1 Wildcard Pattern

Definition 1.1. 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 $v.name$.
- 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 $v.name = v'.name$ and v' has k ordered children that can be matched by $pt_1, \ldots, pt_k$, respectively.

Example 1.1. 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 1.2. 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.

- $pt^a = v^a(pt_1^a, \ldots, pt_k^a)$ and $pt^b = v^b(pt_1^b, \ldots, pt_k^b)$ have the same root nonterminals (i.e., $n^a.name = n^b.name$) and $pt_i^a \preceq pt_i^b$ for $i = 1, \ldots, k$.

Theorem 1.1. *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, $pt^a = v^a(pt_1^a, \ldots, pt_k^a)$, $pt^b = v^b(pt_1^b, \ldots, pt_k^b)$, and $v^a.name = v^b.name$. Also, $pt_i^a \preceq pt_i^b \wedge pt_i^b \preceq pt_i^a$ for $i = 1, \ldots, k$. As proved recursively, $pt_i^a = pt_i^b$ 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, $pt^c = v^c(pt_1^c, \ldots, pt_k^c)$, $pt^b = v^b(pt_1^b, \ldots, pt_k^b)$, and $pt^a = v^a(pt_1^a, \ldots, pt_k^a)$. We must have $v^c.name = v^b.name = v^a.name$ and $pt_i^a \preceq pt_i^b \wedge pt_i^b \preceq pt_i^c$ for $i = 1, \ldots, k$. Then $pt_i^a \preceq pt_i^c$ for $i = 1, \ldots, k$, which is proved recursively.

□

Definition 1.3. 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 1.2. *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 (**??**) that mines the wildcard patterns modulo example. We show the properties of **??** in the remainder of this section.

Algorithm 1 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 1.3. **??** *is bound to terminate.*

Proof. As shown in **??**, 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. □

Theorem 1.4. ?? *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 **??** 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. **??** 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. □