Scaps: Type-Directed API Search for Scala

Lukas Wegmann
1plusX AG, Switzerland
wegmaluk@gmail.com

Farhad Mehta Peter Sommerlad
Mirko Stocker
Institute for Software, University of Applied Sciences Rapperswil, Switzerland
{first}.{last}@hsr.ch

Abstract

Type-directed API search, using queries composed of both keywords and type signatures to retrieve definitions from APIs, are popular in the functional programming community. This search technique allows programmers to easily navigate complex and large APIs in order to find the definitions they are interested in. While there exist some effective approaches to address type-directed API search for functional languages, we observed that none of these have been successfully adapted for use with statically-typed, object-oriented languages. The challenge here is incorporating large and unified inheritance hierarchies and the resulting prevalence of subtyping into an API retrieval model. We describe a new approach to API retrieval and provide an implementation thereof for the Scala language. Our evaluation with queries mined from Q&A websites shows that the model retrieves definitions from the Scala standard library with 94% of the relevant results in the top 10.

Categories and Subject Descriptors D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques

Keywords API Search and Retrieval, Polarized Types

1. Introduction

A crucial part of creating high quality software is the reuse of existing functionality provided by in-house or third-party programming libraries. This ensures that functionality is not unnecessarily reimplemented and lowers the risk of introducing erroneous behavior. Code reused over various projects has a greater chance of being reliable since it has been well-tested in production.

Discovering existing functionality is a task that requires either deep knowledge of the relevant libraries, or appropriate tools that provide convenient access to the definitions in a library. Popular examples of such tools are code completion assistants that list the accessible members of the object in question. Although, there are several reasons why code completion is not always able to provide developers with a complete picture of suitable functionality: First, the structure of an API greatly influences the discoverability of its functionality. Indirections like factories, utility classes, extension methods and implicit conversions often hinders developers from quickly discovering library features [4, 16]. Second, programming in the functional style results in numerous abstractions of transformations over data structures. While it would be favorable to provide as much of these abstractions as possible as library functions, it becomes more difficult for users to quickly discover important features. And finally, varying naming schemes amongst programming environments further complicates API discovery. Developers used to names like `filter` and `mkString` in one environment will likely have some troubles when switching to an environment that uses `where` and `join` for the same operations. To overcome these problems, developers often resort to universal search engines like Google to find a specific implementation. While these search engines regularly provide good results, users have to scan the result pages for suitable content. Additionally, general search engines may retrieve outdated information referring to an older revision of a library.

In order to alleviate these problems search engines, like Hoogle for Haskell [8], allow searching for values based on their type signature. Hoogle retrieves definitions related to the query type ordered by their relevance to the query. This assumes that developers usually know what types they have and of what type the result should be, but do not know how to get there. The great number of questions of the form "How to create X from Y" on popular Q&A websites supports this assumption.

While the idea to use types to direct API searches is not a new one [13], there are almost no applications of this idea outside the functional programming community, even though such a tool would definitely be useful for object-oriented languages, like Scala, that leverage a high level of type safety. However, while attempting to adopt Hoogle to

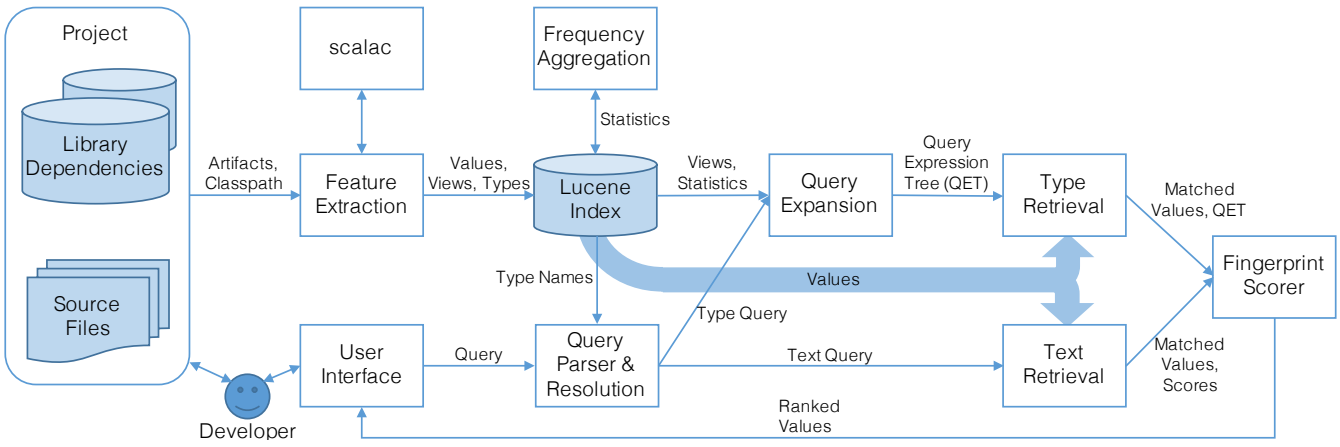

Figure 1. Dataflow amongst Scaps components

Scala, we observed, that the prevalence of subtype relations and the unified type hierarchy are major obstacles to a type graph based approach. The type graph tends to become more complex and the number of potentially matching types is bigger. This is why we propose, for Scala and similar languages, a term-based approach as discussed in this paper and implemented by the Scaps[1] search engine.

1.1 Overview

Figure 1 illustrates the architecture of the Scaps search engine and its logical building blocks. Arrows indicate the flow of data between the following components:

Developers The targeted user group of Scaps. Developers maintain the project, configure and invoke the indexing process, and issue queries.

Project The files to be indexed. This may include source files and library dependencies. The public and global definitions in these files constitute the API of the project that can be searched with Scaps.

Feature Extraction Uses the Scala compiler to extract the information required from source files and libraries. The extracted entities are value definitions, type definitions and type views. Scaps uses type views to represent subtyping, implicit conversions and similar relations between types. Additionally, each value is associated with a type fingerprint. A type fingerprint is a list of terms that characterizes the type of a value in an unstructured form.

Lucene Index Persists extracted entities and statistics. Lucene provides efficient retrieval of documents by terms.

Frequency Aggregation Aggregates the specificity of fingerprint terms. Terms with a higher specificity occur less frequently in a project's API and are assumed to be more relevant to a query.

User Interface Provides some facility to invoke queries and displays the list of retrieved values ordered by their relevance to the query. Currently, a web based search interface is available.

Query Parser & Resolution Parses a query and splits it into a list of keywords and a type. Additionally, each type referred to in the query type is resolved such that it unambiguously refers to a type in the indexed API. E.g. the query $join$: `List` $\Rightarrow$ `String` is transformed to the type query `scala.collection.immutable.List`[_] $\Rightarrow$ `java.lang.String` and the text query $join$.

Query Expansion Uses type views and statistics to determine what type fingerprint terms are relevant to a type query. The output is a query expression tree (QET) that is later used to score type fingerprints of retrieved values.

Type Retrieval Retrieves values with fingerprints containing the most dominant terms in the QET.

Text Retrieval Uses Lucene's text retrieval facilities to retrieve values whose name or doc comment matches keywords in the text query. Each retrieved value is associated with a relevance based score.

Fingerprint Scorer Evaluates the fingerprint of each retrieved value with the QET to assess its relevancy to the type query and combines this score with the score of the text retrieval.

Besides *Feature Extraction*, which depends on the Scala compiler, all components and the underlying concepts are, to some degree, agnostic to the target programming language and not only applicable to Scala. Type fingerprints, type views and query expression trees are part of the Fingerprint Evaluation Model (FEM) which is discussed in more details in section 2 to section 4. Section 5 explains how FEM is applied to Scala's type system. Section 6 discusses the experimental evaluation of the quality of our model. Finally,

[1] http://www.scala-search.org

section 7 reviews related work and section 8 concludes the paper.

2. Type Fingerprints

We use some ideas of the Vector Space Model (VSM) [14] to quickly select potentially matching types from the index and reduce the number of types that need to be compared to the query type. One of the basic assumptions of VSM is that documents and queries can be decomposed into distinct terms where each term characterizes the document or query independently of the other terms. Hence, to find documents that are relevant to a query, it is sufficient to only consider what terms are used in the document. The semantic relations between these terms are assumed to be irrelevant during the retrieval process. Also, the order of terms does not influence the retrieval results. This assumption allows to use a simple representation of the documents (e.g., bag-of-words) and fast retrieval techniques like inverted indexes. With type fingerprints, we propose such an unstructured representation of types.

A type fingerprint of a type T is a list of the type names occurring in the normalized proper type T'. Furthermore, each type name is annotated with the variance polarity of its position in the type. To start with an illustrative example, the type $\forall A.\texttt{List}[A] \Rightarrow \texttt{Int} \Rightarrow A$ of a polymorphic function `elementAt` is represented by the type fingerprint $\{-\texttt{List}, -\top, -\texttt{Int}, +\bot\}$ (using $\top$ and $\bot$ for the top and bottom type respectively).

The first step in the fingerprint transformation is to annotate each part of a type with its variance polarity. This step is formalized in Polarized System $Ob_{\omega<:\mu}$ [3]. In the following, we use the notation $+T$, $-T$ and $\circ T$ to refer to a type T occurring at a covariant, contravariant or invariant position respectively and the notation $T :: K$ to refer to a type T of kind K. A type T used at a position with polarity p is written as pT. The polarity of a type of a value is $+$. Given a type constructor $C :: * \xrightarrow{p} *$ with polarity p, the application $C[T]$ at a position with polarity q is annotated with $qC[pqT]$. The composed polarity pq is given in table 1. The transformation from a type T in a context p to an annotated type is given by the function $\alpha(T, p)$:

$$\alpha(C[T_1, ..., T_n], p) = pC[\alpha(T_1, pq_1), ..., \alpha(T_n, pq_n)] \quad \text{if } C :: k_1 \xrightarrow{q_1} ... \xrightarrow{q_n} k_n \tag{1}$$

$$\alpha(\forall X_1, ..., X_n.T, p) = \forall X_1, ..., X_n.\alpha(T, p) \tag{2}$$

$$\alpha(T, p) = pT \tag{3}$$

For example, the function type constructor $\Rightarrow :: * \xrightarrow{-} * \xrightarrow{+} *$ is contravariant over the first argument and covariant over the second argument. Furthermore, the Scala standard library defines type $\texttt{List} :: * \xrightarrow{+} *$. Accordingly, $\alpha(\forall T.\texttt{List}[T] \Rightarrow \texttt{List}[T], +) = \forall T.+\Rightarrow[-\texttt{List}[-T], +\texttt{List}[+T]]$.

Table 1. Composition of polarized type constructors pq [3]

	$\circ$	$+$	$-$
$\circ$	$\circ$	$\circ$	$\circ$
$+$	$\circ$	$+$	$-$
$-$	$\circ$	$-$	$+$

The second step is to transform annotated types to proper annotated types of kind $*$. This is achieved by substituting type parameters by their upper or lower bounds respectively, depending on their polarity. The notation $T :: k(L, U)$ refers to a type variable T of kind k with a lower bound L and an upper bound U. If omitted, the bounds $\bot$ and $\top$ are implied. Occurrences of $+T$ are substituted by $+L$, $-T$ by $-U$ and $\circ T$ by the unknown, polarized type $\circ?$. For example, a function of type $\forall T :: *(\bot, \texttt{Num}).T \Rightarrow T$ has a proper annotated type $+\Rightarrow[-\texttt{Num}, +\bot]$.

The final step is to flatten the proper annotated type to a multiset of polarized type names. Hence, the annotated proper type $+C[-X, -X]$ is flattened to the fingerprint $\{+C, -X, -X\}$. In the following, we use the notation $\text{FP}(T)$ to refer to the fingerprint of a type T.

Altogether, we assume that the fingerprint representation of types retains sufficient information to characterize a definition while it allows term-wise comparison. Given a suitable similarity function, which we will discuss in section 4.2, it is possible to retrieve definitions of similar types to a query type without specifying matching rules as used in [13] and [20]. This advantage becomes more apparent when comparing fingerprints of the following definitions:

$$FP(v_1 : \texttt{1} \Rightarrow \texttt{Int}) = \{+\Rightarrow, -\texttt{1}, +\texttt{Int}\}$$
$$FP(v_2 : \texttt{Int} \Rightarrow \texttt{1}) = \{+\Rightarrow, -\texttt{Int}, +\texttt{1}\}$$
$$FP(v_3 : (\texttt{Int} \Rightarrow \texttt{1}) \Rightarrow \texttt{1}) = \{+\Rightarrow, -\Rightarrow, +\texttt{Int}, -\texttt{1}, +\texttt{1}\}$$
$$FP(v_4 : (\texttt{1} \Rightarrow \texttt{Int}) \Rightarrow \texttt{1}) = \{+\Rightarrow, -\Rightarrow, +\texttt{1}, -\texttt{Int}, +\texttt{1}\}$$

Both definitions v_1 and v_3 share the terms $+\Rightarrow, -\texttt{1}$ and $+\texttt{Int}$. In fact, v_1 and v_3 behave similar from a callers point of view. In both cases, the caller may receive a value of `Int`, either immediately returned by v_1 (if v_1 terminates) or passed to the callback provided as an argument to v_3 (if the callback is invoked by v_3). Accordingly, v_2 and v_4 both consume an `Int` and their similar behavior is expressed by similar fingerprints. Along this line, other isomorphisms and relaxed equivalence relations between types are sustained by the fingerprint transformation: Curried and tupled forms of functions, e.g., $\{-U, -V, +T\}$ is a subset of both $\text{FP}((U, V) \Rightarrow T)$ and $\text{FP}(U \Rightarrow V \Rightarrow T)$. Boxed versus unboxed types, e.g., $\text{FP}(U \Rightarrow V) \subset \text{FP}(C[U] \Rightarrow D[V])$. Reordering of arguments, e.g., $\text{FP}(U \Rightarrow V \Rightarrow T) = \text{FP}(V \Rightarrow U \Rightarrow T)$. And, renaming of type parameter names, e.g., $\text{FP}(\forall A.\forall B.A \Rightarrow B \Rightarrow A) = \text{FP}(\forall X.\forall Y.X \Rightarrow Y \Rightarrow X)$.

Also, the loss of information during the fingerprint transformation is less severe than one would expect at a first glance. When applied to actual APIs, the number of finger-

print collisions is typically small. Hence, there are only few identical fingerprints that originated from definitions with different types. For example, the `List` class in the Scala collection library defines and inherits 177 operations with 118 distinct type signatures. After applying FP 107 distinct fingerprints are left. From these fingerprints there are 7 derived from more than one type. Furthermore, almost all of these fingerprint collisions occur between operations that have very similar semantics like `foldLeft`/`foldRight` and `reduceLeft`/`reduceRight`. Collisions between unrelated operations occur only due to methods inherited from Java's `Object` type which often have a more liberal type signature than necessary. One such case is `equals` : $\forall A.\texttt{List}[A] \Rightarrow \top \Rightarrow \texttt{Bool}$ and `contains` : $\forall A.\texttt{List}[A] \Rightarrow A \Rightarrow \texttt{Bool}$ where both methods share the fingerprint $\{+\Rightarrow, -\texttt{List}, -\top, +\Rightarrow, -\top, +\texttt{Bool}\}$. Due to the proper type conversion it is not possible to distinguish between the top type $\top$ in `equals` and the unconstrained type parameter used at contravariant position in `contains`.

3. Type Views

In order to consider subtyping relations between types, we use type views to judge whether an annotated type in a fingerprint is "viewable" as another annotated type. This is achieved by introducing additional judgements of the form $\Gamma \vdash p_1 T \rhd p_2 U$ which denote that "a type T used at a position with variance p_1 can be viewed as a type U with variance p_2". Like subtyping, type views are reflective and transitive:

$$\frac{}{\Gamma \vdash pU \rhd pU} \text{ (V-REFL)}$$

$$\frac{\Gamma \vdash pU \rhd pV \qquad \Gamma \vdash pV \rhd pW}{\Gamma \vdash pU \rhd pW} \text{ (V-TRANS)}$$

Furthermore, we use additional rules to derive type views from the inheritance hierarchy:

$$\frac{U \texttt{ extends } V}{\Gamma \vdash \alpha(V, +) \rhd \alpha(U, +)} \text{ (V-EXT-COV)}$$

$$\frac{U \texttt{ extends } V}{\Gamma \vdash \alpha(U, -) \rhd \alpha(V, -)} \text{ (V-EXT-CONV)}$$

Thus, an inheritance declaration derives type views for a co- and contravariant context. For example, a class definition `class IntList extends List[Int]` derives the type views $+\texttt{List}[+\texttt{Int}] \rhd +\texttt{IntList}$ and $-\texttt{IntList} \rhd -\texttt{List}[-\texttt{Int}]$. We deliberately do not derive type views from the available subtype judgements because we want to exclude most features of the type system like polarized application of type constructors. This ensures that a definition like `class List[+T] extends It[T]` does only derive the abstract type view $\forall U.-\texttt{List}[-U] \rhd \forall U.-\texttt{It}[-U]$ and no applied statements like $-\texttt{List}[-\texttt{Int}] \rhd -\texttt{It}[-\top]$. V-EXT-COV and V-EXT-CONV are a consequence of the subtype rules on polarized type constructors discussed in [3] but with reversed direction. This is motivated by the fact that the aim is to retrieve subtypes of the query type. For example, the statement $\texttt{IntList} :> \texttt{List}[\texttt{Int}]$ implies, that a query for $\texttt{List}[\texttt{Int}]$ can be answered with a definition of type `IntList`. Thus, the terms $+\texttt{List}$ and $+\texttt{Int}$ in the query fingerprint can be substituted by $+\texttt{IntList}$ which is encoded by the type view $+\texttt{List}[+\texttt{Int}] \rhd +\texttt{IntList}$.

A type used at co- or contravariant positions may also match the same type in an invariant context:

$$\frac{}{\Gamma \vdash pU \rhd \circ U} \text{ (V-INV-ANY)}$$

This rule is a consequence of the order of information content of polarities [3]. Therefore, (V-INV-ANY) simply considers less restricted types. For example, both terms $-\texttt{Int}$ and $+\texttt{Int}$ from the queries $\texttt{Int} \Rightarrow \texttt{Unit}$ and $\texttt{Unit} \Rightarrow \texttt{Int}$ respectively should match $\circ\texttt{Int}$ derived from a definition of type $\texttt{Array}[\texttt{Int}]$ because the array both consumes and produces elements of type `Int`.

And finally, every invariant type can be seen as the unknown type ? in an invariant context:

$$\frac{}{\Gamma \vdash \circ U \rhd \circ ?} \text{ (V-INV-?)}$$

This accords to the proper annotated type transformation discussed in section 2 that maps type parameters at invariant positions to $\circ?$. Thus, a query type $\texttt{Ref}[\texttt{Int}]$ with $\texttt{Ref} :: * \xrightarrow{\circ} *$ should match definitions of the same type or a polymorphic type $\forall T.\texttt{Ref}[T]$ with the fingerprint $\{+\texttt{Ref}, \circ?\}$. The polarized type $\circ?$ is a wildcard that carries no further information than some type in some context is expected.

Type views form a simple system to reason about subtype relations between annotated types. This concept is necessary to efficiently expand types to match possible subtypes of the query type as discussed in section 4. In section 5 we will introduce additional rules for type views that allow to incorporate implicit type coercions into the retrieval process.

4. Query Expression Trees

A Query Expression Trees (QET) is an expanded representation of a query type. During type retrieval, a QET fulfills two functions: First, a QET is used to identify dominant fingerprint terms that have a high relevance to the query. Second, the fingerprints retrieved from the index with these dominant terms are evaluated with the QET to score the relevance of the whole fingerprint.

A query expression tree (QET) consists of three types of nodes: *sum* nodes, *max* nodes and *leaf* nodes. A *leaf* node represents a single fingerprint term associated with a score that captures the relevance of the fingerprint term to the query. For instance, $+\texttt{List}^{0.6}$ is a leaf node with a score of 0.6. A *sum* node combines subtrees that can be matched together. During evaluation, the score of a sum tree is the sum of the scores of all matched subtrees. For instance, $E = \oplus(+\texttt{List}^{0.6}, +\texttt{Int}^{0.4}, +\texttt{Bool}^{0.2})$ is a sum node with three subtrees. Applying the fingerprint $\{+\texttt{List}, +\texttt{Bool}\}$

to E yields a score of 0.8. Finally, a *max* node combines alternative subtrees. The score of a max node is the score of the subtree that matched with the highest score. For instance, the max node $\oslash(\oplus(+\texttt{List}^{0.6}, +\bot^{0.1}), +\bot^{0.2})$ with two subtrees yields a score of 0.7 when applied with $\{+\texttt{List}, +\bot\}$ and a score of 0.2 when applied with $\{+\bot\}$.

4.1 Constructing QET

A QET is constructed by applying the function QET to a proper annotated query type pT:

$$\text{QET}(pT) = \text{MAX}(pT, 1) \tag{4}$$

$$\text{MAX}(pT, f) = \oslash\{\text{SUM}(qA, f_a, d_a) \mid pT \rhd qA\}$$
$$\text{where } f_a = f \cdot (1 - \text{COST}(pT \rhd qA)),$$
$$d_a = \begin{cases} 0, & \text{if } pT = qA \\ 1, & \text{otherwise} \end{cases} \tag{5}$$

$$\text{SUM}(pC[qT_1,...,qT_n], f, d) =$$
$$\oplus(\text{SUM}(pC, f_c, d), \text{MAX}(qT_1, f_c), ..., \text{MAX}(qT_n, f_c))$$
$$\text{where } f_c = \frac{f}{1+n} \tag{6}$$

$$\text{SUM}(pT, f, d) = pT^s$$
$$\text{where } s = f \cdot (1 - dw_d) \cdot \text{ITF}(pT) \tag{7}$$

Equation (4) defines that a QET is constructed with MAX and an initial fraction value f of 1. Equation (5) states that MAX of a proper annotated type pT is a $\oslash$-node including all types that are viewable from pT mapped over SUM with the cost-adjusted fraction value f_a and a distance value d_a.

Furthermore, eq. (6) states that SUM of a type constructor pC with the arguments qT_1 to qT_n is a $\oplus$-node with the type constructor without arguments applied to SUM and the arguments applied to MAX. The fraction f is evenly distributed over the parts created from the type constructor pT and the arguments pT_i. Equation (7) states that SUM of a type pT without arguments is a leaf node pT^s weighted with the score s.

As defined in eq. (7), each QET leaf node is weighted with a score composed of its fraction value f, the distance d and the *Inverse Term Frequency* (ITF) of the term associated with the node. In the following, we will further elaborate these three statistics and motivate their use.

Fractions are motivated by the observation, that not all leafs in a QET represent the same fraction of a query. For example, the type views $+\texttt{List}[+\texttt{Int}] \rhd +\texttt{IntList}$ and $+\texttt{Int} \rhd +\bot$ expands the query type `List[Int]` to the following QET:

$$\oslash\left(\oplus\left(+\texttt{List}, \oslash\left(+\texttt{Int}, +\bot\right)\right), +\texttt{IntList}\right)$$

While $+\texttt{List}$, $+\texttt{Int}$ and $+\bot$ each represent half of the query, $+\texttt{IntList}$ replaces the complete original type and should therefore be weighted accordingly with a fraction of 1.

Another issue is, that some views are destructive and information carried in type arguments is lost by applying such views. For example, a view derived from the Scala standard library is $\forall A.-\texttt{List}[-A] \rhd -\texttt{Immutable}$ where `Immutable` is a marker trait for immutable data structures. This view is destructive as it does not retain information about `List`'s type argument. To limit the weight of QET branches derived from destructive view applications eq. (5) uses an adjusted fraction $f_a = f \cdot (1 - \text{COST}(pT \rhd qA))$. A good estimate of this COST function is the number of type parameters in the original type pT that do not occur in the alternative type qA over the number of atomic types pT is constructed of. Hence, $\text{COST}(\forall A.-\texttt{List}[-A] \rhd -\texttt{Immutable})$ is 0.5 and $\text{COST}(\forall A.-\texttt{List}[-A] \rhd -\texttt{It}[-A])$ is 0.

The *Inverse Type Frequency* (ITF) is inspired by the Inverse Document Frequency (IDF) statistic commonly used by implementations of the vector space model [14]. Like IDF, ITF should capture the specificity of a fingerprint term such that types occurring more frequently in the document collection can be penalized. But using the plain number of definitions that contain a fingerprint term is not very accurate in capturing its specificity because subtype relations are not considered. Instead, we use a different notion of specificity: *The specificity of a fingerprint term pT is the inverse of the probability $P(pT)$ that the term will occur in a query expression.* Hence, terms occurring in almost all QET expanded from arbitrary queries, like $-\top$ and $+\bot$, have a relatively low specificity. To approximate a term's probability $P(pT)$ we use the types of all definitions in the document collection D as hypothetical queries. The number of expanded QETs that include the term pT gives the absolute document type frequency $\text{df}(pT)$ and the probability $P(pT) = \text{df}(pT)/|D|$. Finally, we define ITF as

$$\text{ITF}(pT) = 1 - \ln\Big(e\,P(pT) + \big(1 - P(pT)\big)\Big)$$

The factor e and the term $(1 - P(pT))$ is added to normalize ITF in the range $[0, 1]$. This is not a necessity but makes it easier to incorporate ITF in leaf scores and relate it to the other scoring statistics.

The distance factor is used to slightly boost terms occurring in the original query type over derived terms. This is necessary, because some implicit conversions (see section 5) derive bidirectional views such that $+A \rhd +B$ and $+B \rhd +A$ which leads to $\text{ITF}(+A) = \text{ITF}(+B)$ and equal scores for definitions of type A and B if a user issues the query A. w_d in eq. (7) is a parameter to the model that determines the penalty for derived terms.

Besides fractions, distance and ITF, we also considered other scoring factors that did not result in the expected improvements of the retrieval model. One statistic that has been tested is the inverse of the number of $\oplus$ nodes between the leaf and the root of a QET. This *depth* statistic was motivated by the assumption that types at a deeper nesting level are less relevant to a user's information need, e.g. C is less relevant

than A or D in the type query $A[B[C]] \Rightarrow D$. Although, this assumption was not supported by our evaluation results. Additionally, we used a more sophisticated distance statistic that incorporated the number of steps between the original term and an alternative term in the class linearization of the type hierarchy [11]. Later, we observed, that ITF alone is sufficient to discriminate terms by their distance to the original type. Because a definition contributing to the probability $P(pA)$ does also increase $P(qB)$ if $pA \rhd qB$, ITF(pA) is always greater or equal to ITF(qB).

4.2 Evaluating QET

The evaluation semantics of applying a single fingerprint term to a QET are straightforward:

$$\text{EVAL1}(pT, \oplus(c_1, ..., c_n)) = \sum_{i=1}^{n} \text{EVAL1}(pT, c_i) \tag{8}$$

$$\text{EVAL1}(pT, \oslash(c_1, ..., c_n)) = \max\{\text{EVAL1}(pT, c_1), ..., \text{EVAL1}(pT, c_n)\} \tag{9}$$

$$\text{EVAL1}(pT, qU^s) = \begin{cases} s & \text{if } p = q \wedge T = U \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Because addition distributes over the maximum operator we can conclude that QETs are distributive over $\oplus$. Hence, $\oslash(\oplus(A, X), \oplus(B, X)) = \oplus(\oslash(A, B), X)$. Furthermore, $\sum_{i \in \{e\}} i = e$ and $\max\{e\} = e$ which allows to omit intermediate nodes with exactly one subtree: $\oslash(\oplus(A)) = A$. Those properties are useful for compressing QETs before evaluation to reduce the number of required evaluation steps. Especially the out-factorization of common subtrees helps to distinctly reduce the size of QETs. In the following, we will also implicitly apply some of those transformations to give a more comprehensible view of example trees.

A crucial part of the Fingerprint Evaluation Model is the function EVAL(q, f) that calculates the score of the fingerprint f applied to the QET q. Just mapping all terms t_i in f over EVAL1(q, t_i) and accumulating the resulting scores, violates the restriction on *max* nodes that at most one sub expression per $\oslash$-node can contribute to the score of a fingerprint. For example, the fingerprint $\{-\texttt{Seq}, -\top, +\bot\}$ (derived from $\forall A.\texttt{Seq}[A] \Rightarrow A$) should yield the same score as $\{-\texttt{Seq}, -\top, -\top, +\bot\}$ ($\forall A.\texttt{Seq}[A] \Rightarrow A \Rightarrow A$) when applied to the QET given in fig. 2, even tough the latter contains the matching term $-\top$ twice. Instead, evaluating QETs can be seen as the optimization problem of marking terms of the fingerprint in the QET such that the sum of the scores of marked leafs is maximized, without violating the evaluation semantics described at the beginning of section 4. To efficiently solve this problem, we use an heuristic that yields sufficiently accurate results in $O(n * m + m \log m)$ where n is the size of the QET and m is the number of terms in the fingerprint [17].

Besides differences between distinct types of definition and query, also varying complexity between those types

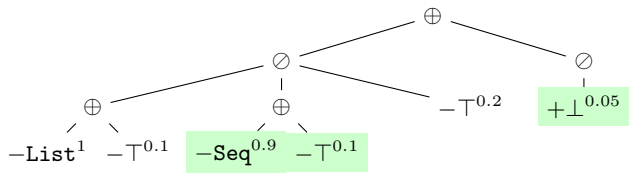

Figure 2. QET of the query $\forall A.\texttt{List}[A] \Rightarrow A$; the terms contributing to the score of the fingerprint $\{-\texttt{Seq}, -\top, -\top, +\bot\}$ are highlighted in green

can (but not necessarily should) influence the similarity between definition and query type. For example, given a query $q : A \Rightarrow C$ with less arguments than a definition $d : A \Rightarrow B \Rightarrow C$. d will not be penalized during QET evaluation despite the superfluous argument B. A useful metric to capture these differences is the number of distinct fingerprint terms that can be removed from a fingerprint d without affecting the score when evaluating a QET q with d:

$$\text{PEN}(d, q) = \frac{1}{w_p|\{t \in d \mid \text{EVAL}(d, q) = \text{EVAL}(d \setminus t, q)\}| + 1}$$

This gives a factor PEN$(d, q) \in [0, 1]$ that penalizes partially matched fingerprints and is used to get an adjusted fingerprint score:

$$\text{SCORE}(d, q) = \text{PEN}(d, q) \cdot \text{EVAL}(d, q)$$

Note, that the given definition of PEN differs heavily from a feasible implementation. The actual implementation requires no additional evaluation of q and is incorporated in the EVAL function.

4.3 Indexing and Retrieving API Definitions

To ensure fast access to API definitions, we use an index that maps fingerprint terms to definitions. During retrieval, it is sufficient to fetch only those definitions that are likely to achieve good scores and evaluate the according fingerprints against the QET. A good approximation of potentially high scoring fingerprints is to consider only fingerprints containing at least one of the dominant terms in the QET.

To get the dominant terms we use the list of terms occurring in the QET ordered by descending associated weights. The longest prefix of this list whose sum of type frequencies is below a certain threshold is then considered as the list of dominant terms. For example, the QET in fig. 2 consists of the ordered terms $\{-\texttt{List}, -\texttt{Seq}, -\top, -\bot\}$. Given according type frequencies and a certain threshold, $\{-\texttt{List}, -\texttt{Seq}\}$ may be the prefix of dominant terms. Using type frequencies to limit the number of dominant terms ensures that also queries with relatively common types, like $\texttt{Int} \Rightarrow \texttt{String}$, and queries with very specific types will have approximately the same number of potentially high scoring fingerprints. Applying this technique to an index of more than 100'000 definitions reduced the number of required fingerprint evaluations per query to a few thousand. For larger collections,

the frequency threshold for selecting dominant terms is also a useful parameter to balance performance and precision of the search engine.

5. Mapping Language Features

With FEM, we have presented a general model to retrieve API definitions with type signatures. To demonstrate the applicability of this model, we provide an implementation for the Scala programming language. The implementation defines a mapping of various language features to type fingerprints and views.

All definitions in the value namespace of a library are added to the index. This includes class and object members, constructors and object definitions. The mapping of Scala types to Polarized $Ob_{\omega<:\mu}$ is relatively straightforward. Because we did not yet consider higher-kinded type parameters in our model, all type parameters are considered to be of kind $*$. Furthermore, due to Scala's definition-site variance annotations, the polarity of argument types can be determined by looking up the definition of the according type constructors.

In order to give a unified view on class members and global definitions, all value, method and constructor types are mapped to according function types before passed to the fingerprint transformation function. Thus, a method definition `def f(a: A): B` is considered to be of type $A \Rightarrow B$ if defined on an object and of type $C \Rightarrow A \Rightarrow B$ if defined as a member of class C. Accordingly, a value `val v: A` is of type A or $C \Rightarrow A$ respectively.

By applying the view rules given in section 3, type views are derived from all public class and trait definitions.

Furthermore, we can apply some extensions to basic FEM to support some Scala specific language features. Especially user-defined implicit conversions are interesting to demonstrate the flexibility of the type view abstraction:

$$\frac{\texttt{implicit def } f(x:U):V}{\Gamma \vdash \alpha(V,+) \rhd \alpha(U,+)} \text{ (V-IMPL-COV)}$$

$$\frac{\texttt{implicit def } f(x:U):V}{\Gamma \vdash \alpha(U,-) \rhd \alpha(V,-)} \text{ (V-IMPL-CONV)}$$

Thus, an implicit conversion from A to B derives the two type view judgements $-A \rhd -B$ and $+B \rhd +A$.

Other language features are not yet incorporated into the model or have been purposefully omitted. While structural typing would be an interesting extension to the basic model, especially with focus on other languages that solely use structural subtyping, we observed that this feature is rarely used in Scala libraries and there was little need to apply such an extension. Another, not yet supported, feature are implicit parameters that are frequently used to implement the *type class pattern* in Scala [9]. Currently, the `implicit` keyword in parameter lists is ignored during feature extraction. As already mentioned, also higher-kinded type parameters are only partially implemented. This is to some extent a necessity because full support for the combination of higher-kinded types and subtyping enables queries that would no longer terminate during the expansion to expression trees. Furthermore, such a high level of detail was not necessary to retrieve results with a high accuracy from the test collection.

6. Evaluation

The implementation described in section 5 has been evaluated to check whether our model and its various extensions bring the expected improvements to the effectiveness of API retrieval.

6.1 Methodology

We assembled a test collection of 61 queries covering two Scala libraries: The Scala standard library (2.11.7) and *scala-refactoring* (0.6.2) [15]. The standard library is an obvious choice. It provides many useful tools like mutable and immutable collections, concurrency primitives and I/O helpers. Especially the collection part of the library provides a vast number of operations on various types that are not necessarily easy to discover. As a representative of a domain specific library we included *scala-refactoring*.

Each test query is associated with a list of definitions that are relevant to the according information need. The main source of those information needs has been the Q&A website *stackoverflow.com* and our personal experience with *scala-refactoring*. To collect the sample queries, we looked for questions tagged with "Scala" that have been answered with a reference to one or more definitions in the standard library. Afterwards, we formulated one or more queries that represent the question and identified matching definitions. We also ensured that various usage patterns are covered by the test collection. Thus, the collection includes both navigational and informational queries, generic and specific formulations of the same information need (e.g. `sort` : `Array`$[$`Int`$]$ $\Rightarrow$ `Unit` vs. `sort` : `Array`$[$`A`$]$ $\Rightarrow$ `Unit`) and queries using some textual keywords or none (e.g. `join` : `List`$[A]$ $\Rightarrow$ `String` vs. `List`$[A]$ $\Rightarrow$ `String`).

To measure how well an instantiation answered a particular query we use the *average precision* (AP) metric [7] which equals to the area under the precision/recall curve. Furthermore, the effectiveness over a complete test collection is the *mean average precision* (MAP) which is the arithmetic mean of the AP of all queries in the collection.

Because the various weighting factors of FEM influence the effectiveness of the model, the first step of the evaluation is to find a good parameterization. To ensure a fair comparison we use an automated approach to train each instantiation. And, to avoid overfitting, we randomly split the test collection into a training and a validation set of equal size. For each instantiation we randomly generate parameterizations and use the training set to find the parameterization yielding the highest MAP. Afterwards, the best parameterizations are compared between instantiations using the validation set.

Table 2. Evaluated instantiations of the prototype with the according configurations; numbers in brackets denote the range in which the according parameters are generated

	No. Configurations	Type Frequencies	Polarized Types	Type Views	w_p - Penalty Weight	w_d - Distance Weight	Doc Boost
I_0	20	-	-	-	-	-	[0,0.5]
I_1	20	-	on	-	-	-	[0,0.5]
I_2	100	on	-	-	[0,0.3]	-	[0,0.5]
I_3	100	on	on	-	[0,0.3]	-	[0,0.5]
I_4	200	on	on	on	[0,0.3]	[0,1]	[0,0.5]

Table 3. Evaluation results with the best parameterizations for each instantiation; all parameters and results have been rounded to two decimal places

	w_p	w_d	Doc	**MAP**	R_5	R_{10}
I_0	-	-	0.39	**0.36**	0.47	0.50
I_1	-	-	0.39	**0.44**	0.52	0.59
I_2	0.24	-	0.19	**0.66**	0.70	0.76
I_3	0.26	-	0.21	**0.67**	0.71	0.82
I_4	0.03	0.27	0.10	**0.79**	0.85	0.94

Table 2 lists the setup of the individual instantiations with the ranges in which parameters are generated. *No. Configurations* states how many parameterizations are generated and compared within one instantiation. I_0 is the baseline instantiation that is used to estimate the performance of a simple text-based API search tool. It roughly corresponds to applying a regular expression on the Scaladoc documentation of an API. While inherited members can be discovered, type hierarchies are not considered in argument and result types. Disabling polarized types in I_0 causes fingerprints of definitions and queries to be created in an invariant context and all type parameters are mapped to $\circ?$. I_1 extends the baseline approach with polarized types. I_2 also uses non-polarized types but penalization of non-matching terms and type frequencies are enabled. This corresponds to a more sophisticated text-based approach that could be implemented with common text retrieval techniques. I_3 uses an identical setup but with polarized types. And finally, I_4 instantiates the complete fingerprint evaluation model as presented in this paper including type views and the distance factor.

6.2 Results

Table 3 lists the best parameterizations of each instantiation and the scores yielded by applying the instantiations to the validation set. Besides MAP, the table also includes recall at 5 (R_5) and recall at 10 (R_{10}) which is easier to interpret. For example, I_0 ranked 47% of the relevant results in the top 5 and 50% in the top 10. FEM increased this value to 85% and

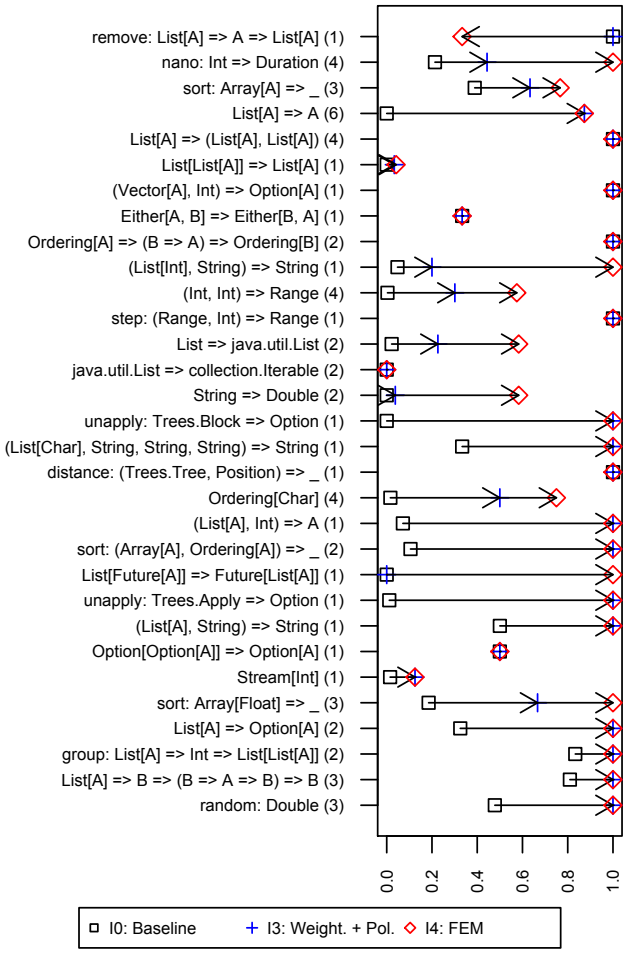

Figure 3. Per query comparison of the average precision (AP) of selected instantiations; the numbers in parens indicate how many definitions have been identified as relevant to the query; arrows indicate how AP changes with increasing complexity of the instantiation

94% respectively. As expected, all three scores increase with additional features enabled (from I_0 to I_4).

Figure 3 compares the per query scores of I_0, I_3 and I_4 of all 31 queries in the validation set. I_0, I_3 and I_4 answered 19%, 52% and 61% of the queries with a perfect AP of 1. Thus all n relevant results of a query have been listed with ranks 1 to n. On the other hand, the fraction of queries with AP of 0 is 19%, 6% and 3% respectively. In our setup, AP = 0 denotes that none of the relevant results have been listed in the top 100 results. Using the Wilcoxon signed rank test we find significant improvements on AP from I_0 to I_3 ($p < 0.01$) and from I_3 to I_4 ($p < 0.05$).

From a performance point of view, preparing the index from the source files and collecting the type frequency statistics took about 3 minutes. In total, 110'888 definitions, 2'116 types and 27'091 type views have been extracted

from the two libraries. Furthermore, the average query time has been between 250 ms and 400 ms depending on the instantiation. Because we used the same infrastructure for all instantiations, a more detailed performance comparison would not be reasonable with this setup.

6.3 Discussion

Given the data presented, we can draw the following conclusions: First, comparing I_0 to I_1 and I_2 to I_3 shows that the polarity of the position at which a type is used carries information that can be leveraged to improve the performance of API retrieval. Second, a greater improvement in accuracy can be achieved when incorporating type frequencies and the penalization of unmatched terms (comparing I_0 to I_2 and I_1 to I_3). Third, type views further broaden the kinds of queries that can be answered with API retrieval (I_4).

While the data allows relative statements about the performance of the system, it does not state whether the search engine will be perceived as a useful tool by users. A clear answer to this question would require further user studies. Although, in our personal experience, the search engine delivers helpful results when working with large Scala libraries that mainly use basic type system features like inheritance, implicit conversions and first-order type constructors. For libraries that use more complex types, the quality of the search results is less stable and a more detailed knowledge of the library structure is necessary to formulate effective queries.

One notable result can be observed with the query `List`$[$`Future`$[A]]\Rightarrow$`Future`$[$`List`$[A]]$ whose expected result is `Future.sequence` with the signature

```
def sequence[A, M[X] <: TraversableOnce[X]](
    in: M[Future[A]])(
    implicit cbf:
      CanBuildFrom[M[Future[A]], A, M[A]],
    executor: ExecutionContext): Future[M[A]]
```

This query includes many aspects that can complicate API retrieval in Scala. First, users are likely to use the familiar type `List` in the query instead of more general base types like `TraversableOnce`. Second, the relevant definition uses a highly generalized signature with the higher-kinded and bounded type parameter `M`. And finally, `sequence` also depends on the implicit context variables `cbf` and `executor`, which further complicates the resulting fingerprint. Nevertheless, the search engine successfully retrieves the correct definition.

The query `remove` : `List`$[A] \Rightarrow A \Rightarrow$ `List`$[A]$ with the expected result `diff`$[B >: A]$: `List`$[A] \Rightarrow$ `GenSeq`$[B] \Rightarrow$ `List`$[A]$ illustrates another difficulty with API retrieval: Finding a good parametrization that trades off scores contributed by keyword matches, type matches and structural matches. In contrast to I_0 and I_3, I_4 retrieved the expected definition with rank 3 which results in an AP of 0.33. In this case, I_4 prioritized structural matching higher than the occurrence of the keyword `remove` in the doc comment of `diff`.

Altogether, the evaluation indicates that FEM is suitable to answer complex API queries and precision remains high even if the relevant definition is not a subtype of the query.

Concerning the performance of the search engine, we can state that the response time is not impressive but sufficient for interactive use. Experiments with a greater corpus of indexed libraries showed that the main factor contributing to query time is not API size but the size of the QET which depends on the number of sub-/super-types of the individual types in the query. This number is relatively high for classes of the Scala collection hierarchy, for example, `List` has 35 supertypes and the most generic collection type `GenTraversableOnce` has 352 subtypes. Other Scala libraries tend to have less complex type hierarchies and our model performs accordingly better.

7. Related Work

The related work on API retrieval can roughly be grouped into four topics: code completion, signature matching, code synthesis and code snippets retrieval.

Code completion is used in integrated development environments (IDE) to display available methods that can be invoked on a given object. Often, completion assistants also offer some means to filter the displayed set of methods by name or, sometimes, by return type. Depending on the features of the target language, code completion may also include methods provided through extension methods (C#) or implicit classes (Scala). For example, the *Scala IDE* Eclipse plug-in also considers methods provided through implicit conversions, but only if the according conversion is in scope. Although, code completion has its shortcomings: The heavy bias towards member methods and the limited filter capabilities. Our API retrieval model supports more specific queries and is not limited to member methods. Extensions to code completion are the "Method Recommendation" and "Object Construction" features introduced in [2]. The former lists methods that accepts arguments of the type in question and the later finds call chains that create a value of a certain type.

Signature matching describes techniques that aim to solve the same problem as described in this paper. Although, most previous work on this topic targets languages not providing class-based inheritance [8, 13, 20] or use signature matching for refining results retrieved by latent semantic analysis [19]. Altogether, we did not find any previous work that addresses signature matching for type systems similar to $Ob_{\omega<:\mu}$ which integrates class-based inheritance with higher-order parametric polymorphism.

The approach that offers the functionality closest to our work is probably [12]. This tool uses partial expressions with "holes" as search queries for C# APIs and retrieves expressions matching these holes. For some of these partial expression, the search problem is identical to API retrieval as discussed in this paper but the approach does not exclusively focus on signature matching. Also, the paper does not cover parametric polymorphism.

The problem addressed by *code synthesis tools* can be seen as a superset of the API retrieval problem. Instead of retrieving definitions that can be reused to solve a common task, code synthesis tools generate code snippets spanning multiple statements. *InSynth* [5] suggests well-typed expressions that can be inserted at a certain program point based on an expected type and the definitions in scope. The programmer does not need to provide further information besides the current program context, but the code synthesis only considers local and imported declarations. *Prospector* [6] on the other hand, considers all declarations in a project, but the query is restricted to a single input and a single output type.

Finally, *code snippet retrieval* addresses a similar problem as code synthesis tools but use existing code to mine for code snippets that can be retrieved. The query may be a free-text formulation of a task [1] or a combination of a free-text description and types from the current context [18]. A major difference to our approach is that code snippets retrieval aims to find example code that demonstrates the usage of an API and API retrieval aims to retrieve reusable abstractions.

8. Conclusion

This paper shows that API retrieval can be adopted to object-oriented languages with complex type hierarchies by using a term based model. Our model can be used to quickly answer information needs that would be hard to answer using code completion.

Future work includes the integration with IDE that enables developer to easily index Scala projects including dependencies and issue queries from within the code editor. Also, a detailed user study to assess the impact of API retrieval on developer's daily work would be useful. Furthermore, some aspects of Scala are not yet fully incorporated into the model. Especially retrieving functionality provided through the type class pattern [10] requires some further work. And finally, the retrieval model could be adapted to languages with type systems similar to Scala's, for example, Java or C#.

References

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

[2] Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6813 LNCS:79–104, 2011.

[3] Dominic Duggan and Adriana Compagnoni. Subtyping for Object Type Constructors. In *In FOOL 6. Foundations of Object-Oriented Languages*, 1999.

[4] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. *Proceedings - International Conference on Software Engineering*, pages 302–311, 2007.

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

[6] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. *ACM SIGPLAN Notices*, 2005.

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

[8] Neil Mitchell. Hoogle: Finding Functions from Types, 2011.

[9] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. *ACM SIGPLAN Notices*, 43(10):423, 2008.

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

[11] Martin Odersky and Matthias Zenger. Scalable component abstractions. *ACM SIGPLAN Notices*, 40(10):41, oct 2005.

[12] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*, page 275, 2012.

[13] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, pages 174–183, 1991.

[14] Gerard Salton, Andrew Wong, and Chungshu Yang. A vector space model for automatic indexing. In *Communications of the ACM*, volume 18, pages 613–620, 1975.

[15] Mirko Stocker. Scala Refactoring. Technical report, University of Applied Sciences Rapperswil, 2010.

[16] Jeffrey Stylos and Brad a. Myers. The implications of method placement on API learnability. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, page 105, 2008.

[17] Lukas Wegmann. Scaps: Type-directed API Search for Scala. Technical report, University of Applied Sciences Rapperswil, 2015.

[18] Yi Wei, Nirupama Chandrasekaran, Sumit Gulwani, and Youssef Hamadi. Building Bing Developer Assistant. Technical report, Microsoft Research; MSR-TR-2015-36, 2015.

[19] Yunwen Ye and Gerhard Fischer. Supporting reuse by delivering task-relevant and personalized information. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 513–523, 2002.

[20] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: a Tool for Using Software Libraries. *Transactions on Software Engineering and Methodology*, 4(April):146–170, 1995.