1 Version Spaces Algorithms

1.1 Construction

Algorithm 1 describes the algorithm for constructing the version space given the input code snippet r, the set of path expressions E, the bounded length of path programs k, and the bounded number of set operations m.

Algorithm 1 Constructing Version Spaces

1: **function** ConstructVersionSpace(r, E, k, m)
2: $\sigma_s \leftarrow \{r\}$
3: $\Sigma_0 \leftarrow \{\sigma_s\}$
4: $\Sigma \leftarrow \Sigma_0$ ▷ Accumulated states
5: $\Pi \leftarrow \emptyset$ ▷ Accumulated edges
6: **for all** $i = 1, \ldots, k$ **do** ▷ Path programs within length k
7: $\Sigma_i \leftarrow \{[\![e]\!](\sigma) | \sigma \in \Sigma_{i-1}, e \in E\} - \Sigma$ ▷ New states found at the i-th iteration
8: **for all** $e \in E, \sigma \in \Sigma_{i-1}$ **do**
9: $\Pi(\langle\sigma\rangle, [\![e]\!](\sigma))$.Append($e$)
10: $\Sigma \leftarrow \Sigma \cup \Sigma_i$
11: $L \leftarrow \Sigma$
12: **for all** $i = 1, \ldots, m$ **do** ▷ Algebraic programs within m operators
13: $L' \leftarrow \emptyset$
14: **for all** distinct pairs $(\sigma_1, \sigma_2) \in L \times L \cup L \times (\Sigma - L)$ **do**
15: **for all** set operators op **do**
16: $\sigma' \leftarrow op(\sigma_1, \sigma_2)$
17: L'.Append(σ')
18: $\Pi(\langle\sigma_1, \sigma_2\rangle, \sigma')$.Append($op$)
19: $L \leftarrow L' - \Sigma$
20: $\Sigma \leftarrow \Sigma \cup L$
21: **return** $\langle\Sigma, \Pi, \sigma_s\rangle$

1.2 Enumeration

Algorithm 2 describes the recursive algorithm for enumerating the programs consistent with the example input σ_s and the example output σ_t from the version space Π.

Algorithm 2 Enumerating Programs from Version Spaces

1: **function** ENUMERATE(Π, σ_s, σ_t)
2: **if** $\sigma_t = \sigma_s$ **then**
3: **return** $\{\epsilon\}$ ▷ ϵ is the empty program
4: $\mathbb{P}_c \leftarrow \emptyset$ ▷ Programs evaluated to σ_t
5: **for all** $\langle \sigma \rangle \xrightarrow{E} \sigma_t \in \Pi, e \in E$ **do**
6: **for all** $P \in$ ENUMERATE(Π, σ_s, σ) **do**
7: $\mathbb{P}_c$.APPEND(P/e)
8: **for all** $\langle \sigma_1, \sigma_2 \rangle \xrightarrow{op} \sigma_t \in \Pi$ **do**
9: **for all** $P_1 \in$ ENUMERATE(Π, σ_s, σ_1), $P_2 \in$ ENUMERATE(Π, σ_s, σ_2) **do**
10: $\mathbb{P}_c$.APPEND($op(P_1, P_2)$)
11: **return** $\mathbb{P}_c$

2 Soundness & Bounded Completeness

Soundness. A synthesis algorithm is *sound* if all the synthesized programs satisfy the given input-output examples.

Theorem 2.1. *The synthesis algorithm in* EXCALIBUR *is sound.*

Proof. The synthesis algorithm of EXCALIBUR is based on version spaces. During the construction of version spaces (Section 1.1), each outgoing state is evaluated by the associated expressions/operators from the input states, so the transitions between the states are sound. Also, the paths to the example output σ_t from the example input σ_s must be sound. Since each synthesized program is enumerated from the paths in version spaces (Section 1.2), the programs are guaranteed to be sound. □

Bounded Completeness. A synthesis algorithm is *bounded-complete* if, for any program P^* that satisfies the input-output examples (i.e., $[\![P^*]\!](\sigma_s) = \sigma_t$), $P^* \in \mathbb{P}_c$ if P^* has a maximum path length k^* and the number of set operators is m^* and the bound of synthesis $k \geq k^*$ and $m \geq m^*$.

Theorem 2.2. *The synthesis algorithm in* EXCALIBUR *is bounded-complete.*

Proof. We show that Algorithm 2 is bounded-complete for any given output states, including the example output, by mathematical induction.

The base case is that the synthesized program is empty (both the path length and the number of set operators are 0) when the output state is σ_s, which is handled in lines 2–3 of Algorithm 2.

If all programs within $(k', 0)$ are included, all programs within $(k'+1, 0)$ $(k'+1 \leq k)$ are also included, which is shown by the loop in lines 5–7.

If all programs within (k', m') are included, all programs within $(k', m'+1)$ $(m'+1 \leq m)$ are also included, which is shown by the loop in lines 8–10. □

3 Wildcard Pattern

Definition 3.1. A wildcard pattern pt is a tree structure for matching ASTs. Suppose the user-provided example code snippet is defined on a set of vertices V, a wildcard pattern pt is defined on $V \cup \{*\}$, where `*` is a symbol for wildcard matching. A wildcard pattern pt is defined recursively:

- pt can be a terminal $v \in V$ and matches AST terminals with the name $v.sort$.
- pt can be a wildcard symbol `*` and matches any AST.
- pt can be a tree structure $v(pt_1, \ldots, pt_k)$, i.e., the root is a nonterminal $v \in V$, and the children are wildcard patterns $pt_1, \ldots, pt_k$. An AST v' is matched if $v.sort = v'.sort$ and v' has k ordered children that can be matched by $pt_1, \ldots, pt_k$, respectively.

Example 3.1. An example wildcard pattern is `expression(*,'+',*)`, where `expression` is a nonterminal and `'+'` is a terminal. This wildcard pattern matches any ASTs that are addition expressions.

Enumerating all possible wildcard patterns present in the user-given example code is difficult due to the excessive number of patterns. Consider that every subtree of an AST is a choice of being `*`, and these choices have diverse combinations.

Therefore, we propose to group wildcard patterns by their effects on the example code, i.e., the ASTs matched by patterns. For each group, we choose the most "explicit" pattern as the group's representative because an explicit pattern can be turned less explicit by replacing some subtrees with `*`. We refer to such representative patterns as *wildcard patterns modulo example*. In the following, we formulate explicitness and the most explicit pattern in each group.

Definition 3.2. We define a partial order $\preceq$ among wildcard patterns to compare their explicitness in matching ASTs. A pattern pt^a is more explicit than or equal to a pattern pt^b (denoted as $pt^a \preceq pt^b$) if all the ASTs that

can be matched by pt^a can also be matched by pt^b. $pt^a \preceq pt^b$ holds if and only if any of the following holds:

- $pt^b = *$.
- pt^a and pt^b are terminals with the same name.
- $pt^a = v^a(pt^a_1, \ldots, pt^a_k)$ and $pt^b = v^b(pt^b_1, \ldots, pt^b_k)$ have the same root nonterminals (i.e., $n^a.sort = n^b.sort$) and $pt^a_i \preceq pt^b_i$ for $i = 1, \ldots, k$.

Theorem 3.1. *The $\preceq$ relation on wildcard patterns is a partial order.*

Proof. First, $\preceq$ is reflexive:

- If $pt = *$, then $pt \preceq pt$ by definition.
- If pt is a terminal, then $pt \preceq pt$ by definition.
- Otherwise, $pt = v(pt_1, \ldots, pt_k)$, then $pt \preceq pt$ holds if and only if $pt_i \preceq pt_i$ for $i = 1, \ldots, k$, which is proved recursively.

Also, $\preceq$ is antisymmetric. Supposed $pt^a \preceq pt^b$ and $pt^b \preceq pt^a$.

- If $pt^a = *$ and $pt^a \preceq pt^b$, by definition, pt^b can only be $*$, so $pt^a = pt^b$.
- If pt^a and pt^b are terminals with the same name, $pt^a = pt^b$ obviously.
- Otherwise, $pt^a = v^a(pt^a_1, \ldots, pt^a_k)$, $pt^b = v^b(pt^b_1, \ldots, pt^b_k)$, and $v^a.sort = v^b.sort$. Also, $pt^a_i \preceq pt^b_i \wedge pt^b_i \preceq pt^a_i$ for $i = 1, \ldots, k$. As proved recursively, $pt^a_i = pt^b_i$ for $i = 1, \ldots, k$. Hence, $pt^a = pt^b$.

Finally, $\preceq$ is transitive, i.e., $pt^a \preceq pt^b \wedge pt^b \preceq pt^c \Rightarrow pt^a \preceq pt^c$.

- If $pt^c = *$, then $pt^a \preceq pt^c$ by definition.
- If pt^c is a terminal, then we must have $pt^c = pt^b = pt^a$.
- Otherwise, $pt^c = v^c(pt^c_1, \ldots, pt^c_k)$, $pt^b = v^b(pt^b_1, \ldots, pt^b_k)$, and $pt^a = v^a(pt^a_1, \ldots, pt^a_k)$. We must have $v^c.sort = v^b.sort = v^a.sort$ and $pt^a_i \preceq pt^b_i \wedge pt^b_i \preceq pt^c_i$ for $i = 1, \ldots, k$. Then $pt^a_i \preceq pt^c_i$ for $i = 1, \ldots, k$, which is proved recursively.

□

Definition 3.3. We define a join operation $\vee$ such that $pt^a \vee pt^b$ is a wildcard pattern that covers the ASTs matched by the inputted wildcard patterns pt^a and pt^b. Specifically, $pt^a \vee pt^b$ is defined as follows:

- $pt^a \vee pt^b = *$, if either pt^a or pt^b is $*$, or the root nodes in pt^a and pt^b have different names.
- If pt^a and pt^b are terminals with the same name, $pt^a \vee pt^b = pt^a = pt^b$.
- In the case $pt^a = v^a(pt^a_1, \ldots, pt^a_k)$, $pt^b = v^b(pt^b_1, \ldots, pt^b_k)$, and $v^a = v^b = v$, $pt^a \vee pt^b = v(pt^a_1 \vee pt^b_1, \ldots, pt^a_k \vee pt^b_k)$.

Theorem 3.2. *Wildcard patterns partially ordered by $\preceq$ is a join-semilattice under $\vee$.*

Proof. We can show that $pt^a \preceq pt^a \vee pt^b$ with the previous two conditions (the pattern is $*$ or a terminal) as the base conditions. For the condition where $pt^a = v^a(pt^a_1, \ldots, pt^a_k)$, $pt^b = v^b(pt^b_1, \ldots, pt^b_k)$, and $v^a = v^b$, we can show $pt^a_i \preceq pt^a_i \vee pt^b_i$ for $i = 1, \ldots, k$ recursively.

Similarly, we have $pt^b \preceq pt^a \vee pt^b$.

Furthermore, we can show that if the least upper bound of pt^a and pt^b is l, then $l = pt^a \vee pt^b$.

For the two base conditions, the result is obvious.

For the condition where $pt^a = v^a(pt^a_1, \ldots, pt^a_k)$, $pt^b = v^b(pt^b_1, \ldots, pt^b_k)$, and $v^a = v^b = v$, l must have the form $v(l_1, \ldots, l_k)$, and $l = pt^a \vee pt^b$ if and only if $l_i = pt^a_i \vee pt^b_i$ for $i = 1, \ldots, k$, which can be proved recursively. □

We have defined the concept of explicitness ($\preceq$) and the most explicit pattern covering a group of patterns ($\vee$). We give an iterative algorithm (Algorithm 3) that mines the wildcard patterns modulo example. We show the properties of Algorithm 3 in the remainder of this section.

Theorem 3.3. *Algorithm 3 is bound to terminate.*

Proof. As shown in Theorem 3.2, the $u \vee v$ returns the least upper bound of u and v, and Σ is initialized by the explicit AST subtrees in I, which are also least upper bounds of themselves. Hence, Σ stores at most all the least upper bounds of the AST subsets in I, which is finite. After each iteration, L is either updated with unseen wildcard patterns, or becomes empty. If L becomes empty, the algorithm terminates. If L is updated with unseen wildcard patterns, the algorithm will terminate after a finite number of iterations. □

Algorithm 3 Mining wildcard patterns modulo example

```
1: function MineLUBs(I)
2:     Σ ← {Pattern(v) | v ∈ I.V}        ▷ Σ is the accumulated wildcard patterns.
3:     L ← Σ                              ▷ L is the newly mined wildcard patterns.
4:     while L ≠ ∅ do
5:         L' ← {u ∨ v | u ∈ L, v ∈ L ∪ Σ}
6:         L ← L' − Σ
7:         Σ ← Σ ∪ L
8:     return Σ
```

Theorem 3.4. *Algorithm 3 is sound (i.e., all the discovered patterns are LUBs of some AST subtrees in I) and complete (i.e., no LUB or equivalent class is missed).*

Proof. (Soundness) Σ is initialized with the LUBs for each AST subtree in I (LUB for a subtree is the subtree itself), and $u \vee v$ returns the LUB of u and v, which is the LUB of the subsets of ASTs represented by u and v. Therefore, all the wildcard patterns in Σ are LUBs of some AST subtrees in I.

(Completeness) For any non-empty subset $\{v_1, \ldots, v_k\} \subseteq I.V$, the corresponding least upper bound must be in Σ. The iteration at line 5 of Algorithm 3 ensures that $v_1 \vee v_2$, $v_1 \vee v_2 \vee v_3$, $\ldots$, $v_1 \vee \ldots \vee v_k$ are all computed. Theorem 3.2 shows that $a \vee b$ is the least upper bound of a and b. Hence, the least upper bound of $\{v_1, \ldots, v_k\}$ must be in Σ. □

4 Alice Results

We used the Eclipse plugin in the virtual machine provided by Alice [1].

Due to the limitation of Alice's expressiveness, only 2 out of the 44 tasks in our benchmark can be completely run, and 6 can be partially run after manually splitting a disjunctive task into multiple sub-tasks. Among the 8 tasks, one does not return any result (hangs), two find all instances, and five miss part of the instances. For example, to search if statements checking `* == null`, we selected a statement `if (a == null) {}` and marked the condition as a feature. The Eclipse plugin then generated a program and used it to search for other candidate examples. While most examples were

[1] `https://ucla.app.box.com/s/yp952moxbcyd7mxollsnlkdfqm6weajd`

found, a special case `if (null == null) {}` was missed. The ALICE plugin does not log the generation process or the generated program, which hinders a further analysis on such failures. Nonetheless, the major problem is still the mismatch of scopes.

The following table tabulates the results of ALICE on the 44 tasks in our benchmark. Note that a task that is out of scope may have multiple symptoms, e.g., unsupported syntax and type checking.

Table 1: ALICE Results

Scope	Symptom	Count
In	Hang	1
	Miss instances	1
Partial	Success	2
	Miss instances	4
Out	Syntax	25
	Type	10
	Other semantics	5

5 Additional Binary Classification Results

Setting	Accuracy		Precision		Recall		#Unparsed	
	CLASS	CoT	CLASS	CoT	CLASS	CoT	CLASS	CoT
GPT4o [1]	89.76%	85.66%	52.70%	44.03%	76.38%	80.06%	0	0
GPT4o [2]	89.56%	86.36%	51.98%	47.01%	76.14%	79.93%	0	0
GPT4o [3]	90.19%	85.36%	54.63%	45.07%	77.20%	82.23%	0	0
QwenCoder [1]	79.33%	84.03%	46.52%	46.76%	80.78%	76.47%	2	1
QwenCoder [2]	78.50%	84.88%	47.22%	45.06%	79.32%	79.06%	2	0
QwenCoder [3]	81.79%	83.79%	46.41%	46.66%	83.01%	80.98%	1	1
OpenCoder [1]	62.18%	67.51%	32.86%	49.56%	51.15%	50.36%	4	7
OpenCoder [2]	62.75%	67.45%	23.31%	41.36%	55.76%	56.46%	1	6
OpenCoder [3]	66.21%	71.97%	25.34%	49.18%	46.13%	47.77%	4	7

6 Prompts

This section listed two prompts (CLASS and CoT) used in the experiments. Their difference is in the one-shot examples. The main experiments in the

evaluation are based on CLASS, and CoT is used for comparison in RQ4.

6.1 Classification

You are a Java expert and user assistant. The user would describe what kind of codes they want to extract. Your task is to analyze the user task and classify some code snippets according to their relations with the user task. Below are some examples.

User Task
"arithmetic expressions that do not involve integers"

Example Code Snippet
```

```
```
public class Foo {
    public void test() {
        int a = 12 * 3;
        double b = 2.3 - 8.6;
        double c = a / b;
        double d = b + b;
    }
}
```
```

The matched codes are ` 2.3 - 8.6 `, ` b + b `.

### Assistant's Task
The user-desired codes would be extracted by an XPath-like program, which navigates through AXIS nodes to reach the desired nodes. An AXIS node usually matches some part of the user description but NEEDS NOT match ALL. It can sometimes be the opposite. Your task is to first identify the AXIS constructs from the user description and list them one by one. Then, you need to read each given code, identify its syntactic type (e.g., expression, statement), and compare the code with the AXIS constructs you identified above. Finally, classify the nodes as one of the following:

`Axis (x)` denotes the code is exactly the (x) Axis construct you identify.
`Outer` denotes the code is an outer scope of target codes or axes.
`Inner` denotes the code is an inner part of target codes or axes.
`Irrelevant` denotes the code has no relevance to the target codes or axes.

Nodes to classify:
(1) ` 12 `
(2) ` 12 * 3 `
(3) ` c `
(4) ` b + b `
```

(5) ` 8.6 `

Answer
According to the user task "arithmetic expressions that do not involve integers", an axis construct can be any one of:
(a) arithmetic expressions
(b) integer expressions
(c) arithmetic operators

The classifications:
(1) Axis (b).
(2) Axis (a) (b).
(3) Irrelevant.
(4) Axis (c).
(5) Inner.

User Task
`<user-given description>`

Example Code Snippet
``` `<example input>` ```

The matched codes are `<example outputs>`.

Assistant's Task
The user-desired codes would be extracted by an XPath-like program, which navigates through AXIS nodes to reach the desired nodes. An AXIS node usually matches some part of the user description but NEEDS NOT match ALL. It can sometimes be the opposite. Your task is to first identify the AXIS constructs from the user description and list them one by one. Then, you need to read each given code, identify its syntactic type (e.g., expression, statement), and compare the code with the AXIS constructs you identified above. Finally, classify the nodes as one of the following:
`Axis (x)` denotes the code is exactly the (x) Axis construct you identify.
`Outer` denotes the code is an outer scope of target codes or axes.
`Inner` denotes the code is an inner part of target codes or axes.
`Irrelevant` denotes the code has no relevance to the target codes or axes.

Nodes to classify:
(1) `<code fragment 1>`
(2) `<code fragment 2>`
...

Answer

6.2 Chain of Thought
```

You are a Java expert and user assistant...

Answer
According to the user task “arithmetic expressions that do not involve integers”, an axis construct can be any one of:
(a) arithmetic expressions
(b) integer expressions
(c) arithmetic operators

The classifications:
(1) ` 12 ` is an expression. It is integer-type, so it matches the axis => Axis (b).
(2) ` 12 * 3 ` is an arithmetic expression. Albeit it is on integers, it still matches the axes listed above => Axis (a) (b).
(3) ` c ` is a declared id. It is with double type. It does not match any axis definition => Irrelevant.
(4) ` + ` is an arithmetic operator. It matches the axis => Axis (c).
(5) ` 8.6 ` is an expression. It is a double expression and nested in target codes, but it does not match the axis listed above => Inner.

User Task
...