREN: '(' -> type(LPAREN);
117 TYPE_QUERY_RPAREN: ')' -> type(RPAREN);
118 TYPE_QUERY_COMMA: ',' -> type(COMMA);
119 TYPE_QUERY_LT: '<' -> type(LT);
120 TYPE_QUERY_GT: '>' -> type(GT);
121
122 TYPE_QUERY_WILDCARD: WILDCARD -> type(WILDCARD);
123 TYPE_QUERY_IDENTIFIER: IDENTIFIER_CHAR+ -> type(IDENTIFIER);
124
125 TYPE_QUERY_WS: WS -> skip;
126
127
128 // POST_MODULE_MODE: Used when lexing modules at the end of the query.
129 // At this point we only expect modules and cycle between this mode and
```

```
130  // POST_MODULE_MORE_MODE.
131  mode POST_MODULE_MODE;
132  POST_MODULE_IDENTIFIER: MODULE_IDENTIFIER -> type(MODULE_IDENTIFIER);
133  POST_MODULE_SEPARATOR: MODULE_SEPARATOR -> type(MODULE_SEPARATOR);
134  POST_MODULE_TERMINATOR: WS -> skip, mode(POST_MODULE_MORE_MODE);
135
136
137  // POST_MODULE_MORE_MODE: We have finished lexing a module at the end of the
         query.
138  // The only thing that may occur now is another module spec.
139  mode POST_MODULE_MORE_MODE;
140  POST_MODULE_MORE_SUB: SUB -> type(SUB), mode(POST_MODULE_MODE);
141  POST_MODULE_MORE_ADD: ADD -> type(ADD), mode(POST_MODULE_MODE);
142  POST_MODULE_MORE_WS: WS -> skip;
```

Listing A.2: Query Syntax ANTLRv4 Lexer Grammar

B Configuration Parameters

TypeSearch has various configuration properties to customize the behavior of the application. These configuration properties are provided through SpringBoot's externalized configuration[1]. Table B.1 shows all available configuration properties, as well as their default values if not specified.

Table B.1: TypeSearch Configuration Properties

Property	Description
`typesearch.plugins.java:`	
`parallel`	Whether Java language extraction should make use of multi-threaded parallel processing. Defaults to `false`.
`typesearch.plugins.java.jdk:`	
`source-zip`	Location of the ZIP archive containing the JDK source code. Defaults to `<java.home>/lib/src.zip`.
`typesearch.plugins.java.maven:`	
`artifacts`	List of Maven artifacts to download and add to the ALDM / proof context.
`dependencies`	List of Maven artifacts used only as dependencies for the indexed artifacts. Will not be added to the index.
`typesearch.search:`	
`keyword-search-implementation`	Currently, only the value `keyword` is supported.
`type-search-implementation`	`proof` (default) for the *TSaPS* search implementation, or `fingerprint` for the legacy search implementation.
`unused-query-parameter-penalty`	Penalty added to a proof for every query parameter that is not used in the proof. Defaults to `15.0`.
`typesearch.search.proof:`	
`max-search-depth`	Base for calculating the maximum number of proof rules a proof may have. The final maximum search depth is calculated based on this number and the number of type variables and input parameters of the query. Defaults to `3`.
`max-search-penalty`	Maximum penalty a proof may accrue before the proof search branch is abandoned. Defaults to `30.0`
`max-function-composition-depth`	Maximum depth of function composition allowed in a proof. Defaults to `0`.

[1] `https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.external-config`

Table B.1: TypeSearch Configuration Properties (Continued)

Property	Description
`universal-parameter-instantiation-mode`	Defines how instantiations for type variables that only occur as function parameter types are searched. `NONE`: Skip such hypotheses. `GOAL` (default): Only try types that occur in the goal. `UNIFICATION`: Defer instantiation until a later unification step. Not yet supported. `ONE`: Same as `ALL`, but terminate search as soon as an instantiation has led to a successful proof. `ALL`: Attempt instantiation with all valid types.
`resolution-candidate-search-mode`	Defines how resolution candidates are searched. `ONE`: Terminate search after a resolution has led to a successful proof. `ALL` (default): Consider all resolution candidates.
`typesearch.search.proof.rules:`	
`forall-hyp`	Penalty for adding a $\forall^{\text{hyp}}$ proof rule. Defaults to `1.0`
`hyp`	Penalty for adding a hyp proof rule. Defaults to `1.0`
`replace-hyp`	Penalty for applying a type mapping. Defaults to `1.0`
`resolution`	Penalty for adding a resolution proof rule. Defaults to `1.0`
`sub`	Penalty for adding a sub proof rule. Defaults to `1.0`

C API Specification

The REST-API of the *TypeSearch* web server is specified using the OpenAPI 3 standard [Fou21]. The REST-API specification can also be found at `https://api.typesearch.dev`. This specification is used to generate Java server code and TypeScript client code using the OpenAPI-Generator[1] tool.

```yaml
openapi: '3.0.3'

info:
  title: TypeSearch REST API
  version: 1.0.0

paths:
  /api/v1/languages:
    get:
      operationId: getLanguagesV1
      description: Returns a list of all programming languages this server supports queries for.
      tags:
        - TypeSearch
      responses:
        '200':
          description: Success
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string

  /api/v1/modules:
    get:
      operationId: getModulesV1
      description: Returns all indexed modules for all supported programming languages.
      tags:
        - TypeSearch
      responses:
        200:
          description: Success
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ModulesResponseDtoV1'

  /api/v1/search-results:
    get:
      operationId: getSearchResultsV1
      description: Retrieves search results based on the given query.
      tags:
        - TypeSearch
      parameters:
        - name: language
          in: query
          description: The programming language plugin to search through.
          schema:
            type: string
          example: Java
          required: true
        - name: query
          in: query
          schema:
            type: string
          required: true
        - name: limit
          in: query
          description: The maximum number of results to be included in the response.
```

[1] `https://github.com/OpenAPITools/openapi-generator`

```yaml
            schema:
              type: integer
              format: int32
              minimum: 1
        responses:
          '200':
            description: List of search results
            content:
              application/json:
                schema:
                  $ref: '#/components/schemas/SearchResultResponseDtoV1'
          '400':
            description: Unsupported language or invalid query.
            content:
              application/problem+json:
                schema:
                  type: object
          '422':
            description:
              The provided query was syntactically valid, but could not be processed.
              This can occur e.g. when a type referenced in the query cannot be resolved.
            content:
              application/problem+json:
                schema:
                  $ref: './apiProblem.yaml'


  /api/v1/result-feedback:
    post:
      operationId: sendResultFeedback
      description: Sends feedback information about a search result.
      tags:
        - TypeSearch
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/ResultFeedbackDtoV1'
      responses:
        '204':
          description: Feedback was sent successfully.


components:
  schemas:
    ModuleDtoV1:
      properties:
        group:
          type: string
          nullable: true
        name:
          type: string
        version:
          type: string
          nullable: true
      required:
        - name
      example:
        group: ch.ost.typesearch
        name: web-rest-api
        version: 1.4.2

    ModulesResponseDtoV1:
      additionalProperties:
        type: array
        items:
          $ref: '#/components/schemas/ModuleDtoV1'
      example:
        Java:
          - group: JDK
            name: java.base
```

```yaml
            version: 17
          - group: ch.ost.typesearch
            name: web-rest-api
            version: 1.4.2

    QueryDtoV1:
      properties:
        type:
          type: string
          enum: [KEYWORD, TYPE, COMBINED]
        value:
          type: string
      required:
        - type
        - value
      example:
        type: COMBINED
        value: "map : Stream<A>, (A -> B) -> Stream<B>"

    QueryOptionsDtoV1:
      properties:
        language:
          type: string
        includedModules:
          type: array
          items:
            $ref: '#/components/schemas/ModuleDtoV1'
        excludedModules:
          type: array
          items:
            $ref: '#/components/schemas/ModuleDtoV1'
      required:
        - language
        - includedModules
        - excludedModules
      example:
        language: Java
        includedModules:
          - name: java.base
          - group: org.springframework
            name: spring-core
            version: 6.0.10
        excludedModules: []

    QueryWithOptionsDtoV1:
      properties:
        query:
          $ref: '#/components/schemas/QueryDtoV1'
        queryOptions:
          $ref: '#/components/schemas/QueryOptionsDtoV1'
      required:
        - query
        - queryOptions

    ResultFeedbackDtoV1:
      properties:
        query:
          $ref: '#/components/schemas/QueryWithOptionsDtoV1'
        resultId:
          type: string
        rank:
          type: integer
          format: int32
          minimum: 1
        feedback:
          type: string
          enum:
            - WHAT_I_LOOKED_FOR
            - SHOULD_BE_HIGHER
            - SHOULD_BE_LOWER
            - SHOULD_NOT_APPEAR
            - OTHER
        comment:
          type: string
```

```yaml
            minLength: 10
            maxLength: 200
        required:
          - query
          - resultId
          - rank
          - feedback

    SearchResultResponseDtoV1:
      properties:
        numResults:
          type: integer
          format: int32
          minimum: 0
          example: 1
        query:
          $ref: '#/components/schemas/QueryWithOptionsDtoV1'
        results:
          type: array
          items:
            $ref: '#/components/schemas/SearchResultDtoV1'
      required:
        - numResults
        - query
        - results

    SearchResultDtoV1:
      properties:
        valueDef:
          $ref: '#/components/schemas/ValueDefDtoV1'
        score:
          type: number
          format: double
        explanation:
          type: string
          description:
            "A string containing an explanation of how this result's score
                 was calculated.
            The format of the string depends on the language and storage
                 plugins used."
      required:
        - valueDef
        - score

    ValueDefDtoV1:
      properties:
        id:
          type: string
          description:
            Usually a fully-qualified-qualifiedName including a type
                 signature to disambiguate overloaded functions.
            Note that this ID is only required to be unique within the module
                  that declares this ValueDef.
            I.e. the combination of module+id is guaranteed to be unique
                 across all possible ValueDefs.
        qualifiedName:
          type: string
          description:
            The qualified qualifiedName of this ValueDef. This qualifiedName
                 is unique within the same module, except for
            possible overloads of a function.
        shortName:
          type: string
          description:
            The qualifiedName of this ValueDef, without any qualifying
                 information, such as the declaring parent.
        parentName:
          type: string
          description:
            The qualifiedName of the "parent" that declares this ValueDef.
                 The "parent" is usually a type, such as
            the class declaring a member function.
        source:
          type: string
```

```
            description:
              An approximate representation of how this ValueDef was declared
                   in the source code.
          module:
            $ref: '#/components/schemas/ModuleDtoV1'
          documentation:
            type: string
            nullable: true
        required:
          - id
          - qualifiedName
          - shortName
          - parentName
          - source
          - module
        example:
          id: java.util.stream.Stream#map(java.util.Function)
          qualifiedName: java.util.stream.Stream#map
          shortName: map
          parentName: java.util.stream.Stream
          source: <R> Stream<R> map(Function<? super T, ? extends R> mapper)
          module:
            organization: ch.ost.typesearch
            name: web-rest-api
            version: 1.4.2
          documentation:
            /**
             * Returns a stream consisting of the results of applying the given
             * function to the elements of this stream.
             *
             * <p>This is an <a href="package-summary.html#StreamOps">
                  intermediate
             * operation</a>.
             *
             * @param <R> The element type of the new stream
             * @param mapper a <a href="package-summary.html#NonInterference">
                  non-interfering</a>,
             *               <a href="package-summary.html#Statelessness">
                  stateless</a>
             *               function to apply to each element
             * @return the new stream
             */
```

Listing C.1: TypeSearch REST API Specification

```
title: "API Problem"
description: "API problem response, as per [RFC 7807 application/problem+json
    ](https://tools.ietf.org/html/rfc7807)."
type: "object"
properties:
  type:
    description: "A URI reference [RFC3986] that identifies the problem type."
    type: "string"
    format: "uri-reference"
    maxLength: 128
  title:
    description: "A short, human-readable summary of the problem type."
    type: "string"
    maxLength: 120
  status:
    description: "The [HTTP status code](https://www.rfc-editor.org/rfc/
        rfc9110#name-status-codes)\
      \ generated by the origin server for this occurrence of the problem."
    type: "integer"
    format: "int32"
    minimum: 100
    maximum: 599
  instance:
    description: "A URI reference that identifies the specific occurrence of
        the problem."
    type: "string"
    format: "uri-reference"
    maxLength: 128
```

```
26      detail:
27        description: "A human-readable explanation specific to this occurrence of
                the\
28          \ problem."
29        type: "string"
30        maxLength: 2048
```

Listing C.2: API Problem Object Specification

D Code Metrics

Table D.1 shows various metrics of the *TypeSearch* implementation.

Table D.1: TypeSearch Code Metrics

Metric	Value
Lines of Code	26 948
Average Cyclomatic Complexity	1,74
Test Cases	898
Instruction Coverage	90 %
Branch Coverage	83 %
Method Coverage	95 %
Class Coverage	98 %