Fine-grained Analysis of Similar Code Snippets

Jessie Galasso, Michalis Famelis, and Houari Sahraoui

Université de Montréal, DIRO, Canada
{jessie.galasso-carbonnel,michalis.famelis,houari.sahraoui}@umontreal.ca

Abstract. Code recommendation aims to help programmers in their coding endeavors by suggesting appropriate code snippets to complete their program. Code recommendation approaches such as code search or code repair may rely on code snippets or code templates extracted from existing projects to provide these suggestions. In this context, extracting and characterizing reusable and recurring code structures beforehand is thus essential. In this paper, we characterize recurring code structures through parametrizable code templates. Code templates can outline the common structure in code snippets along with their variation points, hence providing a convenient way to define their structural similarity. Pattern Structure is a mathematical data analysis framework for organizing objects depending on their similarity: it produces a structure supporting clustering, analysis, and knowledge discovery tasks. We propose an approach leveraging this framework and similarity defined through code templates to highlight and organize groups of similar snippets. The produced structure contains all relevant code templates as well as refinement relationships between them, and can be used to support both manual and automated analysis. We present a case study where we apply this approach to analyze snippets for the task of code sophistication, which consists of identifying and suggesting missing conditional paths in programs.

Keywords: Code recommendation · Code template · Code similarity

1 Introduction

Code recommendation covers a set of tasks aiming at suggesting relevant code snippets to programmers to assist their coding endeavors. It includes suggesting the most likely next tokens (code auto-completion [9]), patches to fix a defect (automated program repair [13]) or examples of similar snippets to the one under development (code search [10]). Even though they do not share the same goal nor use the same approaches, these tasks have in common to rely on existing code bases to extract reusable knowledge about which code snippets to suggest and when. Some approaches, notably in code search and program repair, are founded on the reuse of code snippets or more generic code structures (i.e., abstract representations of the code) identified beforehand to perform their recommendations.

In this paper, we address the problem of characterizing and extracting the recurring structures in a set of code snippets (e.g., Listing 1.1) to foster reuse for code recommendation approaches. We use parametrizable code templates [5] as a mean to characterize recurring code structures. A code template is a code snippet with placeholders (missing code parts), which can be replaced with concrete values to instantiate a concrete code snippet. In other words, it is a parametrizable code snippet, representing all concrete code snippets which could be instantiated from it. We show how templates can serve as a mean to formalize structural similarity between snippets by outlining their common code parts along with their variation points. For instance, Figure 2 (right-hand side) presents a template with one placeholder denoted $\langle ? \rangle$. Replacing $\langle ? \rangle$ by `0` leads to the snippet S_1 in Listing 1.1, and by `None` leads to S_2: this template thus outlines a common structure between these two snippets. Templates are widely used to capture code idioms to be reused by programmers in IDEs [11].

We propose an approach to assist developers in identifying important code templates to characterize recurring code structures in a given set of code snippets. This approach relies on Pattern Structure [7], a data analysis framework for knowledge extraction and representation. Following the framework, we first create clusters of code snippets sharing structural properties, and characterize each cluster with a template representing the structural similarity of its contents. Then, we organize these clusters in a hierarchy which reflects relations of specialization and generalization between templates. The obtained hierarchical structure naturally emphasizes what is common and what varies between clusters of similar code snippets, at several levels of increasing detail, which enables fine-grained analysis of recurring code structures. Pattern Structure guarantees that we uncover all relevant templates in the considered set of snippets and organizes them in a way that support exploration and knowledge discovery. We implemented a prototype to extract and browse the relevant templates of a given set of fragments. We present a case study about analyzing snippets for code sophistication, a code recommendation task aiming at suggesting missing behaviors in a program.

This paper is organized as follows. We discuss the characterization of similar code snippets with templates in Section 2. Section 3 shows how to leverage these templates to produce a hierarchical clustering of similar snippets with Pattern Structure. In Section 4, we discuss how the properties of this hierarchical structure are useful for analyzing and reusing code templates, and present four use cases. Section 5 presents the results of applying this approach on sophistication snippets to detect recurring code structures in missing conditional paths. Related work is presented in Section 6 and Section 7 concludes this paper.

(S_1)

```
if len(x) == 0:
    return 0
```

(S_2)

```
if len(x) == 0:
    return None
```

(S_3)

```
if len(x) == 0:
    continue
```

(S_4)

```
if len(x) == 0:
    return y.values[0]
```

(S_5)

```
if x.values[0] == 0:
    return None
```

Listing 1.1. Similar code snippets

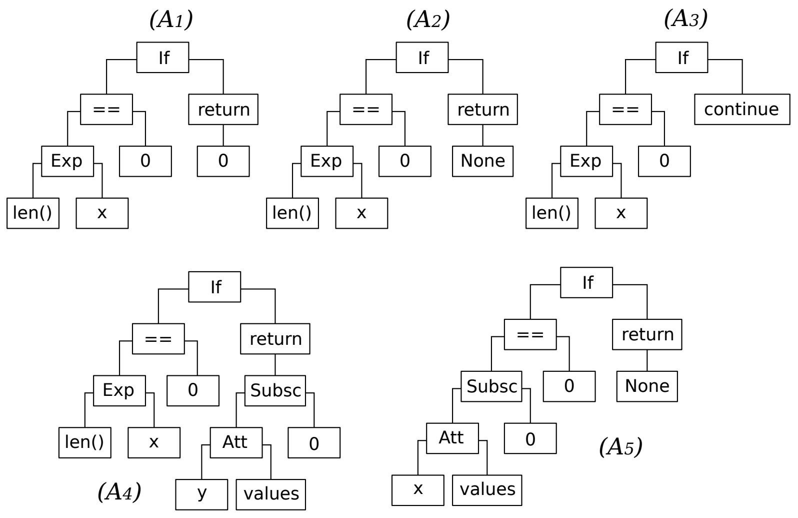


Fig. 1. Simplified ASTs of the code snippets from Listing 1.1

2 Characterizing Similar Code Snippets with Templates

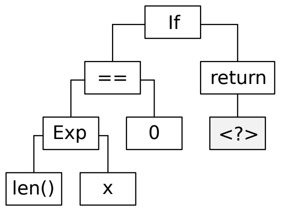
To motivate our approach, in Listing 1.1 we introduce 5 snippets corresponding to conditional paths written in Python. Figure 1 shows their corresponding abstract syntax trees (ASTs). All 5 snippets share a common structure, i.e., a conditional check, followed by a return statement, but differ on the checked condition and the return value.

2.1 Structural Code Templates

A code template is a code snippet with *"holes"* or *placeholders* that usually replace identifiers (class, method or variable names) or literals [5]. Replacing the placeholders of a template by concrete values generates a concrete code snippet. We say that the obtained snippet *instantiates* the template, or that the template *matches* the snippet. Placeholders can eventually be typed to restrict the values which can replace them.

Code snippets which only differ in their identifiers and literals are said to be *lexically similar*. They possess the same AST structure but may differ at the leaf nodes. For instance, snippets S_1 and S_2 (Listing 1.1) are lexically similar: their corresponding ASTs A_1 and A_2 have the same structure (i.e., the same internal nodes) and only the leaves under their `return` node differ. A template thus has an AST in which at least one leaf is a special placeholder node. Templates are lexical abstractions of all code snippets that have the same structure but have different identifiers and literals. Such templates are called *lexical templates*. Figure 2 shows a template with one placeholder (denoted $\langle ? \rangle$) that is a lexical abstraction of snippets S_1 and S_2.

Evans et al. [5] generalized lexical abstractions such that template placeholders may replace not only a leaf but any sub-tree of the AST. Placeholders then still correspond to leaves in the AST of the template (they cannot be internal nodes), but can now be



```
if len(x) == 0:
  return <?>
```

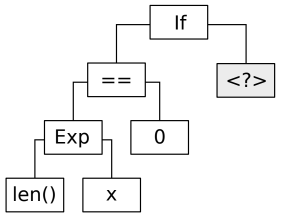
Fig. 2. Template with one placeholder, representing a conditional check and a return statement

replaced by complex expressions. In this more general case, we can substitute the placeholder in Figure 2 by the sub-tree of the expression `y.values[0]`, obtaining the snippet S_4. Evans et al. called this generalization *structural abstraction*, as code snippets instantiated from these templates may have different AST structures: they are thus *structural templates*. In our example, the template of Figure 2 matches snippets S_1 and S_2, which are lexically similar, but also S_4 even though it is structurally different than the other two. In this paper, we use structural templates (henceforth, simply "templates") to outline the similar code parts of a set of snippets and indicate where they differ and how.

The same set of snippets can be structurally abstracted by more than one template. For instance, both templates of Figure 2 and Figure 3 match S_1, S_2 and S_4. In what follows, we focus on using one particular template matching a given set of snippets, which is the one characterizing their maximal similarity (i.e., the least abstract template) with the smallest number of variation points (i.e., placeholders). We call it the *common template* of the set of snippets. The AST of the common template corresponds to the largest common AST of the considered snippets, with the smaller number of placeholders needed to instantiate these snippets.

Definition 1 (Common template AST). *The common template AST of a given set of ASTs corresponds to the AST containing all nodes whose root path (path between the node and the root) are present in all given ASTs. The common template AST also contains placeholders indicating the first occurrence of a different node in a downward path from the root to the leaves.*

The template of Figure 3 does not represent the common template of $\{S_1, S_2, S_4\}$, because the `return` node that is common to the three corresponding ASTs is not included in the template. Rather, their common template is the one presented in Fig-



```
if len(x) == 0:
  <?>
```

Fig. 3. Template with one placeholder, representing a conditional check

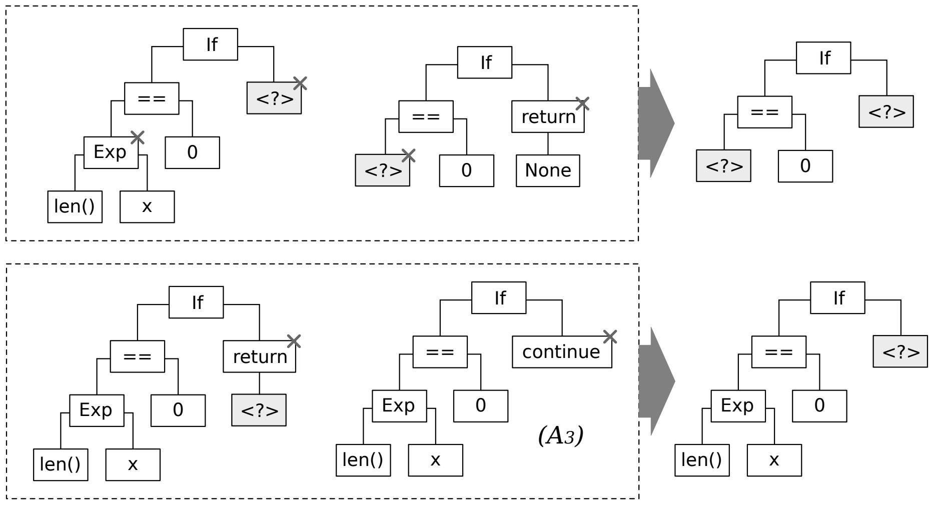


Fig. 4. Similarity templates between two templates (top) and between a snippet and a template (bottom). Crosses show the closest nodes to the root that are not common in the two ASTs, which will correspond to a placeholder in the common template.

ure 2. On the other hand, the template in Figure 3 is the common template of the set $\{S_1, S_2, S_3, S_4\}$, given that S_3 does not have a `return` node.

The common template of a set of snippets characterizes their similarity in terms of their shared AST structure. The size of the common template may hint at how similar the snippets are: the bigger the common template, the more structural similarity they share. In the basic case where we have two or more syntactically equivalent snippets, the common template is a template without placeholders, that cannot be instantiated, i.e., a snippet.

2.2 Similarity Between Templates

As templates are generalized ASTs that can potentially have special leaf nodes to represent placeholders, we can extend to them the notion of similarity based on common AST nodes. Following Definition 1, given a set of templates, we can extract a common template. We show the ASTs of two templates and their common template in Figure 4 (top). Divergent nodes, yielding to placeholders in the common template, are marked in the figure with an X. Extraction of common templates can also be applied to mixed sets that contain both templates and snippets. We show in Figure 4 (bottom) the similarity template extracted between A_3 (the AST of snippet S_3) and the template of Figure 2.

We explained previously that snippets can be instantiated from their common template by replacing a placeholder with a concrete AST sub-tree. In a similar way, a template can be instantiated from another template by *specializing* a placeholder, i.e., replacing it by a sub-tree which may itself contain placeholders. This operation amounts to template *refinement*. Instantiating a concrete code snippet from a template can thus be seen as a process of step-wise template refinement, each step further specializing a placeholder until all placeholders are replaced by concrete code.

Next, we present a mathematical framework which leverages common templates and template refinement to organize snippets and templates by similarity. We then show how to use it for fine-grained analysis and extraction of recurring code structures.

3 Structuring Templates by Similarity with Pattern Structure

Pattern Structure [7] is a structural framework for data analysis and knowledge discovery and representation of a set of *objects* characterized by *descriptions*. It enables building canonical hierarchies of specialization which organize objects depending on the *similarity* of their descriptions. In this section, we present the key definitions of this framework, and show how we apply it to organize templates by similarity.

Input. As input, the framework considers a triple $(O,(D,\sqcap),\delta)$, where O is a finite set of objects, D is a set of descriptions and $\delta : O \to D$ maps each object to its description. The set of descriptions in D can be of any type and must be associated with a *similarity operation* (denoted $\sqcap$). When applied on a set of descriptions from D, this similarity operation must return another description from D representing the similarity, or generalization of its arguments. The similarity operation is idempotent, commutative and associative, and is associated with a subsumption relation $\sqsubseteq$ providing the descriptions in D with a specialization/generalization order:

$$\forall d_1, d_2 \in D, d_1 \sqsubseteq d_2 \Leftrightarrow d_1 \sqcap d_2 = d_1$$

Thus, $(D,\sqcap)$ forms a meet-semilattice, i.e., a structure in which each subset of descriptions from D has an upper bound, namely the element in D that describes their similarity.

In this paper, we consider that the objects in O are code snippets (e.g., the one in Listing 1.1). As objects' descriptions, we consider their ASTs: D thus includes the ASTs of the snippets in O (without placeholders, as in Figure 1) and δ maps the snippets in O to their corresponding ASTs in D. Based on the definitions above, the extraction of the common template AST from a set of ASTs corresponds to the similarity operation $\sqcap$. Thus, the set of descriptions D includes *(i)* the ASTs of the snippets in O and *(ii)* all the common template ASTs that can be computed from them. For instance, if the snippets in O are the ones from Listing 1.1, then A_2 and A_3 are included in D. Their similarity ($A_2 \sqcap A_3$) is characterized by the AST $A_{2,3}$ (corresponding to Figure 3), which is thus also in D. In this way, $(D,\sqcap)$ forms a meet-semilattice, in which each subset of ASTs has an upper-bound, namely the AST characterizing their structural similarity. Figure 5 shows an example meet-semilattice based on the three ASTs A_2, A_3 and A_5.

The subsumption relation associated with this similarity operation here organizes the ASTs in D by specialization/generalization: if $A_2 \sqcap A_3 = A_{2,3}$, then $A_{2,3} \sqsubseteq A_2$ and $A_{2,3} \sqsubseteq A_3$. It corresponds to the notion of template refinement discussed in Section 2.2: $A_{2,3}$ is a more generic template than A_2 and A_3, and thus A_2 and A_3 can be instantiated from $A_{2,3}$. In Figure 5, an edge between two ASTs shows the subsumption relation: a template subsumes another template if it can be instantiated from it. The higher a template is placed in the hierarchy, the more abstract it is (i.e., the fewer concrete nodes

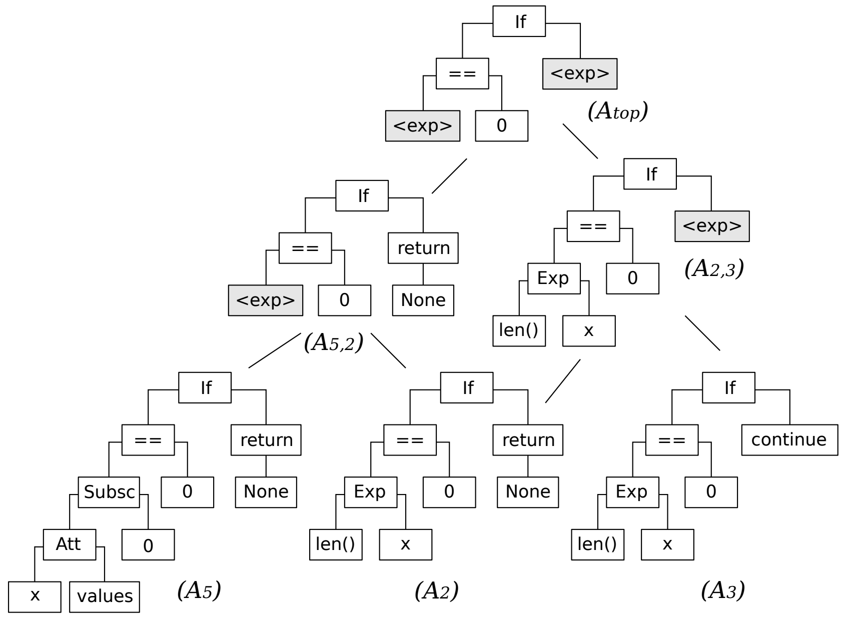


Fig. 5. Excerpt of meet-semilattice of templates. The lowest upper bound of any subset of elements shows their common template

it has). In Figure 5, the template of level 3 (top) represents the similarity of the templates of level 2, and corresponds to the common template of the ASTs of level 1 (bottom).

Pattern concepts. Given a triple $(O,(D,\sqcap),\delta)$, the framework first extracts a finite set of *pattern concepts*, denoted C_K. The extraction process relies on two derivation operators: $\alpha : 2^O \mapsto D$ and $\beta : D \mapsto 2^O$. The operator α associates with a subset of objects $O' \in O$ the most specific description of $(D,\sqcap)$ matching all objects of O'. The operator β associates each description $d \in (D,\sqcap)$ with all objects from O matching it.

$$\alpha(O') = \prod_{o \in O'} \delta(o), \text{ with } O' \in O$$
$$\beta(d) = \{o \in O \mid d \sqsubseteq \delta(o)\}, \text{ with } d \in (D,\sqcap)$$

A pattern concept is a pair (O',d), $O' \in O$, $d \in (D,\sqcap)$, such that $\alpha(O') = d$ and $\beta(d) = O'$. C_K is obtained by applying these two operators on all subsets of O. Each pattern concept represents a maximal group that guarantees that the description d is the most specialized description that describes a given set of objects O', and that there are no objects outside O' that are described by d.

In our example, a pattern concept takes the form of a group of snippets associated with a template. The operator α associates with a set of snippets O' the common template of their ASTs. For instance, $\alpha(\{S_1,S_2,S_3\}) = A_{1,2,3,4}$, where $A_{1,2,3,4}$ is the template presented in Figure 3. The operator β associates a template d with all snippets matching it. Thus, $\beta(A_{1,2,3,4}) = \{S_1,S_2,S_3,S_4\}$. The pair $(\{S_1,S_2,S_3,S_4\}, A_{1,2,3,4})$ is thus a pattern concept, shown as C_{1234} in Figure 6. From the five snippets of List-

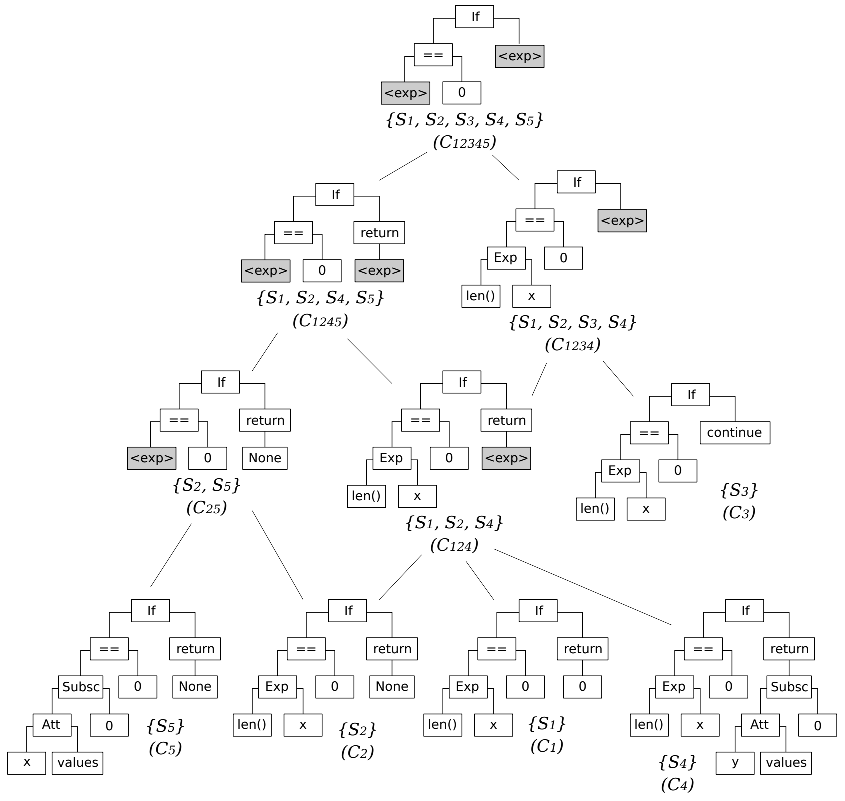


Fig. 6. Pattern concept lattice based on common templates (the bottom concept is omitted because it does not provide any useful information)

ing 1.1, a total of 11 pattern concepts can be extracted: Figure 6 shows 10 of them and omits the trivial pattern concept representing the empty set of snippets.

Specialization order. Finally, the framework provides the set of pattern concepts C_K with a partial order $\leq$. This order is based on the subsumption relation $\sqsubseteq$ which exists between the descriptions of each pattern concept. Given two concepts $C_a = (O_a, d_a)$ and $C_b = (O_b, d_b)$, $C_a \leq C_b$ iff $O_a \subseteq O_b$ and $d_b \sqsubseteq d_a$. C_a is then called a sub-concept of C_b, and C_b a super-concept of C_a. The set of concepts provided with the partial order forms a lattice structure $(C_K, \leq)$ organizing the concepts in a specialization hierarchy.

In Figure 6, the specialization order is represented by the edges. If $A_{1,2,4}$ is the template from Figure 2 that corresponds to the concept C_{124} in Figure 6, then $C_{124} \leq C_{1234}$ because $\{S_1, S_2, S_4\} \subseteq \{S_1, S_2, S_3, S_4\}$ and $A_{1234} \sqsubseteq A_{124}$ (i.e., A_{124} specializes A_{1234}).

4 Discussion

In this section, we discuss the advantages of structuring templates with the Pattern Structure framework and present some use cases.

4.1 Leveraging Pattern Structures Properties

We identified three properties of pattern concept lattices that useful for the analysis and selection of recurring code structures.

The pattern concept lattice is canonical. In other words, only one pattern concept lattice can be built for a given input. The building process is deterministic and does not depend on any parameters. This property guarantees a sound and complete template identification: all maximal templates are represented in the concepts, and there is no concept that does not represent a maximal template. This ensures that all potentially relevant templates are included in the structure.

The partial order provides a specialization hierarchy. Given a concept and the template it represents, its super-concepts show the more abstract templates from which it can be instantiated. In turn, its sub-concepts show the more specific templates that can be instantiated from it. In Figure 6, C_{25}, C_{124} and C_2 are examples of sub-concepts of C_{1245}, and C_{12345} is the only super-concept of C_{1245}. Thus, the concepts at the top of the hierarchy cover large groups of snippets characterized by generic templates. Going down the hierarchy reveals more specific templates corresponding to smaller groups of snippets. The structure depicts the templates of subsets of snippets at different level of granularity, thus enabling to explore recurring templates at increasing levels of detail.

The neighborhood of a concept shows its most similar concepts. The direct sub-concepts of a given concept show the most similar descriptions which are a refinement of its description. While its sub-concepts represent all possible refinements of its associated template, the direct sub-concepts show the possible *minimal* refinements. In our example, the two direct sub-concepts of C_{1245} are C_{25} and C_{124}. C_5 is also its sub-concept but not a direct one. C_{25}, C_{124} and C_5 can therefore all be instantiated from C_{1245}, but only C_{25} and C_{124} are minimal specializations. C_5 is not a minimal specialization as it requires two specialization steps from C_{1245} (first to C_{25} and then to C_5). The same logic applies for the direct super-concepts: we can explore the closest generalizations of a given template by inspecting the super-concepts of its concept. This allows the stepwise exploration of the specializations of a template with only minimal steps. In other words, following a chain from top to bottom corresponds to a stepwise template refinement process. If a template is too abstract to be relevant for reuse, we can analyze the templates of its direct sub-concepts, because they show a way to divide the current group of similar snippets in smaller yet cohesive groups.

4.2 Use cases

Below we describe how the concept lattice can be used to improve code completion.

Extracting templates. Concepts that represent maximal groups of similar snippets and their associated template provide a strong foundation to identify recurring code structures. The pattern concept lattice shows templates' occurrences in a given set of code snippets. This can help define fine-grained metrics to filter a large set and alleviate the effort of manual analysis. The identified templates can be used to better understand recurring code structures in a code base and to define reusable and parametrizable templates. They can help find coding idioms in a given project, or find recurring structures across projects (e.g., API usages).

Exploring the space of templates. Exploratory search [16] is an information retrieval strategy aimed at guiding a user into progressively discovering a set of documents by navigating (or browsing) into a structured space of these documents. Conceptual structures are an advantageous support for exploration [8], notably because the direct neighbors of a concept represent concepts with the most similar descriptions and thus guarantee an exploration process with minimal steps. In our case, it enables navigating from one maximal relevant template to another by following their specialization/generalization relations. This could help discovery and analysis of templates through exploration.

Supporting code recommendation. This way of organizing templates can help improve code completion and code search approaches. Code completion seeks to predict the next tokens a programmer would write at a specific location in the code being written. Recent work targets complex suggestions such as sequences of tokens [9], or even structured code snippets such as API usages [20]. Code search is the task of finding code examples to help the developer implement what they have in mind. Code search approaches usually rely on information retrieval to find in a code database those snippets that correspond to a query; the query is either formulated by the programmer or inferred automatically from the code under development [10]. Our proposed structuring of code templates can help both these tasks. These templates could be used to help assess the pertinence of suggested code, or to recommend similar snippets to the suggested one. We also envision suggesting not only fully formed snippets but relevant also parametrizable templates directly to the programmer. In this case, structured templates could help suggest potential refinements, or help the programmer explore a space of similar and relevant templates. In code search, queried snippets could be reorganized by similarity to simplify the output presented to the user by grouping similar snippets.

5 Case Study: Reusable Templates for Code Sophistication

Behavioral program analysis considers a program mainly as a collection of behaviors, i.e., sets of actions to be performed under certain conditions. Each behavior is an execution path for a given scenario (i.e., combination of input values) that is enabled by path conditions in the body of the program. Unusual combinations of input values may correspond to atypical scenarios which need to be handled by special conditional paths. However, developers are prone to omit such atypical scenarios from specification documents: missing conditional paths are known to be hard to detect [3] and are one of the largest source of bugs [22]. *Code sophistication* [6] is a type of code recommendation that suggests missing conditional paths in the form of *if*-blocks to handle these atypical scenarios. Previous work on missing conditional paths suggests that they recurrently address some typical behavior scenarios, such as error handling, variable assignments, or early returns. In order to produce good sophistication recommendations, we need to better understand and characterize the kinds of conditional paths that are usually missing. We thus focus on code commits where developers added *if*-blocks to add some previously missing behavior corresponding to an unaddressed scenario. By analyzing recurring code structures in such added *if*-blocks, we can better understand and characterize what the kinds of conditional paths that are usually missing from programs and thus produce better sophistication recommendations.

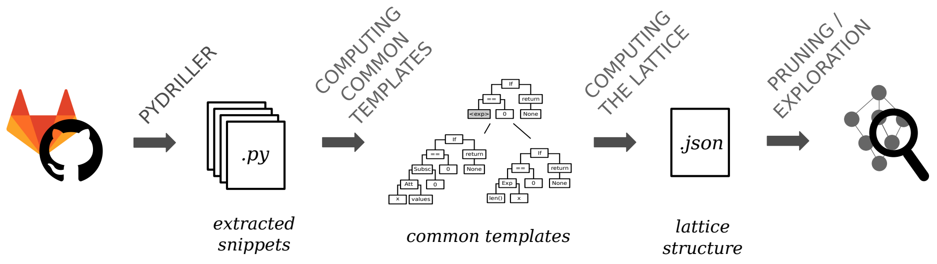


Fig. 7. Process to support the identification of recurring code templates

In this section we try to answer the question: *How can the pattern structure be used to comprehensively explore a set of snippets and identify relevant templates?* We present how we applied our approach to discover and explore recurring code templates in code snippets that represent missing conditional paths. The process is shown in Figure 7. First we extracted a dataset by gathering code snippets that represent missing conditional paths (Section 5.1). Then, we computed common templates and the lattice structure (Section 5.2) based on the extracted snippets. Finally, we show how to support the identification of relevant recurring templates (Section5.3). This is the first step towards building code recommendation tools for code sophistication.

5.1 Dataset

The version history of programs can reveal valuable insights about how programmers address defects in programs. We decided to focus on commits where a developer added conditional paths (*if*-blocks) to the code. Such a code change indicates that the pre-existing code version omitted some important conditional path which the developer tried to include by adding the missing behaviour [22].

We analyzed the version histories of 290 Git repositories from GitHub and GitLab. We used the advanced search of both platforms to select repositories containing code written in Python. We then analyzed all the commits of a repository to find changes in methods according to the following criteria: *a)* no code is deleted by the commit, *b)* the added lines are consecutive (a single snippet is added), and *c)* the added snippet corresponds to an `if` block. This filtering approach allows us to identify changes that focus on adding an *if*-block in a method. For each changed method, we isolated the code snippet added by the commit and stored it in a file. In total, we analyzed more than 2 million commits and extracted 25 000 methods where a conditional path was added by a developer. The added conditional paths amounted to between 1 and 75 lines of code, with an average of 1.9 lines per snippet (plus 1 line with the *if* statement and the condition). The snippets in Listing 1.1 were inspired from snippets in this dataset.

5.2 Implementation

We then apply the proposed approach on the extracted *if*-blocks to characterize recurring code structures in missing conditional paths. We implemented a two-step procedure. First, we compute the common template of a given set of ASTs. Then, we compute the lattice structure by relying on the common template extraction.

Computing the common templates. We used the AST Python library[1] to build the AST of each code snippet. This library provides a parser and an implementation of the Visitor design pattern for handling the nodes of an AST. So, starting from the root node, we traverse pairs of ASTs in parallel to build a third AST that represents their common template. If two nodes have the same type and the same values, we add a copy of them to the common AST and continue with the children nodes. If two nodes are of the same type but have different values (e.g., they are both binary operations, albeit different ones), we add to the common AST a node of this type but with no values (in this case, an empty node of type AST.BinOp). If the two nodes are of different types, we add to the AST an empty node representing an expression. In the two last cases, we interrupt the traversal and do not visit the children nodes of the divergent nodes. The empty nodes act as placeholders ("holes") and correspond to the most specific type that characterizes the two divergent nodes. This produces a common template AST that can itself be traversed during the computation of subsequent template ASTs.

Computing the lattice structure. To extract the maximal templates, we applied the common template extraction on all code snippets in the dataset. After computing the AST of each snippet, we obtained 17 166 unique ASTs, meaning that a third of the extracted snippets were syntactically equivalent. Our method first computes the common ASTs of each pair of these initial ASTs. Then, it computes the common ASTs of each pair of the common ASTs produced in the previous step. This repeats until no new AST is created. After computing the similarity between the initial 17 166 ASTs, we obtained 52 241 new ASTs corresponding to templates. In the next iteration, we generated 8 421 more ASTs, and in the next, 5 more ASTs. After this fourth iteration, no new AST was generated. During this process, we kept track of which pairs of ASTs generated which templates. In this way, we are able to infer the specialization relationships and thus the partial order between templates associated with the obtained ASTs. We stored all ASTs, the related snippets and the partial order in a JSON file.

One disadvantage of Pattern Structures is the size of the structure and the cost involved in its computation. For a Pattern Structure $(O,(D,\sqcap),\delta)$, the associated lattice structure may have a number of concepts up to $2^{min(|O|,|D|)}$, even though it rarely reaches this upper-bound. In our case study, we obtained a total of 77 828 concepts for both the concrete code snippets and the extracted templates. However, not all templates are of interest, especially for our application scenario, where the aim is to focus on the ones representing recurring structures. As discussed earlier, such templates are usually situated at the top of the structure. This allows pruning the structure to reduce its size. For instance, in the presented study, if we choose to include only templates matching at least 200 snippets from the original dataset, we obtain a more manageable structure of 211 concepts. The cost of the lattice computation can be significant, especially in our case, as similarity is computed by tree comparison. With small optimizations, our implementation took about 5 hours to generate the structure from the 25 000 snippets. Further optimizing the generation process is an interesting avenue for future work.

[1] `https://docs.python.org/3/library/ast.html`

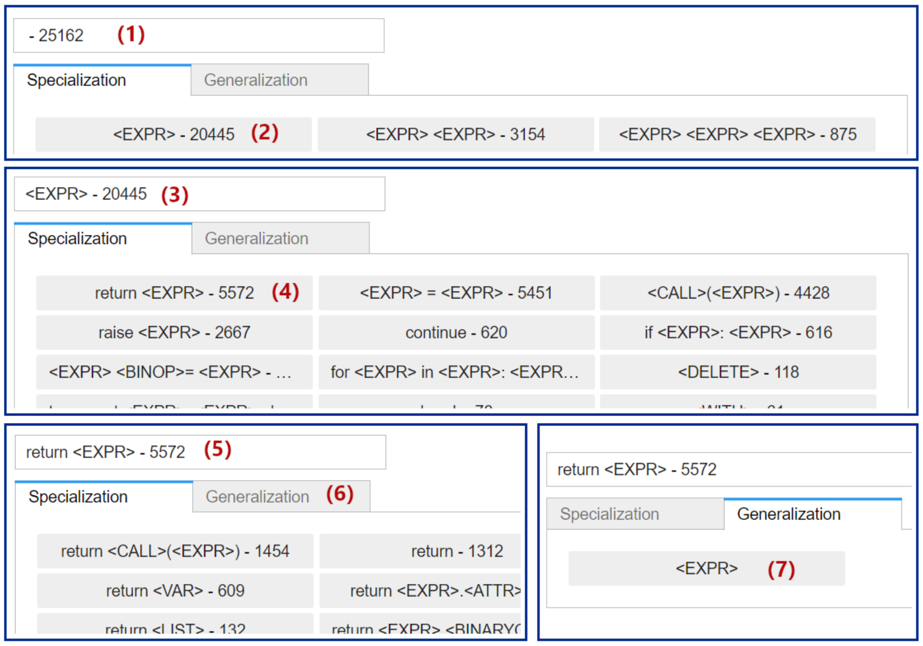


Fig. 8. Screenshots of the navigation interface

5.3 Analysis by Exploration

As described in the previous section, we can prune the structure to only include templates representing many snippets, and perform a manual analysis on this smaller number of templates. However, without leveraging the refinement relations between the templates, this manual analysis would be tedious and error prone. To enable a step by step analysis of the space of templates, we implemented a GUI prototype[2] to perform exploratory search based on the structure stored in the JSON file created in Section 5.2. We show screenshots of the navigation interface in Figure 8.

Given a template on which we want to focus as the basis for exploration (the "focus template"), the interface can display its direct generalizations and direct specializations. The focus template is displayed in the box at the top (e.g., 1, 3 and 5). Below we show two tabs: one for its direct template specializations, and the other for its generalizations. Each tab shows a grid with buttons for the relevant templates. Clicking on a template button changes the focus template to the new template and updates the display to show its own generalizations and specializations. Each template is followed by a number representing its occurrence. In Figure 8 (4), "`return` ⟨EXPR⟩ `- 5 572`" states that 5 572 concrete code snippets of the dataset match the template "`return` ⟨EXPR⟩".

The navigation starts with the common template of all snippets, which in our study happened to be empty. Figure 8 (1) shows the empty template being the focus template,

[2] `https://figshare.com/s/ed329beeeeb27c985632`

which is matched by all of the 25 162 snippets. Snippets with different numbers of lines are not comparable with the current state of the approach, so the direct specializations tab of the empty template shows groups of snippets depending on their number of lines. We can see that out of 25 162 snippets, 20 445 have one line (they match the template "$\langle$`EXPR`$\rangle$"), 3 154 have two lines and 875 have three. If we click on the template representing all snippets with one line (2), it becomes the focus template (3). We can see now that most one-line snippets in the studied dataset return an expression, perform an assignment or call a function or a method. To know which kinds of expressions are returned the most, we can click on the corresponding template (4) which becomes the new focus template (5). It shows that among the snippets returning an expression, most of them return a function/method call, nothing, or a constant (not shown in the figure). If we select the *Generalization* tab (6), we can see that the only direct generalization of `return` $\langle$`EXPR`$\rangle$ is $\langle$`EXPR`$\rangle$. Clicking on (7) will make us come back to (3).

To sum up, we tried two approaches to analyze the recurring templates in a set of snippets organized with pattern structures. The first one consisted in pruning the structure using a threshold to retain only the concepts at the top, representing the templates matched by the most snippets of the original dataset. This approach helps to highly reduce the size of structure and thus its computation time. However, analyzing the retained set of templates as is can be tedious. The second approach leverages the refinement relations between the concepts to create an interface for exploring local areas of the structure step by step. It enables to explore the whole structure in a rather intelligible way without pruning it. Combining both approaches to be able to try different thresholds and create small explorable sets of templates is left for future work.

6 Related Work

Several researchers have worked on detecting groups of similar code snippets based on their ASTs. Kontogiannis et al. [14] and Jiang et al. [12] identified clusters of similar snippets based on feature vectors of structural program features, including features based on the AST. Other methods focus on representing the ASTs as sequences to find recurring subsequences. Koschke et al. [15] proposed to linearize AST nodes by a preorder traversal to apply suffix tree clone detection. Suffix trees allow fast implementation of string search operations such as finding occurrences of patterns. Yang et al. [23] generated sequences from ASTs in which leaves were replaced by value types (lexical abstraction), and used the Smith-Waterman sequence alignment algorithm to compute similarity scores. Nichols et al. [21] proposed a hybrid approach combining a structural detection similar to the one of [23] and a nominal clone detection approach. Nominal clone detection is based on the idea that similar snippets will use similar identifiers.In comparison, our work relies on the AST structure to group similar snippets. Some authors work directly on the structure of ASTs. Merlo et al. [17] first identified different syntactic blocks (statements, methods, classes, etc) and their inclusion relationships for the AST of a project. Then, they created clusters of similar blocks to detect higher level structural similarity. Baxter et al. [2] found similar snippets by detecting recurring exact sub-trees in ASTs. They checked the parents of sequences of exact sub-trees to detect near-miss clones. In our work, we do not consider similarity between sub-trees if their parents differ: in this way, the extracted templates keep a consistent code structure which can be refined. In [4], Chodarev et al. used a pattern recognition algorithm to detect lexical templates. Narasimhan et al. [19] proposed a method to merge similar snippets as part of a refactoring task. In comparison, we do not provide an implementation of the merged similar snippets, but a structure showing the possible paths of parametrization of the common parts of the similar snippets.

Other works used templates to characterize and manage similar snippets. Evans et al. [5] first generated candidate templates from the AST: for each node, they took the full sub-tree rooted in this node and replaced descendants at several levels of depth by placeholders. Then, they retained templates occurring at least twice and applied an operation called *pattern improvement* which computes the largest template matching all its occurrences. Contrary to their work, we do not identify potential patterns in an AST, but we rely on their definition of structural templates and pattern improvement to characterize similarity between a given set of snippets. We then extend these notions to identify the set of all maximal relevant templates of a set of snippets and structure them by refinement. In the context of code repair, Bader et al. [1] proposed an approach to extract recurring edit patterns in the form of pairs of ASTs representing the code before and after the repair. They generalized pairs of edit patterns by computing common templates of the associated ASTs and organize them in a hierarchy represented by a dendrogram. Contrary to our approach, their building process performs approximations to increase computation efficiency, and does not guarantee the identification of all maximal templates. Molderez et al [18] addressed the problem of automatically generalizing or specializing a given template. They use a search-based approach relying on a set of concrete snippets which should be matched or not by the new template to identify a sequence of modifications to be applied on the current one. In our work, we organize all templates in a structure that emphasize all minimal modifications that need to be performed to obtain a template from another.

7 Conclusion and Future Work

We presented an approach for fine-grained analysis of the similarity of a set of snippets to support the identification and reuse of recurring code structures in the form of templates. We showed how templates can characterize the similarity of snippets, and how to produce a hierarchical clustering of similar snippets using the Pattern Structure framework. We applied this approach to detect relevant templates for code sophistication, and showed how the hierarchical structure we produced can support exploratory search to navigate the space of extracted templates.

In the future, we plan to investigate additional ways of capturing similarity. For example we could adapt the current similarity discovery operation between snippets to consider similarity between snippets that have different numbers of statements. We also want to improve the quality of extracted templates by pre-processing the snippets before assessing their similarity. We plan to extend our current approach to consider lexical information and improve the variability expressed by the extracted templates. Finally, we plan to evaluate different analysis approaches to determine the practicability of extracted templates for end-users.

References

1. Bader, J., Scott, A., Pradel, M., Chandra, S.: Getafix: Learning to fix bugs automatically. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–27 (2019)
2. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Int. Conf. on Software Maintenance. pp. 368–377. IEEE (1998)
3. Chen, T.H., Nagappan, M., Shihab, E., Hassan, A.E.: An empirical study of dormant bugs. In: 11th Working Conference on Mining Software Repositories. pp. 82–91 (2014)
4. Chodarev, S., Pietriková, E., Kollár, J.: Haskell clone detection using pattern comparing algorithm. In: Int. Conf. on Engineering of Modern Electric Systems. pp. 1–4. IEEE (2015)
5. Evans, W.S., Fraser, C.W., Ma, F.: Clone detection via structural abstraction. Software Quality Journal **17**(4), 309–330 (2009)
6. Galasso, J., Famelis, M., Sahraoui, H.A.: Code sophistication: From code recommendation to logic recommendation. CoRR **abs/2201.07674** (2022)
7. Ganter, B., Kuznetsov, S.O.: Pattern structures and their projections. In: Int. Conf. on Conceptual Structures. pp. 129–142. Springer (2001)
8. Godin, R., Pichet, C., Gecsei, J.: Design of a browsing interface for information retrieval. In: Int. Conf. on Research and development in information retrieval. pp. 32–39 (1989)
9. Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.: On the naturalness of software. Communications of the ACM **59**(5), 122–131 (2016)
10. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: 27th Int. Conf. on Software engineering. pp. 117–125 (2005)
11. Jacob, F., Tairas, R.: Code template inference using language models. In: Proceedings of the 48th Annual Southeast Regional Conference. pp. 1–6 (2010)
12. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: Int. Conf. on Software Engineering. pp. 96–105. IEEE (2007)
13. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: Int. Conf. on Software Engineering. pp. 802–811. IEEE (2013)
14. Kontogiannis, K.A., DeMori, R., Merlo, E., Galler, M., Bernstein, M.: Pattern matching for clone and concept detection. Automated Software Engineering **3**(1), 77–108 (1996)
15. Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: 2006 13th Working Conference on Reverse Engineering. pp. 253–262. IEEE (2006)
16. Marchionini, G.: Exploratory search: from finding to understanding. Communications of the ACM **49**(4), 41–46 (2006)
17. Merlo, E., Lavoie, T.: Computing structural types of clone syntactic blocks. In: 2009 16th Working Conference on Reverse Engineering. pp. 274–278. IEEE (2009)
18. Molderez, T., Roover, C.D.: Search-based generalization and refinement of code templates. In: Int. Symposium on Search Based Software Engineering. pp. 192–208. Springer (2016)
19. Narasimhan, K.: Clone merge–an eclipse plugin to abstract near-clone C++ methods. In: IEEE/ACM Int. Conf. on Automated Software Engineering. pp. 819–823. IEEE (2015)
20. Nguyen, A.T., Nguyen, T.N.: Graph-based statistical language model for code. In: 37th Int. Conf. on Software Engineering. vol. 1, pp. 858–868. IEEE (2015)
21. Nichols, L., Emre, M., Hardekopf, B.: Structural and nominal cross-language clone detection. In: Int. Conf. on Fundamental Approaches to Software Engineering. pp. 247–263 (2019)
22. Raghavan, S., Rohana, R., Leon, D., Podgurski, A., Augustine, V.: Dex: A semantic-graph differencing tool for studying changes in large code bases. In: 20th Int. Conf. on Software Maintenance. pp. 188–197. IEEE (2004)
23. Yang, Y., Ren, Z., Chen, X., Jiang, H.: Structural function based code clone detection using a new hybrid technique. In: Annual Computer Software and Applications Conference (COMPSAC). vol. 1, pp. 286–291. IEEE (2018)