IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 7, 2021, accepted October 29, 2021, date of publication November 16, 2021, date of current version December 8, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3128575

Pattern Matching Based on Object Graphs

WEI KE, (Member, IEEE), AND KA-HOU CHAN, (Member, IEEE)
School of Applied Sciences, Macao Polytechnic Institute, Macau, China
Engineering Research Centre of Applied Technology on Machine Translation and Artificial Intelligence, Ministry of Education, Macao Polytechnic Institute, Macau, China

Corresponding author: Wei Ke (wke@ipm.edu.mo)

This work was supported by the Macao Polytechnic Institute, Macau, under Project RP/ESCA-03/2020.

ABSTRACT Pattern matching has been widely adopted in functional programming languages, and is gradually getting popular in OO languages, from Scala to Python. The structural pattern matching currently in use has its foundation on algebraic data types from functional languages. To better reflect the pointer structures of OO programs, we propose a pattern matching extension to general statically typed OO languages based on object graphs. By this extension, we support patterns having aliasing and circular referencing, that are typically found in pointer structures. With the requirement of only an abstract subtyping preorder on types, our extension is not restricted to a particular hierarchical class model. We give the formal base of the graph model, that is able to handle aliases and cycles in patterns, together with the abstract syntax to construct the object graphs. More complex cases of conjunction and disjunction of multiple patterns are explored with resolution. We present the type checking rules and operational semantics to reason about the soundness by proving the type safety. We also discuss the design decisions, applicability and limitation of our pattern matching extension.

INDEX TERMS Pattern matching, object graph, subtyping, type system, operational semantics, programming language.

I. INTRODUCTION

Pattern matching is a programming language mechanism to test a structure against a pattern at run-time. It has two main purposes, matching and decomposition. Matching is to determine whether the tested structure has the particular components organized in the particular way specified in the pattern. Once the matching is successful, the individual components of the structure can be accessed through new variable bindings introduced in the pattern. Pattern matching supersedes conditional statements in many occasions by stronger specificity, better integrity and higher abstraction, where conditions are organized as a whole structure that is more intuitive and efficient, easier to comprehend, and less complicated.

ALGEBRAIC PATTERN MATCHING

Pattern matching is already common in functional programming. Patterns in the functional world follow the scheme of algebraic data types, or variants, where data objects are generated by hierarchical constructor applications. The patterns are therefore *partial* applications, structured as abstract syntax trees of function invocations. Figure 1 shows the syntax tree

$$Branch(Branch(x, 2, y@Branch(Leaf, 3, Leaf)), 4, z)$$

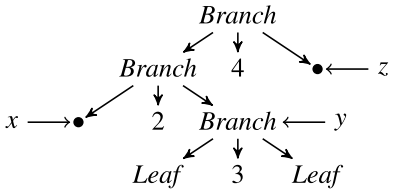

FIGURE 1. Syntax tree of an algebraic pattern.

of a typical binary search tree pattern in a functional language, where x, y and z provide new bindings of variables to the nodes in the application tree. Such a pattern follows the hierarchically structured algebraic constructors, that is simple and easy to match with very clear semantics. That's why pattern matching is widely used in the very early ages of functional languages.

When pattern matching is migrated to OO languages, subtyping between classes must be brought into consideration. A great benefit gained is the ability to perform type tests with variable bindings in an atomic way, that is much safer and eliminates the need of typecasts. However, the algebraic nature of patterns does not change very much. An additional layer is commonly introduced to bridge the

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

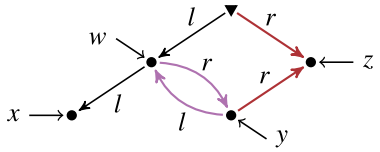

FIGURE 2. Object with aliased and circular relations.

difference between the algebraic view [1] and the underlying pointer structure of an object, for example, the *case class* in Scala [2]. Although the case class constructors create objects with stored structures identical to normal objects, those stored structures are in fact irrelevant. The Scala *extractors* can completely virtualize the case class constructors and abstract the stored structures away. Such pattern matching matches an object against it abstract meaning expressed by the algebraic view rather than the underlying stored structure. Therefore, the semantics of patterns is still in the functional world, with the benefit of higher level abstraction, and the limitation of unable to examine the stored relations of data objects.

The decision to put OO pattern matching in a functional way has its reasons. Besides that the semantics of algebraic patterns is simpler and more familiar to programmers who already know functional pattern matching, the underlying pointer structures of objects can have aliases and cycles, that are more complicated than trees. Figure 2 shows an object with aliased and circular relations. Furthermore, when considering the conjunction and disjunction of multiple patterns with subtyping, a type that is compatible with many types must be resolved. Nevertheless, there's a need to expose and explore the concrete stored structures of objects when writing internal implementations for OO methods. This is often the case where pattern matching is used, since diverging processing of variants externally in OO paradigm usually goes through the subtyping polymorphism. Also, expressing the relations between objects explicitly in the stored structures is a key characteristic of OO programs. Therefore, by addressing these challenges, we propose a pattern matching extension to general OO languages that directly operates on the pointer structures of objects commonly found in most language implementations.

GRAPH-BASED PATTERN MATCHING

We base the theoretical foundation of our extension on the well-established object graph model [3]. An object in this model is a rooted and directed graph with labeled edges. The root represents the object with outgoing edges pointing to components. The edges are labeled with the attributes of the source node. Local variables mark the edges from the stack to roots of objects. Figure 3 shows an object graph on the upper-left, where local variable o leads to the object, which has attributes $o.a$, $o.b$, $o.c$, $o.d$ and $o.e$. Among the attributes, $o.a$ and $o.b$ alias each other and point to the same object. We omit the stack in the shown graphs, since it's somewhat irrelevant to pattern matching. Those edges without a source node are coming from the stack. In our pattern matching graph model,

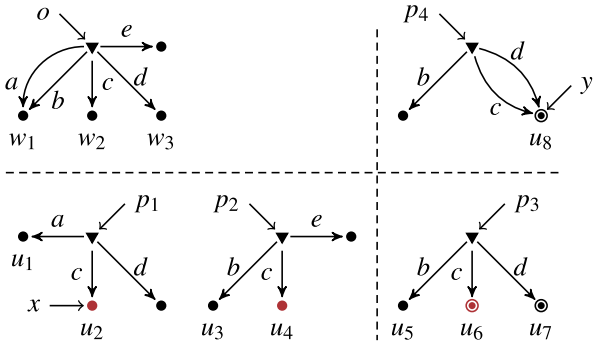

FIGURE 3. Junction of multiple patterns.

both objects and patterns are represented by the object graphs, which are collectively called pattern graphs in the context. The major difference is that a pattern is not a complete object, some or all of the attributes can be omitted.

Given the notion of pattern graphs, our pattern matching is intuitive. A pattern graph matches another when the one *covers* the other. To define the covering between two graphs, we need to figure out the correspondence between the nodes of the two graphs. Beginning with the roots, the nodes led to by the edges with the same label correspond to each other. Cumulatively, the label sequence from the root to a node is the incoming path, or the *trace*, of a node. Although a single path is sufficient to find a node, we must take into account the whole set of incoming paths to a node to obtain the graph structure. The idea to identify a node by how it can be reached is called the trace model [4], where an isomorphism can be built between a node and the set of its incoming paths. The trace model is crucially concise in reasoning about equivalent nodes in different graphs, and plays extremely well with aliases and cycles. There are two aspects in determination of the covering — the layout test and the type test. First, a node w_1 covers another node u_1 when the incoming paths of w_1 include the incoming paths of u_1. To separate the case of aliased nodes, a node can only cover one node in the other graph. As shown in Figure 3, w_1 covers u_1 for $\{[a],[b]\} \supset \{[a]\}$, on the contrary, u_8 covers either u_6 or u_7, but not both. Second, the type $\tau(w_1)$ must be a subtype of $\tau(u_1)$. In OO languages, subtyping is a preorder, i.e., reflexive and transitive [5], [6]. We only stick with this abstraction in order to make the pattern matching general. When all the nodes of a pattern graph are covered by the nodes of the other, we say this pattern is matched by the other. In terms of subtyping, the match relation is covariant.

It's getting more interesting when we look further into the match against multiple patterns. One object can be tested to match all of the patterns, the conjunction, or at least one of the patterns, the disjunction. For example, in Figure 3, object o matches all of p_1, p_2 and p_3, and one of p_3 and p_4. While the tests of multiple patterns are straightforward, the bindings of variable x in the conjunction, and y in the disjunction are more subtle. For conjunction, if there exists a node w_2 that can cover u_2, u_4 and u_6, $\tau(w_2)$ must be a

subtype of all $\tau(u_2)$, $\tau(u_4)$ and $\tau(u_6)$, therefore it's possible to assign a better type to variable x than just $\tau(u_2)$. Also, there exist invalid conjunctions which no graph can possibly match. In Figure 3, p_3 and p_4 cannot be matched by one graph. We need to detect and reject invalid conjunctions. For disjunction, we don't have conflicts between the multiple patterns. However, without knowing which pattern will be eventually matched at run-time, variable bindings are more complicated. For example, what is the type of variable y in the disjunction of p_3 and p_4, $\tau(u_8)$, $\tau(u_6)$ or $\tau(u_7)$? If object o matches the disjunction, for it matches p_3, which node do we bind to y, w_2 or w_3?

CONTRIBUTION AND OUTLINE

With the brief discussion of the problems and ideas, below lists the challenges we are going to address in this work.

1) We build a formal base for the pattern graphs with subtyping. This includes the definitions, morphisms, operations, match relations and algorithms for matching. In addition, we design a language to construct the pattern graphs, in particular with aliases and cycles, as an extension to existing OO languages.
2) We study the compatibility of the pattern matching with the current type-checking and run-time environment, and find a way for smooth integration. We present the type-checking rules and operational semantics, then reason about the type safety.
3) We explore the conjunction and disjunction of multiple patterns, and give our resolution, design decision and algorithms to handle the relatively complex cases at compile-time.

The rest of the article is outlined as follows. Section II summarizes the work in object graphs and pattern matching, from the perspective of OO programming. Section III defines the formal base of pattern matching on object graphs, including pattern conjunction and disjunction, with justifications and constructive algorithms. Section IV presents the language extension for the graph-based pattern matching in abstract syntax, and provides the type-checking rules and operational semantics, together with the proof of type safety. Section V discusses the restriction and limitation of the theory, and the potential solutions. Finally, Section VI concludes the work.

II. RELATED WORK

PATTERN MATCHING IN OO PROGRAMMING

Besides Scala, there have been many attempts to promote pattern matching in OO programs. Because the complexity and multi-paradigm trend of OO languages, a one-for-all solution is difficult. These attempts each have their own focuses. Geller *et al.*, in a full-length technical report [7], proposed a pattern matching framework in a purely OO style for a dynamic typed language: Newspeak, one of the Smalltalk family. Patterns are like normal objects that receive messages to perform the matching operations. This proposal focuses on providing a framework for pattern messaging and pattern combinators, such as conjunction, disjunction, sequencing and negation, as well as the building blocks for structural matching, such as value literals and attribute keywords. How to match a pattern is left abstract and subject to individual object implementations. Widemann and Lepper [8] similarly provided a framework in a library approach requiring generics, that focused on the interfaces and protocols to match patterns against, extract components from and binding variables to objects. Conjunction and disjunction are also supported through the "both" and "either" combinators. Again, how objects and components match patterns is still abstract and therefore shallow to an object.

Solodkyy *et al.* [9] proposed an efficient functional-style pattern matching for C++ as a library, making heavy use of C++ *concepts* and template metaprogramming. A particular pattern is expressed as a template instantiation, and the properties specific to that pattern are constructed at compile-time. Plus the smart use of virtual table pointers as type identifiers for hashing, this gives very minimal performance overhead to the extension. Although C++ extensively supports pointers and references, but the structures are of value semantics. This library is not specific to pointer structures. Ryu *et al.* [10] provided an language extension to Fortress, an experimental language with OO perspective. The extension depends on the *getters* of an object to match its components, and focuses on run-time type tests and multimethods. The fashion of multimethods [11] is a strong application of pattern matching in functional languages, and it has been extensively studied in OO languages in the contrast to the single-dispatch dominance [12]–[14].

Even a bold experiment was carried out in early days by Liu and Myers [15] in their JMatch extension to Java. It generalizes functions and operations to be able to find the arguments back from the results, by introducing the forward and *backward* modes, including automatic iteration to search for elements in a collection. This is more aggressive than the explicit extractors in Scala to decompose objects, with some level of operation inference. Emir *et al.* [16] gave a thorough review of the commonly adopted OO pattern matching techniques, including object decomposition, visitors, type tests, typecase, case classes and extractors. All these techniques are currently available in Scala. Kohn *et al.* [17] proposed a complete solution of pattern matching to the dynamically typed Python language. It lives with the existing Python so well that it will greatly simplify the style of future Python code. This extension has been included in the official Python 3.10 [18], [19], and is thus far the most powerful algebraic pattern matching implementation in the OO world. Finally, OO pattern matching is widely accepted, not only in Scala and Python, it will appear in Java 16 [20] and C# 8 [21], the two most popular production OO languages.

GRAPH-BASED FORMALISM

While most research on OO pattern matching all has focused on type tests and object decomposition in various abstract

ways that coexist with the OO paradigm, with purposes to enhance type safety, expressiveness and conciseness, the pointer structures of the data models in OO programs are also of interest. The semantics of a program must eventually rely on how the data objects are represented [22]. Bornat [23] showed three example proofs of pointer programs in Hoare Logic, with treatment of pointer aliasing by the principle of *spatial separation*. Although the mechanism remains rather low-level, it achieved a level of local reasoning on particular data structures. Ke *et al.* [3] gave a small-step operational semantics of OO programs based on class, object and state graphs. It presents a formal graph model for full featured OO constructs, including stacks and method frames. Ke *et al.* [24] further extended the graph model to a generic type system, to handle recursive types, type variables and type instantiations in the notions of graph morphisms and transformations. Zhao *et al.* [25] employed the transformation of class and object graphs in the refinement of OO data structures, investigating what changes of patterns in the class structure maintained the capability of providing functionalities or services. Zhao *et al.* [26] proposed a graph-based Hoare Logic for reasoning about OO programs. The Hoare proof system consists of a set of logic rules covering most OO constructs, including object creation, local variable declaration and recursive method invocation. The soundness of the logic is given, that every specification proved by the system is valid. There wasn't a proof of completeness, although the logic was believed to be complete.

GRAPH PATTERN MATCHING

Besides the formulation of classes and objects, graphs are widely used in the abstraction of other language notions involving relations, such as query languages. Francis *et al.* [27] described the Cypher 9 query language of the Neo4j graph database. Cypher's core data model consists of values, property graphs and tables. The path patterns in the query language is quite similar to the objects in OO languages, where the nodes are entity patterns like types, and the edges are relations between entities like attributes. Formal semantics is given for the core query language based on the pattern matching. Tong [28] proposed an approach for mapping OO database models into the resource description framework (RDF). This work gives the formal definitions of OO databases and RDF, and maps the query language of OODB to the SPQRQL query of RDF based on graph patterns. Li *et al.* [29] defined taxonomy graphs and simulation admits the "is-a" relation similar to the OO subtyping, but on edge labels rather than node types. By this "is-a" relation, pattern matching on edges is relaxed such that a descendant edge matches an ancestor edge, and the relaxation is bounded by an inheritance distance. Although the purpose of the relaxation was to capture more sensible matches in real-life complex data graphs, the idea can be applied to abstract relations between objects in OO programs.

III. PATTERN GRAPHS AND MATCH RELATION

We first give the formal definitions of graphs, operations, morphisms and relations. We begin with the basic layout graphs consisting of only nodes and edges, and the match relation w.r.t. to a given preorder. Then, we add type assignments to the layout graphs to form the pattern graphs, and describe how to encode program states as graphs. We further expand the theory to match conjunction and disjunction, discuss the restriction and provide our resolution.

A. LAYOUT GRAPHS

Let $\mathscr{N}$ be the set of all nodes, which can be any unique entities, usually associated with types, values and objects, and $\mathscr{A}$ the set of all labels, which are usually variable names. We define the layout graphs on these elements.

Definition 1 (Layout Graph) A layout graph $G = \langle N, E, p\rangle$ is an edge-labeled, rooted and directed graph, where

1) $\text{nodes}(G) = N \subset \mathscr{N}$ is the set of nodes,
2) $\text{edges}(G) = E : N \to \mathscr{A} \to N$ is a partial function, in infix notation, representing the edges, mapping the source node and the label to the target node, and
3) $\text{root}(G) = p \in N$ is the root, and all the nodes are connected to p, $\forall u \in N \cdot \exists \overline{a} \cdot p\overline{Ea} = u$.[1]

We formulate the edges in a layout graph as a *curried* function to imply that the outgoing edges from a node must have unique labels. We write, more intuitively, $u \xrightarrow{a} v \in E$ to denote an edge $uEa = v$. Also, $u \to^* v \in E$ denotes there exists a path from u to v, including single-node paths.

A layout graph represents the layout of an object, where the root becomes the origin. Other nodes of a layout graph are the origins of object components, and each component is represented by the subgraph connecting to its origin.

Definition 2 (Re-rooting) Let $G = \langle N, E, p\rangle$ be a layout graph, and $p' \in N$. We say $G' = \langle N', E', p'\rangle$ is the subgraph re-rooted at node p', denoted as $G' = G \odot p'$, where

1) the nodes are connected to p',

$$N' = \{v \mid \exists p' \to^* v \in E\},$$

2) edges are restricted to the connected nodes, $E' = E|_{N'}$.

Because there may be circular paths going back to the root, re-rooting cannot guarantee a *proper* subgraph. Therefore, it does not necessarily imply a proper reduction. We must explicitly remove all the incoming edges to the root, and examine the residue subgraph. We define the subtraction to remove some edges from a layout graph.

Definition 3 (Subtraction) Let $G = \langle N, E, p\rangle$ be a layout graph, and $X \subseteq E$ the edges to exclude. $G' = \langle N', E', p\rangle$ is the subtraction graph of X from G, denoted as $G' = G \setminus X$, where

1) the nodes are connected via $I = E \setminus X$ to the root,

$$N' = \{v \mid \exists p \to^* v \in I\},$$

[1] The overline $\overline{a}$ notation represents a series $a_1, \ldots, a_n$ of items. However, a variable E bound in the context, if overlined, repeats itself rather than expands into a series, thus $\overline{Ea}$ expands to $Ea_1Ea_2 \ldots Ea_n$.

2) edges are restricted to the connected nodes, $E' = I|_{N'}$.

Since morphisms are functions on nodes, the subtraction gives us a way to break cycles in a graph for a proper reduction on the number of nodes through re-rooting.

B. MATCH RELATION

The structural equivalence of two layout graphs is formalized as an isomorphism, which is a bijection preserving all the outgoing edges of a node.

Definition 4 (Graph Isomorphism) Let $G = \langle N, E, p\rangle$ and $G' = \langle N', E', p'\rangle$ be two layout graphs. G is isomorphic to G', denoted as $G \cong_f G'$, if there exists a bijective function $f : N \to N'$, such that

1) root maps to root, $f(p) = p'$, and
2) edges are preserved, $u \xrightarrow{a} v \in E \iff f(u) \xrightarrow{a} f(v) \in E'$.

If G is isomorphic to a root-preserving subgraph G'' of G', i.e., $\text{root}(G'') = \text{root}(G')$, we say G' *covers* G, denoted as $G' \gtrsim_f G$ or $G \lesssim_f G'$.

Figure 4 illustrates the covering between layout graphs. An object not only has the layout, but also has a type. We associate the type of an object to the root of its layout graph through a type-assignment τ. Suppose there is a preorder $\prec$ defined on these types, e.g., the subclass or subset relation. The type-assignment is there because nodes of graphs are unique, it's inconvenient to reason about ordering between nodes directly. We define the match relation $\sqsubset$ between two layout graphs.

Definition 5 (Graph Match) Let G and G' be two layout graphs, and $\prec$ a preorder on $\text{ran}(\tau)$. G' matches G w.r.t. $\prec$ under morphism f, denoted as $G' \sqsubset_{f|_\prec} G$, or simply $G' \sqsubset_f G$ without ambiguity, if $G' \gtrsim_f G$, such that the preorder exists between the types of any pair of corresponding nodes, i.e., $\forall u \in N \cdot \tau(f(u)) \prec \tau(u)$.

The graph match is a cover relation restricted to the given preorder between the types of nodes. It is trivial to verify that the graph match $\sqsubset$ relation is reflexive. We can also prove that $\sqsubset$ is transitive.

Theorem 1 The graph match $\sqsubset$ relation is a preorder.

Proof. Suppose $G' = \langle N', E', p'\rangle \sqsubset_f G = \langle N, E, p\rangle$ and $G'' = \langle N'', E'', p''\rangle \sqsubset_{f'} G'$. We can verify

1) $f'(f(p)) = p''$,
2) $u \xrightarrow{a} v \in E \implies f'(f(u)) \xrightarrow{a} f'(f(v)) \in E''$, and
3) $\forall u \in N \cdot \tau(f'(f(u))) \prec \tau(u)$.

Hence $G'' \sqsubset_{f' \circ f} G$. The $\sqsubset$ relation is transitive. □

Obviously, if a layout graph G is matched by another graph G', any re-rooted subgraph of G must also be matched by the subgraph of G' re-rooted at the corresponding node.

Theorem 2 If $G' \sqsubset_f G = \langle N, E, p\rangle$, $q \in N$ and $X \subseteq E$, then $G' \odot f(q) \sqsubset (G \setminus X) \odot q$.

Proof. By restricting f to the nodes of $(G \setminus X) \odot q$. □

Corollary 1 If there are two morphisms f_1 and f_2 such that $G' \sqsubset G$ under f_1 and f_2, then $f_1 = f_2$, i.e., the match morphism is unique.

Proof. By induction on the number of nodes. We remove all the incoming edges to the root, and apply Theorem 2. □

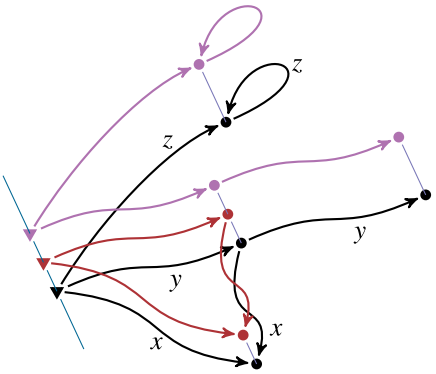

FIGURE 4. Subgraph isomorphism and covering between layout graphs.

On the other hand, by first removing all the incoming edges to the root, we can then re-root the residue subgraph to the adjacent nodes of the root, to reduce the match problem to the subgraphs. By combining the morphisms for these re-rooted subgraphs, if they exist, and adding back the root to the combined morphism, we are able to obtain the morphism for the original layout graph, if it exists.

Theorem 3 Given two layout graphs $G = \langle N, E, p\rangle$ and $G' = \langle N', E', p'\rangle$, Let $G^* = G \setminus \{u \xrightarrow{a} v \in E \mid v = p\}$ be the residue subgraph of G with all the incoming edges to the root removed. If

1) for all the outgoing edges from the root $p \xrightarrow{a} q \in E$,

$$\exists q' \cdot p' \xrightarrow{a} q' \in E' \wedge (p = q \vee G' \odot q' \sqsubset_{f_q} G^* \odot q),$$

2) and for all the incoming edges to the root $u \xrightarrow{b} p \in E$,

$$\exists q \cdot f_q(u) \xrightarrow{b} p' \in E',$$

then $G' \sqsubset_f G$, where $f = \left(\cup_q f_q\right) \cup \{p \mapsto p'\}$.

Proof. Obviously, by the given conditions, the addition of $\{p \mapsto p'\}$ preserves the incoming and outgoing edges of the root. By the definition of layout graphs, all the nodes are connected to the root, therefore any node in a layout graph must be either the root, or in one of the subgraphs re-rooted at the nodes adjacent to the root. Hence all we need to prove is that $\cup_q f_q$ exists, i.e., if $u \in \text{dom}(f_q) \cap \text{dom}(f_r)$ then $f_q(u) = f_r(u)$. This is true by Theorem 2 and Corollary 1, that any (sub)match morphism is unique. □

Although finding the subgraph isomorphism is NP-complete in general, it can be efficiently done for rooted graphs by a simple depth-first search (DFS), led to by Theorem 3. We can determine whether there is a match between two layout graphs by trying to construct the match morphism during the DFS. If the construction fails, we can conclude that the match morphism does not exist.

Algorithm 1 gives the precise construction procedure. It takes the preorder and type-assignment on the nodes and two layout graphs as the input, and output a morphism mapping between the nodes of the two graphs, if the match can be established, otherwise an exception is raised. The procedure initializes the morphism to an empty map, and starts traversing both layout graphs synchronously from their

Algorithm 1 Construction of Match Morphism

Input: preorder: $\prec$, type-assignment: τ, layout graphs: G and G'.
Output: match morphism: f if $G' \sqsubset_f G$.
Exception: *Mismatch*.

```
begin
    ⟨N, E, p⟩ = G, ⟨N', E', p'⟩ = G'
    f ← ∅
    dfs_match(E, p, E', p')
end
function dfs_match(E, p, E', p') begin
    if p ∈ dom(f) then
        raise Mismatch if f(p) ≠ p'
    else
        raise Mismatch if τ(p') ⊀ τ(p)
        raise Mismatch if p' ∈ ran(f)
        f ← f ∪ {p ↦ p'}
        forall the p -a-> q ∈ E do
            raise Mismatch if p' -a-> q' ∉ E'
            dfs_match(E, q, E', q')
        end
    end
end
```

corresponding roots. The dfs_match function performs the DFS. It checks the 1-1 mapping and the preorder relation between the two roots, and accumulates the mapping to the morphism. Then the function checks, w.r.t. the edge labels, the correspondence of the adjacent nodes, and recurs on each pair of them.

C. PATTERN GRAPHS

A pattern graph $P = \langle G, T\rangle$ is a layout graph P with a map T of type-assignment, which associates each node with a type. A type can be an usual class, a subclass, a set of values and particularly a singleton set of one value. The preorder $\prec$ on the node types of such pattern graphs is exactly the subtype relation between two types. In principle, a type t' is a subtype of another type t if all objects or values of t' are elements of t, i.e., elements$(t') \subseteq$ elements(t).

Definition 6 (Subtype) A type is either a class $c \in \mathscr{C}$ or a set of values $V \subset \mathscr{V}$. The subtype relation $\prec$ is inferred by the subtyping rules as follows,

$$\text{(Reflexive)}\ \frac{c \in \mathscr{C}}{c \prec c} \qquad \text{(Subclass)}\ \frac{c, d \in \mathscr{C} \quad c \in \text{supers}(d)}{d \prec c} \qquad \text{(Subset)}\ \frac{W \subseteq V \subseteq \mathscr{V}}{W \prec V}$$

$$\text{(Value}-\text{Type)}\ \frac{V \subseteq \mathscr{V} \quad t \in \mathscr{C} \quad \forall v \in V \cdot \text{class}(v) = t}{V \prec t},$$

where supers(d) is the set of superclasses of d, and class(v) is the class of v.

Theorem 4 The subtype relation $\prec$ in Definition 6 is a preorder.

Proof. All we need is that the subclass relation being unidirectional and transitive, and so it is. □

Definition 7 (Pattern Match) Given two pattern graphs $P = \langle G, T\rangle$ and $P' = \langle G', T'\rangle$, P' matches P w.r.t. preorder $\prec$ on node types, denoted as $P' \sqsubset|_\prec P$, if $G' \sqsubset|_\prec G$, where $\tau(u') \prec \tau(u) \triangleq T'(u') \prec T(u)$.

An object corresponds to a node in a pattern graph, with the root representing the main object, and the edges leading to component objects. In most cases, an object must terminate at some value nodes, which are primitives in a programming language, such as integers and booleans. It is common to use the value itself as the node. This formulation complicates the graph model by having concrete and multiple types of nodes. To keep the nodes pure and abstract, we associate a node p with its value value(p) through a run-time function. We then store this value as a singleton set of one value when constructing the type map of a pattern graph. The real type of the node can be obtained from the value, since any terminus does not start a structure, and must have a known type in the type system.

The state Σ of a running program can be represented by a layout graph H, together with the above mentioned value function and the run-time type information, where H consists of a linked list of scope nodes as a stack, with outgoing edges as local variables to the roots of the objects [3]. Therefore, a pattern graph can be extracted from a state by re-rooting H at the root p of the object. The map of type assignment is constructed from type(p), which returns the actual type of the object.

Definition 8 (Pattern Extraction) A pattern graph $P = \langle G, T\rangle$ is extracted from a state $\Sigma = \langle H, \text{value}, \text{type}\rangle$ at node p, denoted as $P = \Sigma \odot p$, where $G = H \odot p$, and for all $u \in$ nodes(G),

$$T(u) = \begin{cases} \text{type}(u) & \text{if } u \notin \text{dom(value)}, \\ \{\text{value}(u)\} & \text{if } u \in \text{dom(value)}. \end{cases}$$

D. MATCH CONJUNCTION AND DISJUNCTION

When we use pattern matching, it is natural to think about matching one object against multiple patterns, and take the conjunction or disjunction of the individual matches. When taking the conjunction of multiple patterns, we want to know if their overlapping parts are consistent. This observation can tell us if it is possible to have some object matching all these patterns simultaneously. On the other hand, when taking the disjunction of multiple patterns, the overlapping parts give us the common component of the individual matches. This component carries the properties that we can ensure no matter which branch succeeds.

To check whether multiple pattern graphs are consistent, we try to construct a union graph that matches these layouts. The construction is divided into two parts. First, we make sure that an isomorphism is possible from each layout to the union. This tells whether the patterns are structurally consistent. Second, we make sure that there exists at least one type that

is a subtype of all the types associated with the corresponding nodes of the patterns, better there exists an infimum. This tells whether the patterns are consistent in the type system.

Aliasing is an inherit feature of object references, if a node is a common target of two edges, it can be mapped to two nodes of different layouts separately. However, there is no way for this node to be mapped to the two nodes of the union graph simultaneously. We need to address the aliasing issue when we use the union graph to deal with match conjunction.

Definition 9 (De-aliasing) Let $G = \langle N, E, p\rangle$ and $G' = \langle N', E', p'\rangle$ be two layout graphs. G is a de-aliased graph of G', denoted as $G \rightrightarrows_f G'$, if there exists a total function $f : N \to N'$, such that

1) root is preserved, $f(p) = p'$,
2) edges are preserved, $u \xrightarrow{a} v \in E \implies f(u) \xrightarrow{a} f(v) \in E'$ and $u' \xrightarrow{a} v' \in E' \implies \exists u \xrightarrow{a} v \in E \cdot (u', v') = (f(u), f(v))$.

For a layout graph G'', if $\exists G \rightrightarrows G' \cdot G'' \gtrsim G$, we say G'' covers some de-aliased graph of G', denoted as $G'' \gtrsim\!\!\rightrightarrows G'$. If $\exists G \rightrightarrows G' \cdot G'' \lesssim G$, we say G'' is covered by some de-aliased graph of G', denoted as $G'' \lesssim\!\!\rightrightarrows G'$.

Definition 10 (Union) $G_\diamond$ is a union of multiple layout graphs $\overline{G}$, denoted as $G_\diamond \cong_f \cup\overline{G}$, if $\overline{G \lesssim_f G_\diamond}$, and for all G' such that $\overline{G \lesssim G'}$, we have $G_\diamond \lesssim\!\!\rightrightarrows G'$.

The union is the minimum layout graph that covers all the given layouts, and cannot be further de-aliased.

Theorem 5 Given multiple layout graphs $\overline{G}$, if there exists a layout graph G' that covers all of them, there exists the union of $\overline{G}$.

Proof. Since we have $\overline{G \lesssim_f G'}$, we can construct the union $G_\diamond$ from $\overline{f}$ and $\overline{G}$. The nodes of $G_\diamond$ are mapped from $\overline{\mathrm{dom}(f)}$, and combined only when necessary. If a node u_i in $\mathrm{dom}(f_i)$ has the set of incoming paths intersecting with that of a node u_j in $\mathrm{dom}(f_j)$, both u_i and u_j should be mapped to one node [4]. Otherwise, they should be mapped to different nodes. The existence of $\overline{f}$ guarantees the set of incoming paths of a node in one layout will not lead to more than one node in another layout. Such construction splits the target nodes whenever possible, ensuring that the result cannot be further de-aliased. □

As illustrated in Algorithm 2, we try to construct the morphisms $\overline{f}$ from each graph to the union by traversing the graphs simultaneously in DFS, while checking for conflicts. In each iteration of the DFS, we collect the nodes $\overline{p_i}$ that have a common incoming path only in those graphs $\overline{G_i}$, where $\overline{G_i}$ is a subset of $\overline{G}$. These nodes $\overline{p_i}$ should be mapped to the same node in the union. We divide $\overline{p_i}$ into two sets, one set $\overline{p_k}$ are previously visited from other paths, the other set $\overline{p_c}$ are not yet visited. There are two conflicts to check for. First, the visited nodes must all be mapped to one node u. Second, this u must not be an existing target from those graphs $\overline{G_c}$ where the unvisited nodes reside, which means there are two paths leading to one node in $\overline{G_k}$, however leading to two nodes in $\overline{G_c}$. If no conflicts are detected, we assign u, or a new node if u does not exist, as the mapping target of those unvisited nodes $\overline{p_c}$.

Algorithm 2 Construction of Union

Input: layout graphs: $\overline{G}$.
Output: morphisms to union: $\overline{f}$.
Exception: *No-Union*.

```
1  begin
2      ⟨N, E, p⟩ = G
3      f ← ∅
4      dfs_union(E, p)
5  end
6  function dfs_union(E_i, p_i) begin
7      k = {k ∈ i | p_k ∈ dom(f_k)}
8      c = i \ k
9      u = f_k(p_k)
10     raise No-Union if |{u}| > 1
11     if c ≠ ∅ then
12         if u ≠ ∅ then
13             raise No-Union if u_1 ∈ ∪ran(f_c)
14             p_⋄ ← u_1
15         else
16             p_⋄ ← new node
17         end
18         f_c ← f_c ∪ {p_c ↦ p_⋄}
19         forall the a, j = {j ∈ i | p_j -a-> q_j ∈ E_j} do
20             dfs_union(E_j, q_j)
21         end
22     end
23 end
```

Definition 11 (Intersection) $G_\circ$ is an intersection of multiple layout graphs $\overline{G}$, denoted as $G_\circ \cong_{f\circ g} \cap\overline{G}$, if $\overline{G \gtrsim_f \rightrightarrows_g G_\circ}$, and for all G' such that $\overline{G \gtrsim\!\!\rightrightarrows G'}$, we have $G_\circ \gtrsim\!\!\rightrightarrows G'$.

The intersection is the maximum layout graph that all the given layouts cover some de-aliased version of it.

Theorem 6 Given multiple layout graphs $\overline{G}$, there always exists the intersection of $\overline{G}$.

Proof. We can construct the intersection as follows. First, the intersection begins with collecting those incoming paths common in all $\overline{G}$. Then, if any two paths reach the same node in any one of $\overline{G}$, we combine them into one set of incoming paths, otherwise, we keep them as two separate sets. Finally, each set of incoming paths is mapped to a node of the intersection. Similar to the union, such construction splits the target nodes whenever possible, ensuring that the result cannot be further de-aliased. □

As illustrated in Algorithm 3, the construction of intersection is simpler. We again traverse the graphs $\overline{G}$ simultaneously in DFS. In each iteration, we collect only those node $\overline{p}$ with a common incoming path in all the given layouts. If some of the nodes $\overline{p_k}$ are previously visited, they may have been mapped to multiple nodes $\overline{u}$. We take one of the targets u_1 to unify

Algorithm 3 Construction of Intersection

Input: layout graphs: $\overline{G}$.
Output: morphisms to intersection: $\overline{f}$.

begin
  $\overline{\langle N, E, p\rangle = G}$
  $\overline{f \leftarrow \emptyset}$
  dfs_inter($\overline{E}, \overline{p}$)
end
function dfs_inter($\overline{E}, \overline{p}$) **begin**
  $\overline{k = \{k \mid p_k \in \mathrm{dom}(f_k)\}}$
  $\overline{u = f_k(p_k)}$
  if $\overline{u} \neq \emptyset$ **then**
    $p_\circ \leftarrow u_1$
  else
    $p_\circ \leftarrow$ new node
  end
  $\overline{f \leftarrow f \cup \{p \mapsto p_\circ\}}$
  if $|\overline{u}| \neq |\overline{p}|$ **then**
    forall the $a \cdot \overline{p \xrightarrow{a} q \in E}$ **do**
      dfs_inter($\overline{E}, \overline{q}$)
    end
  end
end

the mapping target of all these nodes. If all $\overline{p}$ are unvisited, we map all of them to a new node.

The union and intersection give us the common structures of multiple layout graphs, however without respect to the preorder between node types. If there is an object matching all the given layouts $\overline{G}$, it must contain a subgraph G' that can be de-aliased to the union of $\overline{G}$. Each node u' of G' corresponds to one node $u_\diamond$ of the union in each component match of the conjunction. This $u_\diamond$ must therefore be of a subtype of all the nodes mapped to it from $\overline{G}$. We can assume a better type of $u_\diamond$ and thus u', if there exists an infimum type of all the nodes mapped to $u_\diamond$.

Definition 12 (Match Conjunction) G' matches the conjunction of layout graphs $\overline{G}$, w.r.t. preorder $\prec$, denoted as $G' \sqsubset|_\prec \curlywedge \overline{G}$, if $\overline{G' \sqsubset|_\prec G}$.

Algorithm 4 constructs the type assignment of the union based on the given patterns. We first obtain the union morphisms from the layouts by calling Algorithm 2. Then, for each node $u_\diamond$ in the union, we collect all the layouts having nodes mapped to $u_\diamond$. These layouts correspond to morphisms $\overline{f_k}$ and type assignments $\overline{T_k}$. Thus, we are able to look up the types $\overline{t_k}$ associated with the nodes mapped to $u_\diamond$, and check for the legitimacy of subtyping $\overline{t_k}$. If the type system does not permit such subtyping, we must conclude that the conjunction of these patterns are not possible. If the subtyping is possible, we try to find the infimum $t_\diamond$ of $\overline{t_k}$ and associate it with $u_\diamond$. The type of $u_\diamond$ is *No-Type* if there exists no infimum.

On the contrary, for match disjunction, an object is to match only one of the multiple patterns. This is always possible because neither structural nor subtyping conflicts can occur

Algorithm 4 Construction of Match Conjunction

Input: pattern graphs: $\overline{P}$.
Output: morphisms to union: $\overline{f}$ and type assignment: $T_\diamond$.
Exception: *No-Union*, *No-Conj*.

begin
  $\overline{\langle G, T\rangle = P}$
  Algorithm 2: $\overline{G} \rightarrow \overline{f}$
  $T_\diamond \leftarrow \emptyset$
  $N_\diamond = \cup\overline{\mathrm{ran}(f)}$
  forall the $u_\diamond \in N_\diamond$ **do**
    $\overline{k = \{k \mid u_\diamond \in \mathrm{ran}(f_k)\}}$
    $\overline{t_k = T_k(f_k^{-1}(u_\diamond))}$
    raise *No-Conj* **if** $\nexists t \cdot \overline{t \prec t_k}$
    $t_\diamond = \inf(\overline{t_k})$
    $T_\diamond \leftarrow T_\diamond \cup \{u_\diamond \mapsto t_\diamond\}$
  end
end

Algorithm 5 Construction of Match Disjunction

Input: pattern graphs: $\overline{P}$.
Output: morphisms to intersection: $\overline{f}$ and type assignment: $T_\circ$.

begin
  $\overline{\langle G, T\rangle = P}$
  Algorithm 3: $\overline{G} \rightarrow \overline{f}$
  $T_\circ \leftarrow \emptyset$
  $N_\circ = \cup\overline{\mathrm{ran}(f)}$
  forall the $u_\circ \in N_\circ$ **do**
    $\overline{t = \cup\{T(u) \mid f(u) = u_\circ\}}$
    $t_\circ = \sup(\overline{t})$
    $T_\circ \leftarrow T_\circ \cup \{u_\circ \mapsto t_\circ\}$
  end
end

for only one graph. Any object that matches one of the given layouts $\overline{G}$ must contain a subgraph G' which is a de-aliased graph of the intersection of $\overline{G}$. Each node u' of G' is mapped to the corresponding node $u_\circ$ of the intersection possibly via any component match of the disjunction. This $u_\circ$ must therefore be of a supertype of all the nodes mapped to it, to cope with all the possible component matches. Even if such a supertype exists, it is possibly not unique. We can only associate a type with $u_\circ$ when there is a supremum type of all the nodes mapped to $u_\circ$.

Definition 13 (Match Disjunction) G' matches the disjunction of layout graphs $\overline{G}$, w.r.t. preorder $\prec$, denoted as $G' \sqsubset|_\prec \curlyvee \overline{G}$, if there exists $G'' \in \overline{G}$ such that $G' \sqsubset|_\prec G''$.

Algorithm 5 constructs the type assignment of the intersection based on the given patterns. We first obtain the intersection morphisms from the layouts by calling Algorithm 3. Then, for each node $u_\circ$ in the intersection, we find the nodes mapped to $u_\circ$ in all the patterns. Note that a morphism to the intersection is not injective, there can be more than one nodes

TABLE 1. Abstract syntax of pattern matching.

P	::=	$[r:](K \mid r \mid V)$	pattern
K	::=	$T\ \overline{a = P}$	class pattern
r			pattern reference
V			value set
T			class type
a			attribute
J	::=	$P \mid \curlywedge\overline{P} \mid \curlyvee\overline{P}$	pattern junction
M	::=	$e \ltimes \overline{J \to S}$	match statement
S			statement

mapped to $u_\circ$. Next, we look up the types $\bar{t}$ associated with these nodes. Finally, we try to find the supremum $t_\circ$ of $\bar{t}$ and associate it with $u_\circ$. The type of $u_\circ$ is *No-Type* if there exists no supremum.

IV. PATTERN LANGUAGE

With the detail of the graph model for patterns and layouts fully discussed, we give a language extension for pattern matching to general OO languages. We then present the type-checking rules and operational semantics, and prove the type safety to reason about the soundness of our model.

A. SYNTAX

The abstract syntax to define patterns and matches is given in Table 1. The syntax is much like conventional ones found in other OO and functional languages, with the addition of pattern references. The pattern references play two roles, to be referred to from inside the pattern to produce aliases and cycles, and to be referred to from the match-selected statement as object bindings — variables bound to components of matching objects.

A pattern P can have an optional label, which is a pattern reference r, followed by a pattern specification in one of the three forms — a class pattern, a pattern reference or a set V of values. A class pattern K is a class type T followed by zero or more attribute specifications, each of which is an attribute name a equal to an inner pattern P. For example, the pattern p_4 shown in Figure 3 can be expressed as

$$T_1(b = T_2, c = y : T_3, d = y).$$

The more complex pattern shown in Figure 2 is written as

$$T(l = w : T(l = x : T, r = y : T(l = w, r = z)), r = z : T).$$

A match statement evaluates an expression e and matches the result against many cases in turn. Each case consists of a pattern junction J, which can be a single pattern, a pattern conjunction $\curlywedge\overline{P}$ or a pattern disjunction $\curlyvee\overline{P}$, and an associated statement S. The first successfully matched case selects the associated statement to execute, with the new object bindings introduced by the references declared in the matched pattern junction.

B. PATTERN GRAPH CONSTRUCTION

To carry out the operations of pattern matching and check for its type safety based on the graph model described in Section III, we must first construct the pattern graph for each pattern specified in the abstract syntax. The construction is straightforward following the structure of the specification, with three components to output — a layout graph G, a map T from nodes to types and a map R from pattern references to nodes. Algorithm 6 describes the procedure in detail, where, for clarity, we enclose syntax elements in guillemets $\ll\gg$ to distinguish them from other variables.

Algorithm 6 Construction of Pattern Graph

Input: pattern: $\ll P \gg$.
Output: layout graph: G, type map: T and reference map: R.
Exception: *Undef-Ref*, *Redef-Ref*.

begin
  $N, E, T, R \leftarrow \emptyset$
  $p \leftarrow$ parse($\ll P \gg$)
  raise *Undef-Ref* **if** ran(R) $\nsubseteq N$
  $G \leftarrow \langle N, E, p \rangle$
end
function parse($\ll P \gg$) $\rightarrow p$ **begin**
  if $\ll P \Rightarrow r : Q \gg$ **then**
    $p \leftarrow$ parse($\ll Q \gg$)
    if $\ll r \gg \in$ dom(R) **then**
      $r \leftarrow R(\ll r \gg)$
      raise *Redef-Ref* **if** $r \in N$
      $E \leftarrow E[p/r], R \leftarrow R[p/r]$
    end
    $R \leftarrow R \uplus \{\ll r \gg \mapsto p\}$
  else if $\ll P \Rightarrow T\ \overline{a = Q} \gg$ **then**
    $p \leftarrow$ new node
    $N \leftarrow N \cup \{p\}$
    $T \leftarrow T \cup \{p \mapsto \ll T \gg\}$
    $\overline{q \leftarrow \text{parse}(\ll Q \gg)}$
    $E \leftarrow E \cup \{p \xrightarrow{\overline{\ll a \gg}} q\}$
  else if $\ll P \Rightarrow r \gg$ **then**
    if $\ll r \gg \in$ dom(R) **then**
      $p \leftarrow R(\ll r \gg)$
    else
      $p \leftarrow$ new node
      $R \leftarrow R \cup \{\ll r \gg \mapsto p\}$
    end
  else if $\ll P \Rightarrow V \gg$ **then**
    $p \leftarrow$ new node
    $N \leftarrow N \cup \{p\}$
    $T \leftarrow T \cup \{p \mapsto \ll V \gg\}$
  end
end

In the algorithm, function parse recursively decomposes a pattern specification $\ll P \gg$ into a graph with root p by updating the nodes N, the edges E, the type map T and the reference map R. The parse function performs case analysis on the pattern specification. First, when it is a reference definition $\ll r : Q \gg$, we recursively obtain the root p of $\ll Q \gg$, and

associate it with the reference $\ll r \gg$ in R. It is possible for a reference to be used before definition, we generate a placeholder node r for the forward use, and replace it with the real node at this definition stage. A placeholder is not put into the node set N, therefore we can check if r is indeed a placeholder, or a node that has defined the reference already. For a placeholder r, we need to replace it with the real node p. This replacement is performed in the edges E, for all the target nodes equal to r, and also in the reference map R, for all the image nodes equal to r. Note that we allow a reference to label another reference. Second, when it is a class pattern $\ll T\ \overline{a = Q}\gg$, we create a new node p and associate it with type $\ll T \gg$. We add p to the node set N and link the outgoing edges to the roots recursively obtained from $\overline{\ll Q \gg}$, with labels equal to the attribute names $\overline{\ll a \gg}$. Third, when it is a reference occurrence $\ll r \gg$, we try to look up R for its corresponding node, or create a new placeholder without placing it into the node set N. Last, when it is a value set $\ll V \gg$, we simply create a new node and associate it with $\ll V \gg$ in the type map T.

The graph construction merely creates nodes and links between the nodes. This process is always successful. However, with the addition of references, there are two exceptional cases — a reference is defined more than once (*Redef-Ref*), and a reference occurs but not defined (*Undef-Ref*). The former is detected at the time of parsing a label prefix, while the latter is detected when the parsing is complete, by checking whether all the placeholders have been resolved.

C. TYPE-CHECKING

The pattern matching extension is type-checked in two parts, the pattern and the match. A pattern can be a class pattern, a reference, a pattern conjunction or a pattern disjunction. A class pattern is converted to a pattern graph and checked by recursively reducing to each node and the outgoing edges. A node in a pattern graph is well-typed when all its outgoing edges are labeled with the attributes of the associated class, and also the targets are all well-typed. To avoid infinite loops in the DFS, we add the visited nodes to the typing environment before going into recursion. A reference is checked by resolving it to a node with the reference map. A class pattern is well-typed only when the corresponding pattern graph and all the references declared in the pattern are well-typed.

The type-checking rules are given in Table 2. Each typing rule involves a typing environment Γ and optionally some additional structures, separated by a semicolon, as the context. The result of a rule is the well-typedness and type-assignment of the checked element, possibly with some extra output, also separated by a semicolon, during the checking. This output can be necessary in the subsequent procedures.

For a pattern conjunction, we individually check the component patterns first, and try to construct the match conjunction. Then, we promote the type of each referenced node to the infimum type associated with the union. If such infimum type does not exist, we fall back to the original type of the node. For a pattern disjunction, we try to construct

TABLE 2. Type-checking rules.

$$\text{(NODE)}\ \frac{t = T(p) \qquad \overline{p \xrightarrow{a} q \in E} \qquad p : t, \Gamma;\ E, T \vdash \overline{q : s} \qquad \overline{s \prec \text{class}(t.a)}}{\Gamma;\ E, T \vdash p : t}$$

$$\text{(GRAPH)}\ \frac{P = \langle N, E, p, T\rangle \qquad \Gamma;\ E, T \vdash p : t}{\Gamma \vdash P : t}$$

$$\text{(REF)}\ \frac{p = R(r) \qquad P = \langle N, E, p', T\rangle \qquad \Gamma; E, T \vdash p : t}{\Gamma;\ P, R \vdash r : t}$$

$$\text{(PATTERN)}\ \frac{\text{Algorithm 6} : \text{«}P\text{»} \to G, T, R \qquad P = \langle G, T\rangle \qquad \Gamma \vdash P : t \qquad \overline{r} = \text{dom}(R) \qquad \Gamma;\ P, R \vdash \overline{r : s}}{\Gamma \vdash \text{«}P\text{»} : t, \overline{r : s};\ P, R}$$

$$\text{(CONJ)}\ \frac{\Gamma \vdash \overline{\text{«}P\text{»} : t, \overline{r : s};\ P, R} \qquad \text{Algorithm 4} : \overline{P} \to \overline{f}, T_\diamond \qquad \overline{s' = \min(s, T_\diamond(f(R(r))))} \qquad \hat{t} = T_\diamond(f_1(\text{root}(P_1))) \qquad \overline{t' = \min(t, \hat{t})}}{\Gamma \vdash \text{«}\curlywedge \overline{P}\text{»} : \hat{t}, \overline{\text{«}P\text{»} : t'}, \overline{r : s'};\ \overline{P}, \overline{f}, \overline{R}}$$

$$\text{(DISJ)}\ \frac{\Gamma \vdash \overline{\text{«}P\text{»} : t, \overline{r : s};\ P, R} \qquad \text{Algorithm 5} : \overline{P} \to \overline{f}, T_\circ \qquad \overline{s' = T_\circ(f(R(r)))} \qquad \hat{t} = T_\circ(f_1(\text{root}(P_1)))}{\Gamma \vdash \text{«}\curlyvee \overline{P}\text{»} : \hat{t}, \overline{\text{«}P\text{»} : t}, \overline{r : s'};\ \overline{P}, \overline{f}, \overline{R}}$$

$$\text{(CASE)}\ \frac{\Gamma \vdash \text{«}J\text{»} : t, \overline{r : s};\ \overline{P}, \overline{f}, \overline{R} \qquad \overline{r : s}, \Gamma \vdash \text{«}S\text{»}}{\Gamma \vdash \text{«}J \to S\text{»} : t}$$

$$\text{(MATCH-T)}\ \frac{\Gamma \vdash \overline{\text{«}J \to S\text{»} : t} \qquad \Gamma \vdash e : t' \qquad \overline{t' \prec t \vee t \prec t'}}{\Gamma \vdash \text{«}e \ltimes \overline{J \to S}\text{»}}$$

the match disjunction. Only a referenced node that occurs commonly in all the component patterns has a type, which must be the supremum type associated with the intersection. If such supremum type does not exist, the referenced node has *No-Type*, and subsequent uses of the reference will be ill-typed. This includes the case where a referenced node does not occur in all the component patterns.

A match is a test of an expression against multiple cases. Therefore, to type-check a match, we need to check the individual cases, and whether the expression is type-compatible with all the cases. Each case consists of a pattern and a statement. The checking of the pattern produces the type of the pattern and a series of type-assigned references, as the additional variables available to the statement. We check the type of the pattern against the expression to match, and check the statement under the context with those additional variables declared.

D. OPERATIONAL SEMANTICS

The operation of a pattern match statement is to evaluate the expression, then to match the result against the list of cases in turn, and finally to select the statement in the matched case to execute. In the graph model, the expression $\ll e \gg$ to match is evaluated to a node in the state graph, and then the pattern

TABLE 3. State transition rules.

$$\text{(MATCH-S)}\ \frac{\text{eval}(\langle\!\langle e \rangle\!\rangle, \Sigma) = p \quad P = \Sigma \odot p}{\langle\!\langle e \ltimes \overline{J \to S} \rangle\!\rangle, \Sigma\rangle \to \langle P \ltimes \langle\!\langle \overline{J \to S} \rangle\!\rangle, \Sigma\rangle}$$

$$\text{(MATCH-NONE)}\ \langle P \ltimes \langle\!\langle \emptyset \rangle\!\rangle, \Sigma\rangle \to \Sigma$$

$$\text{(MATCH-SOME)}\ \frac{\langle\!\langle \overline{J \to S} \rangle\!\rangle = \langle\!\langle J_1 \to S_1, \overline{J' \to S'} \rangle\!\rangle \quad \Gamma \vdash \langle\!\langle J_1 \rangle\!\rangle; \overline{P_1}, \overline{f_1}, \overline{R_1}}{\text{match}(P, \langle\!\langle J_1 \rangle\!\rangle, \overline{P_1}, \overline{f_1}, \overline{R_1}) = (\bar{r}, \bar{q})}}{\langle P \ltimes \langle\!\langle \overline{J \to S} \rangle\!\rangle, \Sigma\rangle \to \langle\!\langle \textbf{var}(\overline{r = q}); S; \textbf{end}(\bar{r}) \rangle\!\rangle, \Sigma\rangle}$$

$$\frac{\text{match}(P, \langle\!\langle J_1 \rangle\!\rangle, \overline{P_1}, \overline{f_1}, \overline{R_1}) = \emptyset}{\langle P \ltimes \langle\!\langle \overline{J \to S} \rangle\!\rangle, \Sigma\rangle \to \langle P \ltimes \langle\!\langle \overline{J' \to S'} \rangle\!\rangle, \Sigma\rangle}$$

$$\text{match}(P, \langle\!\langle J_1 \rangle\!\rangle, \overline{P_1}, \overline{f_1}, \overline{R_1}) \triangleq$$

$$\text{(MATCH-ONE)}\ \frac{\langle\!\langle J_1 \rangle\!\rangle = \langle\!\langle P \rangle\!\rangle \quad P \sqsubset_f P_1}{\bar{r} = \text{dom}(R_1) \quad \overline{q = f(R_1(r))}}$$

$$\text{(MATCH-CONJ)}\ \frac{\langle\!\langle J_1 \rangle\!\rangle = \langle\!\langle \curlywedge \overline{P_1} \rangle\!\rangle \quad \overline{P \sqsubset_f P_1} \quad R' = \bigcup \overline{R_1} \quad f' = \bigcup \overline{f}}{\bar{r} = \text{dom}(R') \quad \overline{q = f'(R'(r))}}$$

$$\text{(MATCH-DISJ)}\ \frac{\langle\!\langle J_1 \rangle\!\rangle = \langle\!\langle \curlyvee \overline{P_1} \rangle\!\rangle \quad \exists P' \in \overline{P_1} \bullet P \sqsubset_f P' \quad R' = \bigcup \overline{R_1} \quad f'_1 = \bigcup \overline{f_1} \quad \overline{u' = \bigsqcap \{u \in \text{dom}(f) \mid f'_1(u) = f'_1(R'(r))\}}}{\bar{r} = \text{dom}(R') \quad \overline{q = f(u')}}$$

$$\text{(MISMATCH)}\ \frac{\text{otherwise}}{\emptyset}$$

graph P rooted at this node is extracted from the state graph. The pattern graph is matched in turn against all the cases until a matched case is found, otherwise there is no statement to execute in this match.

The operational semantics is given by the state transition rules listed in Table 3. Before we can match the pattern graph of the expression, we need to obtained the pattern graphs of the patterns by taking the output of the type-checking procedures. Beside to match by constructing the match morphisms, we also need to obtain the references declared in the patterns, as well as their corresponding nodes. This task is done in the match function, divided into three cases, a single pattern, a conjunction, or a disjunction. For a single pattern, a reference r is mapped to a node of the pattern graph P_1 by the reference map R_1, then we can use the match morphism f from the pattern graph to the expression graph P to obtain the corresponding node in the state graph. For a conjunction, since the match of the expression graph must be done against all the component pattern graphs, we have a series of match morphisms from different pattern graphs to the same expression graph, their domains do not intersect with each other. We can simply union these morphisms into one to obtain the nodes of the references.

A disjunction is more complicated, because there requires only one match of the component pattern graphs, and a node in the intersection is possibly de-aliased to two or more nodes in a component pattern. We have restricted the references in a disjunction only to those nodes in the intersection, by assigning *No-Type* to the other nodes. Therefore, we are left with a one-to-many mapping from the intersection to the matched pattern, and thus to the expression graph. We use the non-deterministic choice $\sqcap$ to bind a reference to one of the possible corresponding nodes in the state graph. Note that, in function match, f'_1 is the union of the morphisms from the patterns to their intersection. Once we have the bindings of the references to the nodes in the state graph, we can declare the additional variables as a normal variable declaration, **var** ... **end**, and execute the statement of the matched case in this scope.

E. TYPE SAFETY

The type safety of pattern matching means a well-typed pattern match statement cannot go wrong at runtime. Since the only new operation introduced by pattern matching is the new bindings of pattern references to runtime objects, the statement of the matched case is type-safe when the runtime objects truly have the types associated with the nodes of the matched pattern.

Theorem 7 For the same pattern junction $\ll J \gg$, let $(\bar{r}, \bar{q})$ be the result of function match in Table 3, $\overline{r : s}$ the type assignment obtained by (CASE) in Table 2. We have $\overline{\text{type}(q) \prec s}$.

Proof. We study the type t of the node q associated to a reference r in function match by case analysis. In the case of (MATCH-ONE), t comes from the type assignment of pattern graph P. By the definition of $\sqsubset_f$, $t \prec s$, where s is the type of node $R_1(r)$ in pattern graph P_1, such that $q = f(R_1(r))$. In fact, in (MATCH-SOME), P_1 and R_1 are obtained from the type-checking rules. According to (PATTERN), (GRAPH) and (NODE), s is exactly the type assigned to r in the type-checking.

Similarly, in the case of (MATCH-CONJ), P matches all $\overline{P_1}$. The type t of q is a subtype of all the types $\overline{t_k}$ of the corresponding nodes in each of $\overline{P_1}$. By Algorithm 4, if there exists $\inf(\overline{t_k})$, it is assigned to r in (CONJ), for $\inf(\overline{t_k})$ is a subtype of each $\overline{t_k}$, and t must be a subtype of the infimum by definition. It there exists no infimum, *No-Type* is assigned to the node in the union, thus the $\overline{\min(\textit{No-Type}, t_k)}$ in (CONJ) assigns r with one of $\overline{t_k}$, of which t is a subtype.

In the case of (MATCH-DISJ), P matches only one graph P' of $\overline{P_1}$. The node q bound to r is non-deterministically chosen from nodes $\overline{f(u)}$ in P, matching those nodes $\overline{u}$ in P' that are finally mapped to the same node $u_\circ$ in the intersection of $\overline{P_1}$. The type t of q is a subtype of the corresponding node in P' mapped to $u_\circ$. By Algorithm 5 and (DISJ), r is assigned with the supremum type of all the node in $\overline{P_1}$ mapped to $u_\circ$, or *No-Type* if the supremum does not exist. Obviously, in either case, t is a subtype of the type assigned to r. □

V. DISCUSSION

We formalize pattern matching in the notions of object graphs. Although our intention is to specify patterns according to the underlying stored structures, the nodes, edges,

labels and preorder are in fact abstract, thus can represent a wider range of relations between objects. For example, we can replace concrete attributes with getter methods. The patterns to match will not change in this case, and the pattern graph of the object to examine can be obtained by first calling the getters. Even if side effects exist, the semantics of evaluating the getters can be made unambiguous, possibly imitating lazy evaluation. Edge labels can be parameter positions of a decomposition function, patterns with such positional labels can be matched by the tuple result of the decomposition.

The nodes in our pattern graphs are pure identities. Value objects have to be represented as pure nodes associated with singleton sets. Since we treat value sets as types, matching values is classified as matching types. Because not only an identical type matches, a subtype also matches, this gives the benefit to match a value against a range of values, for a subset can be seen as a subtype. For example, pattern $T(e = [5, \ldots, 9])$ is matched by a tree node whose element is between 5 and 9. This arrangement supersedes the traditional matching of value cases.

The mechanism to specify and detect aliasing is a distinct feature of our system. However, it also brings certain restriction. Sometimes we want to ignore the aliasing by intention, especially for immutable objects whose aliases are safe. For example, we may write a pattern $T(c = S, d = S)$ to test if an object has two components both of type S. While an object with layout p_3 in Figure 3 matches the pattern, but an object with layout p_4 does not. This causes some inconvenience. In the case to ignore aliasing, we can try match conjunction $T(c = S) \curlywedge T(d = S)$ to split the aliased node into two component tests. However, we cannot use match conjunction to ignore cycles, for example, the layout in Figure 2 does not match pattern $T(l = T(r = T(l = T)))$, because there's no way to split a node that is both a source and a target into two tests. A better solution is to construct an intermediate object graph to split the nodes that are to ignore aliases and cycles beforehand, and match the intermediate graph against the pattern. This can be optimized into a slightly modified match algorithm, based on Algorithm 1, to split the nodes on the fly in construction of the match morphism.

The subtyping in our pattern matching only needs to be a preorder in the match of a single pattern. In match conjunction, the node of an object to match must has a subtype of multiple types. In the classical single inheritance hierarchy, unless these types form an inheritance chain, the match cannot succeed. Modern OO languages all have interfaces that support multiple inheritance, therefore, a subtype of multiple unrelated types is possible. However, there may exist no infimum. For a match conjunction $T(a = x : I) \curlywedge T(a = J)$, we may have A and B both extending I and J, but the common $\inf(I, J)$ does not exist. In this case, we must fall back the type of x to I even if we know that x must also have type J when a match succeeds. In languages supporting intersection types [30], $\inf(I, J) = I \cap J$ can be generated and assigned to x for better typing.

On the other hand, to find the type of a variable in match disjunction is even more complicated. Consider the following disjunction,

$$p : T(a = I, b = J, c = x : X) \curlyvee q : T(a = y : K, b = y).$$

Not like in a dynamic language such as Python, where using undefined variables is allowed to raise run-time exceptions, for a statically typed language, we must guarantee that a program passes type-checking must not have such exceptions. First, we must reject the use of any variable like x which is not in the common part of the disjunction. This is done by assigning *No-Type* to x. Second, we must collect all the types of a node that possibly occurs in any part of the disjunction with the same incoming path. For y, the set of incoming paths is $\{[a], [b]\}$, therefore, the type of y must be a supertype of all the types reachable from $\{[a], [b]\}$. The best choice of this supertype is the supremum, $\sup(I, J, K)$, if it exists. Otherwise, we choose *No-Type* to reject y, in fact, although nondeterministic, any supertype of I, J and K will not break the type system. Even if y is properly typed, there is still a problem when an object o matches p rather than q. Both $o.a$ and $o.b$ are legible for the initialization of y with type safety. We take nondeterministic choice here, and any implementation can decide which one to choose in a predictable way. With such complexity handling match disjunction, Scala even decides to forbid variable binding in its *pattern alternatives*.

The graph algorithms, type rules and semantic rules presented in this paper have been implemented in Python, with the test cases. The source code can be downloaded from GitHub for further exploration and verification of the theory.[2]

VI. CONCLUSION

We formalize pattern matching for OO languages based on object graphs. Beyond the algebraic view of data objects found in functional languages, and the addition of type tests for OO subtyping, we develop a pattern matching theory tackling the underlying pointer structures of objects, in particular with the intention to examine aliases and cycles. We define the cover relation on layout graphs and the match relation on pattern graphs in terms of graph morphisms, and study the key properties. The theory is further expanded to incorporate match conjunction and disjunction, with more complex cases addressed. We design the language extension in the form of abstract syntax to specify the patterns and match operations. Key algorithms are given in detail to construct the pattern graph, match relation, conjunction and disjunction. We present the type-checking rules and operational semantics for the pattern matching extension. We give the soundness of the theory by proving the type safety theorem. This theory keeps the graphs and subtyping preorder abstract, thus sets a solid and general foundation for OO pattern matching on pointer structures. Together with the discussion on limitations

[2]github.com/ChanKaHou/OGPM

and potential solutions, this work is a reference base for future language design and improvement.

REFERENCES

[1] F. W. Burton and R. D. Cameron, "Pattern matching with abstract data types," *J. Funct. Program.*, vol. 3, no. 2, pp. 171–190, Apr. 1993.

[2] M. Odersky, P. Altherr, V. Cremet, G. Dubochet, B. Emir, P. Haller, S. Micheloud, N. Mihaylov, A. Moors, L. Rytz, M. Schinz, E. Stenman, and M. Zenger. (2021). *Scala Language Specification Version 2.13*. Accessed: Sep. 30, 2021. [Online]. Available: https://scala-lang.org/files/archive/spec/2.13/

[3] W. Ke, Z. Liu, S. Wang, and L. Zhao, "A graph-based operational semantics of OO programs," in *Proc. Int. Conf. Formal Eng. Methods*. Rio de Janeiro, Brazil: Springer, 2009, pp. 347–366.

[4] C. A. Hoare and H. Jifeng, "A trace model for pointers and objects," in *Proc. Eur. Conf. Object-Oriented Program.* New York, NY, USA: Springer, 1999, pp. 1–18.

[5] M. Abadi and L. Cardelli, *A Theory Objects*. New York, NY, USA: Springer, 1996.

[6] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994.

[7] F. Geller, R. Hirschfeld, and G. Bracha, *Pattern Matching for Object-Oriented Dynamically Typed Programming Language*, vol. 36. Potsdam, Germany: Univ. Potsdam, 2010.

[8] B. T. Y. Widemann and M. Lepper, "Paisley: Pattern matching carte," in *Proc. Int. Conf. Theory Pract. Model Transformation*. Berlin, Germany: Springer, 2012, pp. 240–247.

[9] Y. Solodkyy, G. Dos Reis, and B. Stroustrup, "Open pattern matching for C++," in *Proc. 12nd Int. Conf. Generative Program., Concepts Exper.*, 2013, pp. 33–42.

[10] S. Ryu, C. Park, and G. L. Steele, Jr., "Adding pattern matching to existing object-oriented languages," in *Proc. ACM SIGPLAN Found. Object-Oriented Lang. Workshop*, vol. 5, 2010, pp. 1–11.

[11] R. Muschevici, A. Potanin, E. Tempero, and J. Noble, "Multiple dispatch in practice," *ACM SIGPLAN Notices*, vol. 43, no. 10, pp. 563–582, Oct. 2008.

[12] C. Chambers, "Object-oriented multi-methods in cecil," in *Proc. Eur. Conf. Object-Oriented Program.* Berlin, Germany: Springer, 1992, pp. 33–56.

[13] M. Ernst, C. Kaplan, and C. Chambers, "Predicate dispatching: A unified theory of dispatch," in *Proc. Eur. Conf. Object-Oriented Program.* Berlin, Germany: Springer, 1998, pp. 186–211.

[14] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers, "MultiJava: Design rationale, compiler implementation, and applications," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 3, pp. 517–575, May 2006.

[15] J. Liu and A. C. Myers, "JMatch: Iterable abstract pattern matching for Java," in *Practical Aspects Declarative Languages*, V. Dahl and P. Wadler, Eds. Berlin, Germany: Springer, 2003, pp. 110–127.

[16] B. Emir, M. Odersky, and J. Williams, "Matching objects with patterns," in *Proc. Eur. Conf. Object-Oriented Program.* Berlin, Germany: Springer, 2007, pp. 273–298.

[17] T. Kohn, G. Van Rossum, G. B. Bucher, and I. Levkivskyi, "Dynamic pattern matching with Python," in *Proc. 16th ACM SIGPLAN Int. Symp. Dyn. Lang.*, Nov. 2020, pp. 85–98.

[18] B. Bucher and G. van Rossum. (Sep. 2020). *Pep 634–Structural Pattern Matching: Specification*. Accessed: Sep. 30, 2021. [Online]. Available: https://www.python.org/dev/peps/pep-0634/

[19] T. Kohn and G. van Rossum. (Sep. 2020). *Pep 635–Structural Pattern Matching: Motivation and Rationale*. Accessed: Sep. 30, 2021. [Online]. Available: https://www.python.org/dev/peps/pep-0635/

[20] G. Bierman. (Jul. 2020). *Jep 394: Pattern Matching for Instanceof*. Accessed: Sep. 30, 2021. [Online]. Available: https://openjdk.java.net/jeps/394/

[21] Microsoft. (Jul. 2020). *What's New in C# 8.0*. Accessed: Sep. 30, 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8

[22] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*. Cambridge, MA, USA: MIT Press, 1992.

[23] R. Bornat, "Proving pointer programs in hoare logic," in *Mathematics of Program Construction*, R. Backhouse and J. N. Oliveira, Eds. Berlin, Germany: Springer, 2000, pp. 102–126.

[24] W. Ke, Z. Liu, S. Wang, and L. Zhao, "A graph-based generic type system for object-oriented programs," *Frontiers Comput. Sci.*, vol. 7, no. 1, pp. 109–134, Feb. 2013.

[25] L. Zhao, X. Liu, Z. Liu, and Z. Qiu, "Graph transformations for object-oriented refinement," *Formal Aspects Comput.*, vol. 21, nos. 1–2, pp. 103–131, Feb. 2009.

[26] L. Zhao, S. Wang, and Z. Liu, "Graph-based object-oriented Hoare logic," in *Theories of Programming and Formal Methods*. Berlin, Germany: Springer, 2013, pp. 374–393.

[27] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, May 2018, pp. 1433–1445, doi: 10.1145/3183713.3190657.

[28] Q. Tong, "Mapping object-oriented database models into RDF(S)," *IEEE Access*, vol. 6, pp. 47125–47130, 2018.

[29] J. Li, Y. Cao, and S. Ma, "Relaxing graph pattern matching with explanations," in *Proc. Conf. Inf. Knowl. Manage.*, New York, NY, USA, 2017, pp. 1677–1686, doi: 10.1145/3132847.3132992.

[30] S. Ghilezan, "Strong normalization and typability with intersection types," *Notre Dame J. Formal Log.*, vol. 37, no. 1, pp. 44–52, Jan. 1996.

WEI KE (Member, IEEE) received the Ph.D. degree from the School of Computer Science and Engineering, Beihang University. He is currently an Associate Professor with the Computing Program, Macao Polytechnic Institute. His research interests include programming languages, image processing, computer graphics, and tool support for object-oriented and component-based engineering and systems. His research interests also include the design and implementation of open platforms for applications of computer graphics and pattern recognition, including programming tools, environments, and frameworks.

KA-HOU CHAN (Member, IEEE) received the bachelor's degree in computer science from the Macau University of Science and Technology, in 2009, and the master's degree from the Faculty of Science and Technology, University of Macau, in 2015. He is currently pursuing the Ph.D. degree with the School of Applied Sciences, Macao Polytechnic Institute. His research interests include algorithm analysis and optimization, parallel computing, video coding, neural networks, and computer graphics.