Multi-Language Code Search

by Varot Premtoon

S.B. M.I.T., 2018

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2019

© 2019 Massachusetts Institute of Technology. All rights reserved.

Author: ______________________________
Department of Electrical Engineering and Computer Science
May 22, 2019

Certified by: ______________________________
Armando Solar-Lezama
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor
May 22, 2019

Accepted by: ______________________________
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Multi-Language Code Search

by Varot Premtoon
Submitted to the Department of Electrical Engineering and Computer Science

May 22, 2019
In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Searching for specified code patterns is a common component in many types of programming tools for any language. Unfortunately, there is no code search approach that produces application-friendly search results and supports multiple languages under the same system. This work presents a program representation for multi-language code search called Yograph, which allows languages to share a common representation for the same computation while retaining language-specific information. To bridge syntactic variations in and across languages, a single Yograph can be augmented to represent many equivalent programs and high-level abstractions using equality rules. We also present Yogo, a code search tool for Java and Python that implements Yograph and outputs search results as detailed pointers to AST nodes. Our evaluation shows that, in both languages, Yogo can search for realistic patterns in realistic programs and find matches that look different or are mixed with unrelated code but ultimately perform the same computation.

Acknowledgements

I remember meeting with my adviser, Armando Solar-Lezama, over a year ago with a pile of incoherent thoughts for a project. From that, he saw the potential in solving multi-language code search. He taught me how to do research, both in this field and in general, and patiently guided me until the completion of this work. The financial support he sought for me allowed me to focus on the project. I'm extremely grateful for his vision, advice, and support.

I'm no less deeply indebted to my technical mentor James Koppel. When I first met Jimmy two years ago, I asked if he would be willing to guide and work closely with me. Since then, he has spent hundreds of hour teaching me about programming languages, hearing my ideas, and keeping me on track. His commitment to mentoring gave me confidence to solve difficult problems. Jimmy has many direct and indirect contributions to Yogo, for which I am very thankful.

I also owe a lifetime of thanks to my parents Natee and Weena Premtoon. My mom retired recently, and her full-time job has been praying for the completion of this thesis. She said that my dad once had a dream of me holding an award from NASA. I have never thought of joining the space industry, but this was several years ago when going to school in the U.S. still felt like a distant dream. It was also in the last few months of his life. He believed in me, that I would go far and do great things. I miss him. To you both, thank you for the lifetime of nurture and support. I feel honored to put your names on this page.

Special thanks to my sister Pui for taking care of everyone at home so I could be doing this.

Molly, thank you for your personal support, words of encouragement, and for reminding me to put down my work and have fun every once in a while. Most importantly, thank you for your firm belief in me even when I'm not at my best.

This work is the most challenging project I have accomplished. But it is a milestone that I didn't reach all by myself. Many along the way have extended their helping hands, wise words, and much needed company, some not named here. To them, I owe my deepest gratitude.

Contents

1 Introduction **8**
1.1 Motivating Example and Requirements . . . 8
1.2 The Yogo Search Tool . . . 10
1.3 Contributions . . . 11

2 The Yograph Representation **13**
2.1 The Base Yograph . . . 13
2.1.1 Basic Expressions . . . 14
2.1.2 Memory State and Assignment . . . 15
2.1.3 Array and Object . . . 17
2.1.4 Function Call . . . 17
2.1.5 Conditional and Loop . . . 18
2.2 Representing Paraphrases . . . 20
2.3 Abstraction . . . 23
2.4 Multi-Language Representation . . . 25
2.4.1 Designing Language-Generic Types . . . 25
2.4.2 Designing Language-Specific Types . . . 26
2.4.3 Abstraction Again . . . 27

3 Matching **29**
3.1 Match Pattern . . . 30
3.2 Type of Search . . . 33
3.3 Independence Constraint . . . 36
3.4 Invariance Constraint . . . 37
3.5 Multiple Bindings . . . 38

4 Implementation **40**
4.1 Domain Specific Language . . . 41
4.1.1 Node Type Definition . . . 41
4.1.2 Search Pattern . . . 42
4.1.3 Equality Rule . . . 44
4.1.4 Trigger . . . 44
4.2 Translator . . . 45
4.3 Matcher . . . 47
4.3.1 Representing Yograph . . . 48
4.3.2 Interpreting Rules . . . 49
4.3.3 Asserting Equalities . . . 50
4.4 Loop Depth and Invariance . . . 52
4.5 Independence . . . 54

5 Evaluation **55**
5.1 Evaluation 1: Searching in Real-World Codebases . . . 55
5.1.1 Codebases . . . 55

5.1.2 Patterns 56
5.1.3 Defining Rules 59
5.1.4 Preprocessing LitiEngine 60
5.1.5 Timeout 61
5.1.6 Results 61
5.2 Evaluation 2: Synthetic Fragments 68

6 Related Work **70**
6.1 Code Search 70
6.2 Program Representation 71

7 Conclusion **73**

A Appendix: Search Patterns **76**

List of Figures

1 An example of refactoring in Python. 9
2 Variations over the array frequency count pattern, with different types of loops and irrelevant statements. 9
3 Java example where we want to replace code in (a) and (b) with (c) 10
4 An example Yograph of an expression calculating the squared distance between (1, 4) and (2, 2). $\mathsf{const}_v()$ is written as v and binop_{op} as op. 14
5 An xample of assignments. 16
6 A Yograph for the array frequency count code in Figure 2b. Some nodes are duplicated in the illustration for readability, with superscripts to identify them. Only memory states are shown as loop or conditional nodes. 19
7 An example of a for-each loop is represented. The loop variable `a` is assigned the value of the $\mathsf{foreachV}_l$. The subgraph representing the function call `use(a)` is abbreviated. 20
8 An example of equality rule applications to equate a basic while-loop counter with a higher-level iteration concept. Starting with the Yograph in Box 1, where only memory has been translated to a loop node, applying Rules ASMT-1 and LOOP-DIST results in the subgraph in Box 2 added to the graph. Dotted lines connect nodes that are equivalent. Box 3 is a redrawing of Box 2 for clarity. Applying Rule COUNTER-ITERV then results in the subgraph in Box 4. 22
9 The organization of the Yogo Search Tool and its deployment. The system admin maintains a long-term library of rules and custom types (1), which are reused in every search session (4). Then for each search session, the end-user provides source files (2) and search patterns (3). The tool outputs match results to the end-user (9). 40
10 Example contents of a language-generic node type definition file. 42
11 Example contents of a rule definition file. Typically search patterns are defined separately (by the end-user), but this is not required. 43
12 Excerpts from functions that matched SP2 Squared 2D distance. 64
13 Excerpts from functions that matched SP5 Time elapsed. 64
14 Excerpts from functions that matched SP7 Iterating over a dictionary. . . . 65

List of Tables

1 The number of matches found for each search pattern and codebase. 61

1 Introduction

Code search is a common component in many types programming tools for any language. This includes tools for automatic documentation generation, large-scale refactoring, program repair, and more. In any language, these tasks can be framed as looking for some code pattern then taking some action. Several proposed approaches to these problems are arguably *language-independent* [14, 15, 13, 10], where the same approach can be used to build separate tools for different languages. However, they are not *multi-language*, where a single tool works for multiple languages. The tool development effort still has to be duplicated. Towards the vision of building truly multi-language tools, this work focuses on code search, the common sub-problem. We propose a code search approach that is designed to be an upstream process for other applications, not just for human eyes, and is multi-language. These are goals we have not seen attempted before.

1.1 Motivating Example and Requirements

To understand the requirements of a multi-language code search tool, consider refactoring loops that count the number of times a value occurs in a collection into a concise “count” function call. Figure 1 shows an example in Python. In a large project, we would want a tool to search for all instances of manual frequency count and replace those instances with a simple call to `count`. The search component of the tool will need to run on source files, not execution events or binaries. The search results need to point out what in the source file represent the array, the loop, and the counter variable, among other things, so that the refactoring tool knows what to replace. Outputting a general region or “snippet” of matched code is not sufficient.

The tool should also find *paraphrases* and *discontiguous matches*. *Paraphrases* are different ways of doing the same thing, through different naming, syntactic variation, or different approaches altogether. A *discontiguous match* is a match where the statements that match

```
1 count = 0
2 for a in arr:
3   if a == k:
4     count += 1
5 use(count)
```

(a) Pattern to look for

```
1 count = arr.count(k)
2 use(count)
```

(b) Substitution

Figure 1: An example of refactoring in Python.

```
1 count = 0
2 for i in
      range(len(arr)):
3   if arr[i] == k:
4     count += 1
5 use(count)
```

(a)

```
1 count = 0
2 i = 0
3 while i < len(arr):
4   if arr[i] == k:
5     count += 1
6   i += 1
7 use(count)
```

(b)

```
1 count = 0
2 for i in
      range(len(arr)):
3   if debug:
4     print(arr[i])
5   if arr[i] == k:
6     count += 1
7 use(count)
```

(c)

Figure 2: Variations over the array frequency count pattern, with different types of loops and irrelevant statements.

our pattern are interleaved with irrelevant statements. Figure 2 displays different Python programs that should match our refactoring search pattern.

As it turns out, Java code could use a similar tool. Figure 3 shows the Java analogies of array element frequency count. To support both languages, the search tool's ability to handle paraphrases must stretch to handling variations *across* languages without rebuilding a separate infrastructure for every language. A multi-language code search tool can be used with a multi-language transformation framework such as the Cubix system [9] to efficiently create a single multi-language tool.

Examples of refactoring across languages like this are not uncommon, as people carry bad habits or lack of API knowledge to different projects. An update in APIs with clients in multiple languages sometimes result in similar-looking changes in those languages. Besides, even though complex search patterns tend to be more language-specific, they sometimes depend on simpler, generic concepts like this example.

```java
int count = 0;
for (SomeType x : list)
  if (x == k)
    count += 1;
use(count);
```

(a) Java loop counting 'k' in a list

```java
int count = 0;
for (int i = 0; i < list.size(); i++)
  if (list.get(i) == k)
    count += 1;
use(count);
```

(b) Variation of the same logic

```java
int count = Collections.frequency(list, k);
use(count);
```

(c) Target replacement

Figure 3: Java example where we want to replace code in (a) and (b) with (c)

1.2 The Yogo Search Tool

We present Yogo ("You Only Grep Once"), a multi-language source-level code search tool that semantically matches code paraphrases across languages and discontiguous statements, while producing search results that detail the correspondences between the pieces of the user's search pattern and the matched AST nodes. To our knowledge, no previous work has attempted all of these requirements. Previous semantic code search engines [6, 2, 12, 8] often rely on natural language cues, which give inherently vague semantics. They also tend to output code snippets. Other systems do not match on source code [11] or are not robust to syntax variations [13]. Most importantly, none of them is designed to be multi-language.

At the heart of Yogo is the Yograph ("Your Only Graph") multi-language program representation designed for semantic search. Yograph is influenced by E-graphs, which are a family of data structures first introduced by Downey et al. [4] and popularized in applications by Nelson [7, 3]. As value representations, E-graphs are naturally robust against statement ordering and interleaving. Equality analyses efficiently augment an E-graph with many equivalent programs. Earlier work on E-graphs focus mostly on representing expressions. Our Yograph specifically builds on the E-PEG representation of Tate et al. [17], which is an application of E-graphs on optimization that has support for language constructs such as loops and memory. The downside is that none of the previous representations capture paraphrases across languages or large surface variations. (e.g., using two different libraries

to do the same work). It is also unclear how high-level language features are represented without compiling down to bytecode, which makes existing techniques not suitable for a source-level search.

To address these limitations, the first key idea is to define a coherent system of multi-language node types ("operators" in E-PEG terms) that faithfully captures source information. Yograph node type library contains both language-generic and language-specific types that work together. Equality rules that reason about paraphrases over language-generic types are automatically shared among languages, while other rules make language-specific reasoning. This consequently lowers the per-language development cost without losing language-specific information.

The other key idea is abstraction. Our key to bridging the surface variations is to define abstractions such as "iterate over an array" as a Yograph node type. Then, a system of equality rules is defined to map different implementations from possible different languages into the same abstraction. The set of node types and rules is extensible over time. Abstractions can build on top of one another and enjoy the combinatorial effect in the number of implementations they represent. This is how a single search query, if described in terms of high-level abstractions, can match many different programs.

1.3 Contributions

- The Yograph representation. We present the core components of this multi-language representation, its system of node types, and how equality rules and abstraction allow a single Yograph to represent paraphrases. We also discuss semantic matching on Yograph and what additional primitives beyond graph matching are needed for search.

- Yogo, a working prototype of a multi-language code search tool based on Yograph. We describe the implementation of the tool and the domain specific language for users to define search patterns, rules, and abstractions.

- An evaluation of Yogo. We provide an working set of rules, abstractions, and search patterns to show how Yogo can be used. We evaluate Yogo along with our rule set on real-world Java and Python codebases as well as on synthetic test cases. Our evaluation shows that Yogo is able to search for realistic patterns in realistic programs and find matches that contain paraphrases, discontiguity, and are quite precise. The experience of using it shows that Yogo rules and abstractions are reusable and should make the tool easier to use over time. We also analyze the limitations of our approach.

2 The Yograph Representation

At the very core of our approach is Yograph, a multi-language program representation. Our high-level principle is to have a rich multi-language program representation encapsulates its semantics and as many variations as possible. (This is contrary to the FaCoY approach [8], where most of the effort is in expanding the search query, not the program representation.) However, despite the goal of being semantic, Yograph needs to be reasonably faithful to the source program's syntax because the match result on Yograph needs to trace back to the source AST for downstream applications. This means that Yograph must not over-normalize syntactic variations in or across languages. Lastly, the multi-language representation should allow languages to share resources to lower the marginal cost of supporting new languages

Yograph builds on the E-PEG of Tate et al. [17]. We refine its operator set to be suitable for multiple and possibly high-level languages. We also integrate the notion of abstraction, where node types do not always correspond to concrete operations but are sometimes high-level specifications with possible implementations in many languages. Abstraction is the key idea in bridging very different multi-language paraphrases.

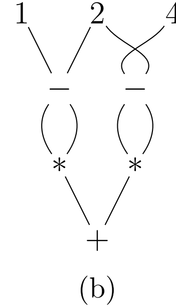
In the following subsections, we begin by describing in somewhat language-generic terms the Base Yograph, which is simplification of Yograph that does not represent equality (as PEG is to E-PEG). After that we explain the modifications that make it Yograph, along with how equality rules and abstractions work. Lastly we explain how this language-generic foundation can be grounded on multiple real programming languages.

2.1 The Base Yograph

The Base Yograph is an expression graph where each node is an operator that may have some attributes and take other nodes as input. A node is a functional representation of a value corresponding to some program expression, memory, or other facts about the program. Below are the outlines of how different program constructs translate to Yograph node types.

```
1  (1-2)*(1-2) + (4-2)*(4-2)
```

(a)



(b)

Figure 4: An example Yograph of an expression calculating the squared distance between (1, 4) and (2, 2). $\mathsf{const}_v()$ is written as v and binop_{op} as op.

These are not exhaustive and are only meant to give a general idea. We use the notation $\mathsf{nodeType}_v(x, y, z, ...)$ to represent a node of type $\mathsf{nodeType}$ with attribute v and arguments $x, y, z, ...$

2.1.1 Basic Expressions

A constant v is represented as $\mathsf{const}_v()$, which takes no argument. Unary and binary operations are represented by binop and unop types whose attribute is the operator and take the operand nodes as input. The expression $1 + (-2)$ is translated as:

$$\mathsf{binop}_{(+)}(\mathsf{const}_1(), \mathsf{unop}_{(-)}(\mathsf{const}_2())).$$

For brevity, we will write $\mathsf{const}_v()$ as v and $\mathsf{binop}_{op}/\mathsf{unop}_{op}$ as op from this point on. Figure 4 contains an example translation of an expression calculating the squared euclidean distance between $(1, 4)$ and $(2, 2)$. Our illustration convention is that nodes take input from the top edges and send output to the bottom edges. Input order matters, while all output edges from a node carry the same value.

Every node in Yograph is fully determined by its type and ordered inputs. The expression $1 + 1$, although written linearly as $+(1, 1)$, only contains one constant node that is the argument to both operand ports of the binary operator $+$. Nodes that evaluate to the same value may be considered distinct; $+(1, 0)$ and $+(0, 1)$ are distinct.

2.1.2 Memory State and Assignment

Node types in Yograph fall into four categories: *memory*, *lvalue*, *rvalue*, and *memory-value pair*. A memory node evaluates to a memory state, which is modeled as a function from addresses to values. An lvalue node evaluates to a memory address. An rvalue node evaluates to a value, as in the range of the memory state function. *const* and *binop* nodes are both rvalues. A memory-value pair node evaluates to a $\langle$memory state, value$\rangle$ pair and generally represents a program operation that has a return value and also changes the program state, such as a function call.

With these categories, Yograph represent memory and memory operations explicitly as nodes. Variables are modeled as identifiers that represent memory addresses, which can be queried against a memory state to retrieve their values.

Each program begins with a genesis memory state represented as a memory node σ_0. A program variable `x` is first translated as $\mathsf{ident}_{\mathrm{x}}()$, where ident is an lvalue node type for identifiers whose attribute is the identifier name. The memory read operator $\mathsf{Q}(\sigma, \lambda)$ (Q for "query") is an rvalue type that represents value stored in memory σ at the address of the lvalue λ. For instance, a program expression `x + y` translates to:

$$+(\mathsf{Q}(\sigma, \mathsf{ident}_{\mathrm{x}}()), \mathsf{Q}(\sigma, \mathsf{ident}_{\mathrm{y}}()))$$

where σ is the program's memory state at the time the expression is evaluated. For brevity, we will be writing $\mathsf{ident}_{\mathrm{x}}()$ as $\mathbf{x}$ (in bold).

Yograph represents assignments with the node type $\mathsf{assign}(\sigma, \lambda, x)$, whose arguments are a memory, an lvalue, and an rvalue, respectively. The type assign is a memory-value pair type. Following C/C++ convention, the memory part represents the new memory state after the assignment, and the value part is equal to input the rvalue. Types $\mathsf{mem}(x)$ and $\mathsf{val}(x)$ represent the memory and the value parts of a memory-pair node x. Here are some basic

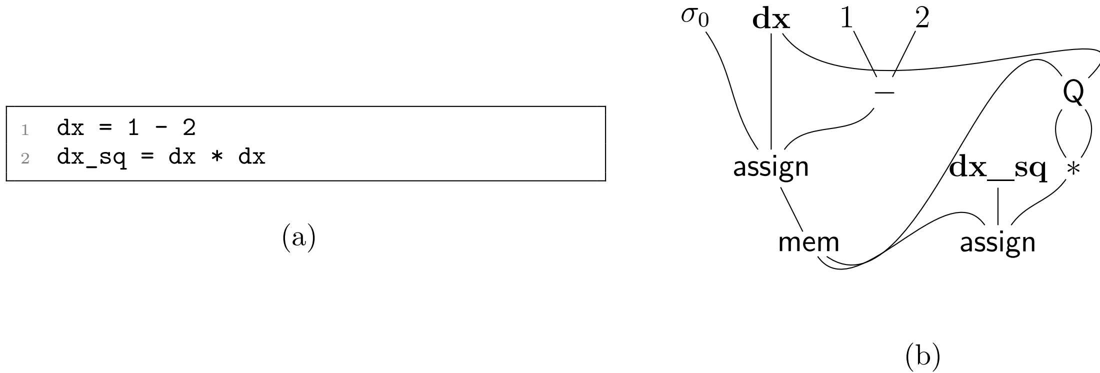


Figure 5: An xample of assignments.

equality rules of an assignment:

$$\mathsf{Q}(\mathsf{mem}(\mathsf{assign}(\sigma, \lambda, x)), \lambda) \Rightarrow x \qquad \text{(ASMT-1)}$$

$$\mathsf{val}(\mathsf{assign}(\sigma, \lambda, x)) \Rightarrow x \qquad \text{(ASMT-2)}$$

We will talk about equality rules in more details soon. For now $LHS \Rightarrow RHS$ means that the nodes described by the patterns in LHS and RHS evaluate to the same value. The first rule states that the memory read after assignment should reflect the assigned value. The second rule simply describes the return value of the assignment as an expression. Figure 5 shows an example of Yograph assignments.

Representing variables and assignments explicitly as opposed to resolving all identifier values during translation time makes Yograph more faithful to the source. Variable names and assignments are now searchable. An example search pattern involving assignments is the three-line swap: `t = a; a = b; b = t`. By resolving identifiers to values at translation time, we lose the ability to match such a pattern. It also allows for uniform treatment of assignments into other types of lvalue, such as array cells, object fields, and pointer references.

2.1.3 Array and Object

Yograph's memory model is explicit about object references. When a Java variable `obj` represents a heap object, its corresponding memory read $\mathsf{Q}(\sigma, \mathbf{obj})$ evaluates to the object's reference. To get the actual object, that reference needs to be queried against a memory again. Given that a reference is an rvalue, we define the $\mathsf{at}(ref)$ lvalue wrapper type that takes a reference and evaluates to the address of the object. The object pointed to by `obj` is represented as $\mathsf{Q}(\sigma, \mathsf{at}(\mathsf{Q}(\sigma, \mathbf{obj})))$. Whole-object queries with at are not frequent in the initial Yograph of a program, since most languages pass around objects by reference most of the time, but are used more by equality rules, e.g., to reason about arrays.

Accessing an array cell uses the lvalue selector type $\mathsf{sel}(obj, key)$, where obj and key are the array reference and the array index. The sel node evaluates to the memory address of that particular array element. As an example, the program expression `-a[1]` is translated as: $-(\mathsf{Q}(\sigma, \mathsf{sel}(\mathsf{Q}(\sigma, \mathbf{a}), 1)))$. Maps/dictionaries work identically. Object field access uses a slightly different node type $\mathsf{dot}(obj, field)$ where $field$ must be an identifier. Assignment into an array cell or object field works the same way as variable assignment, taking a $\mathsf{sel}/\mathsf{dot}$ node instead of an identifier for its lvalue input. Box 4 of Figure 6 contains an example for array access.

2.1.4 Function Call

The $\mathsf{fcall}(\sigma, f, fargs)$ type represents a function call. σ is the memory right before the function call. f is the function expression. $fargs$ is the argument list represented by type $\mathsf{fargs}(x, xs)$, a the cons-style list of function arguments, with x being an argument and xs being either an fargs node or an empty $\varnothing$ node. Function and variable expressions in Yograph are quite similar: named-functions are identifiers queried against a memory state. This allows support for first-class function languages where functions can be reassigned, passed around, or computed from some expressions. The fcall node evaluates to a memory-value pair, with the memory part representing the state after the function execution and the value part

representing the function's return value. Box 3 of Figure 6 shows an example of a function call.

2.1.5 Conditional and Loop

We represent conditionals with $\mathsf{cond}(p, t, f)$ and loop with $\mathsf{loop}_l(init, next)$, which are similar to ϕ and θ in PEG. The attribute l identifies the loop represented. Figure 6 shows an example Yograph for the array count example from Figure 2b. In the figure, the cond and loop in the figure only represent memory states, while values of variables like `i` and `count` are still Q nodes. These Q nodes, reading from conditional or loop-varying memory states, are themselves conditional/loop-varying and are equivalent to some cond/loop nodes that will be added to the graph once we apply equality rules.

The loop node loop runs for infinitely many iterations. To terminate a loop, the $\mathsf{final}_l(p, \theta)$ operator takes in a loop condition and a loop node, which are both loop-varying values, and returns the value of the loop node at the first iteration that p becomes false. In Figure 6, the final node represents the memory state when the loop terminates.

Representing for-each loops need two other constructs: $\mathsf{foreachV}_l(\sigma, src)$ and $\mathsf{foreachP}_l(\sigma, src)$. foreachV takes in the memory right before the loop and the reference to a generator, such as a list, and evaluates to a loop sequence iterating over that generator, followed by undefined values infinitely. Its attribute identifies the loop represented. foreachP is similar except it evaluates to True for as long as the generator is live, then False infinitely. We model for-each loops by using loop and final to represent memory states as before. In addition, the loop variable is assigned to the value of foreachV at the beginning of each loop iteration, and the loop is terminate by feeding the foreachP sequence to the final node. See Figure 7 for a simple example.

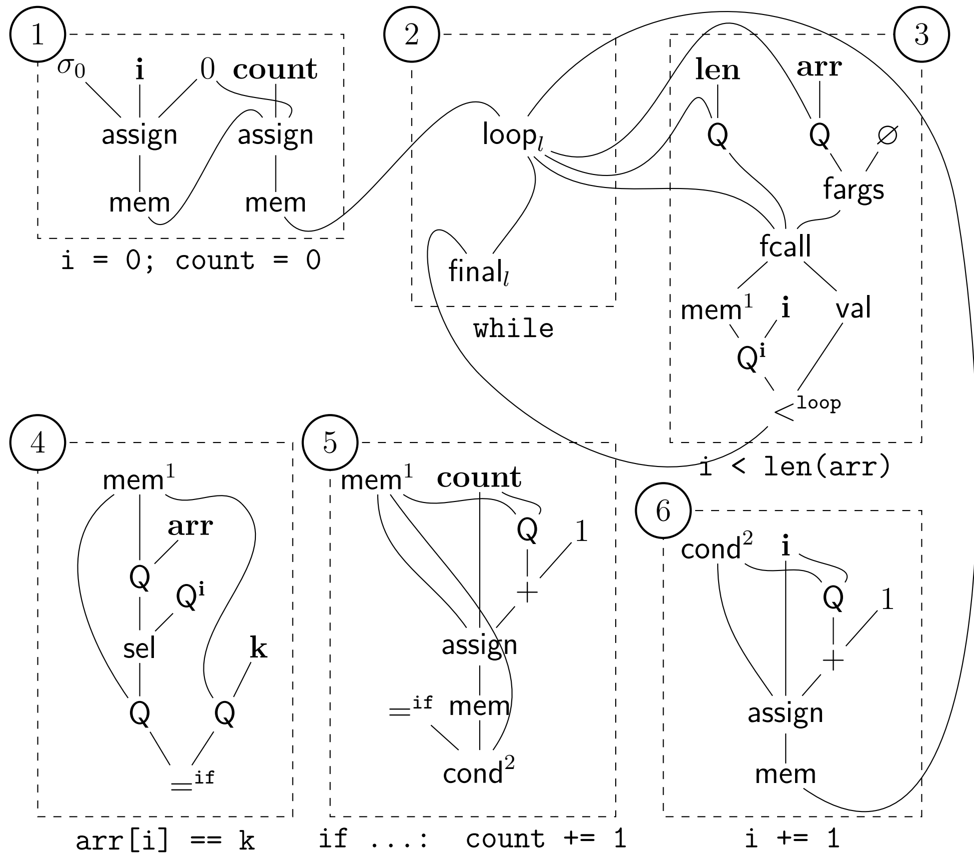


Figure 6: A Yograph for the array frequency count code in Figure 2b. Some nodes are duplicated in the illustration for readability, with superscripts to identify them. Only memory states are shown as loop or conditional nodes.

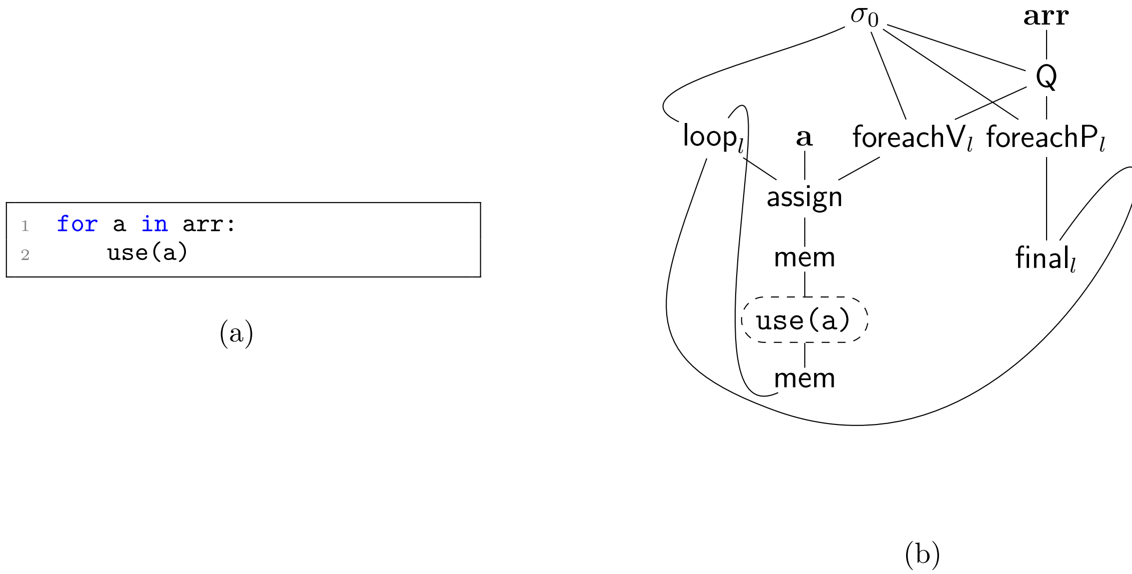


Figure 7: An example of a for-each loop is represented. The loop variable `a` is assigned the value of the $\mathsf{foreachV}_l$. The subgraph representing the function call `use(a)` is abbreviated.

2.2 Representing Paraphrases

The Base Yograph only represents one program at a time. Consider these three Python snippets: (1) `use(0)`, (2) `use(1 * 0)` and (3) `arg = 0; use(arg)`. They correspond to three different Base Yographs. However, if a user searches for a `use` function call with `0` as the argument, all of them ought to match. We need to improve our representations for these three programs to reflect their common semantics.

We follow the techniques proposed for E-PEG [17] by using *equality rules* to reason how the same value is expressed in different ways. Given the equality analysis, Yograph then puts nodes that are equal or equivalent (we use both terms interchangeably) into the same *equivalence classes*. Unlike in Base Yograph, Yograph nodes take in equivalence classes, not nodes, as inputs.

An equality rule is of the form $LHS \Rightarrow RHS$, where LHS and RHS describe graph expressions that are equivalent. Rules are applied over a Yograph. When an LHS matches in the graph, the pattern in the RHS is added to the graph, and the root nodes of the LHS and RHS are asserted to be equal. In snippet (2), the nodes $*(1, 0)$ and 0 initially belong

to two separate equivalence classes. Applying the rule $*(x, 0) \Rightarrow 0$ merges them into one. This means that the node for `use` function calls now takes in an equivalent class with 0 as a member, and will be matched against patterns looking for `use(0)`. Applying rule ASMT-1 from Section 2.1.2 to snippet (3) similarly puts the memory read of `arg` after the assignment (i.e., in `use(arg)`) in the same class as `0`, allowing `use(arg)` to match as well. Equality rules are applied repeatedly until the Yograph stops changing.. This rule application process effectively augments a Yograph with more and more paraphrases.

It is important to point out that equality rules do not really "rewrite" a Yograph. They do not replace the LHS with the RHS. Rather, the RHS is added to the existing graph and made equivalent to the LHS. Our use of the word "rewrite" later in this document refers to this non-destructive process.

We now consider some use cases of equality rules in Yograph to get a sense of how they bridge paraphrases. In a deployment of a Yograph-based tool, a combination of expert users and end-users are responsible for writing the rule system. We will make a clearer distinction in Section 4 on implementation. For now, we discuss in general terms how rules can be used.

The most basic use of equality rules is algebraic equalities, such as $+(a, b) \Rightarrow +(b, a)$ or $not(not(p)) \Rightarrow p$. Our use here is not fundamentally different from [17], and won't be discussed further.

More interestingly, equality rules are used to reason about memory reads with respect to conditionals and while-loops. Yogo translator, described in Section 4, only generates cond and loop nodes for memory states. When Q node reads from a loop state or a conditional state, we want to generate loop and cond nodes to represent the memory read value as well. The following equality rules distribute memory reads to generate control nodes for such values:

$$\mathsf{Q}((p, t, f), \lambda) \Rightarrow \mathsf{cond}(p, \mathsf{Q}(t, \lambda), \mathsf{Q}(f, \lambda)) \quad \text{(COND-DIST)}$$

$$\mathsf{Q}(\mathsf{loop}_l(init, next), \lambda) \Rightarrow \mathsf{loop}_l(\mathsf{Q}(init, \lambda), \mathsf{Q}(next, \lambda)) \quad \text{(LOOP-DIST)}$$

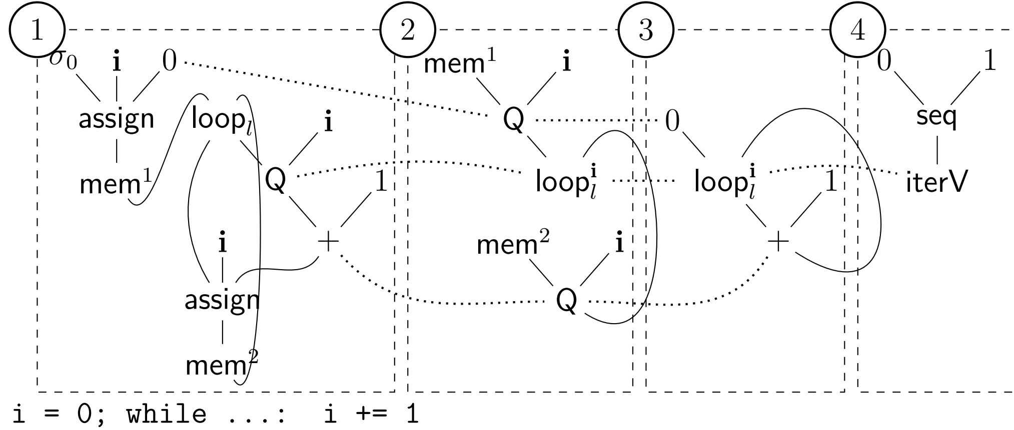


Figure 8: An example of equality rule applications to equate a basic while-loop counter with a higher-level iteration concept. Starting with the Yograph in Box 1, where only memory has been translated to a loop node, applying Rules ASMT-1 and LOOP-DIST results in the subgraph in Box 2 added to the graph. Dotted lines connect nodes that are equivalent. Box 3 is a redrawing of Box 2 for clarity. Applying Rule COUNTER-ITERV then results in the subgraph in Box 4.

In Figure 8, the transition from Box 1 to 3 is a result of applying Rules ASMT-1 and LOOP-DIST. The subgraph in Box 3 is considerably more abstract Box 1 as it encodes any concrete implementations that produce loop value starting with 0 and incrementing by 1. It does not specify a variable name, or that a variable is involved in the computation at all. Thus, equality rules essentially augment the initial graph in Box 1 with several more paraphrases. The same idea can be applied to the example in Figure 6 to populate the graph with `loop` and `cond` nodes for `i`, `count`, `arr[i]`, etc.

A consequence of the above distribution rules is that memory reads of lvalues that are not relevant in the loop or the conditional block also turn into conditional/loop nodes as well. For instance, we cannot find a match for `use(0)` in this snippet: `arg = 0; <if-blocks>; use(arg)` even when `<if-blocks>` does not change `arg`. The following rule connects the chain of equality analysis across conditional boundaries:

$$\text{cond}(_, x, x) \Rightarrow x \qquad \text{(COND-SAME)}$$

Solving the similar problem with loops requires a rule to the effect of "if a memory read for an lvalue does not change inside a loop body, then it is equal to the memory read before and after the loop." Writing this rule requires asserting in the LHS that some memory read is constant across some computation, which is not possible the way we currently write graph patterns. We will come back to this in Section 3.5.

2.3 Abstraction

Using equality rules to reason about math and basic control flow is not enough to bridge larger differences as in Figure 1a and Figures 2. As human, we understand that those code fragments all express different ways of iterating over an array. In other words, we have an abstract notion of iterating over an array that accepts many implementations. Abstraction is the key ingredient to bridging paraphrases.

Users of our approach are allowed to define custom node types to represent abstractions with their own specifications, and then define rules to rewrite concrete implementations to such abstractions. This way, different Yographs that compute the same value get augmented with the same abstraction nodes. This is in the same spirit as building a *cliché* library in the Programmer's Apprentics project [14], although based on a very different representation.

For example, take the abstract notion of iterating over the integers [0, 1, 2, ...]. This computation is present in all of Figures 2 and should be represented the same way. To that end, the user defines an abstract type $\mathsf{seq}(a, k)$ to be an rvalue type that evaluates to an infinite list of numbers $[a, a+k, ...]$. The user also defines $\mathsf{iterV}(src)$ and $\mathsf{iterP}(src)$ to be new rvalue types that represent looping over a collection src. They are analogous to foreachV and foreachP from Section 2.1.5, except instead of taking in a memory and a collection reference as inputs, $\mathsf{iter}*$ takes a whole, possibly abstract, collection. With these definitions, the expression $\mathsf{iterV}(\mathsf{seq}(0, 1))$ represents iterating over integers [0, 1, 2, ...].

After that, the user writes equality rules to equate less-abstract constructs to more-

abstract constructs. In this example, the user should write:

$$i \leftarrow \mathsf{loop}_l(a, +(i, k)) \Rightarrow \mathsf{iterV}_l(\mathsf{seq}(a, k)) \quad \text{(COUNTER-ITERV)}$$

$$\mathsf{foreachV}_l(_, \mathsf{val}(\mathsf{fcall}(_, \mathsf{Q}(_, \mathbf{range}), \mathsf{fargs}(_, \varnothing)))) \Rightarrow \mathsf{iterV}_l(\mathsf{seq}(0, 1)) \quad \text{(PY-RANGE-1)}$$

Rule COUNTER-ITERV gives the already-abstract loop value another level of abstraction. Figure 8 shows an example of this abstraction climbing taking advantage of Rule COUNTER-ITERV. In that example we start with a very low-level memory read-based graph going up to the iterV construct. Rule PY-RANGE-1 is specific to Python. It asserts that iterating over the return value of a range function call with one argument is equivalent to $\mathsf{iterV}(\mathsf{seq}(0, 1))$. Because Yograph loop nodes run to infinity, we need to extrapolate beyond the actual program return value of `range`. Such extrapolations make our abstraction possible but must be made considering the transitivity and symmetry of equivalence. These rules bring all of Figures 2 to have the same loop counter representation.

Other abstractions follow the same process. The user decides on relevant abstract concepts, which may build on top of one another. Then the user devises a rule system to rewrite less abstract constructs into more abstract constructs. New abstractions should have semantics that are compatible with existing types. For example, we designed iterV to output value like that of foreachV, and therefore the two constructs can be made equivalent by rules. seq is an rvalue representing an abstract collection that is compatible as an input to iterV. A new type should fall into one of the type categories (memory, lvalue, rvalue, and memory-value pair).

Climbing up the abstraction hierarchy means encoding more and more paraphrases. In the next section, we show that abstraction is so powerful that bridge differences not only in a language, but also across languages as well.

2.4 Multi-Language Representation

So far, we have introduced Yograph without grounding it much on multiple languages. This section explains what it takes to instantiate Yograph to support multiple real-world languages. We will draw specific examples from Python and Java. Remember that our goal is to keep Yograph as faithful to the source as we can, while still minimizing the per-language effort both to develop and to use tools based on Yograph.

Our principle to striking this balance is to identify language constructs that are common among languages and language constructs that are specific to a language, then represent them both accordingly. As a result, at the most basic level, Yograph has a library of node types such that some represent language-generic computations and some language-specific. A Yograph translation for a Java program would use some of the language-generic types and some of the Java-specific types. With that foundation, the user extends the Yograph type library with node types that represent higher level language-generic abstractions.

This organization saves effort. On the part of the user, some rules can be reused among languages. With a new language, the user focuses on writing rules that make language-specific reasoning and rules that rewrite language-specific constructs into language-generic ones. We also claim that this organization saves the engineering effort to develop a translator from program to Yograph, although we defer the argument to Section 4.

Below we discuss the considerations going into designing language-generic and language-specific types, followed by how abstraction again brings paraphrases together, now from multiple languages.

2.4.1 Designing Language-Generic Types

Deciding what to make language-generic is a tricky choice to make. After all, few things are truly the equivalent between two languages. Something as simple as numerical addition works differently in Python and Java when integer overflow is considered. However, if everything is to be made language-specific, Yograph would be a pretty unhelpful multi-language

representation.

We do not have a one-size-fits-all solution to this. Instead, we draw the line between generic/specific types where we think makes sense for most search applications. This results in the types presented in Section 2.1 being all language-generic. We believe that these types represent concepts that cover a large portion of many languages, enough to be useful in allowing different languages to share resources. At the same time, they represent the language features with enough resolution; e.g., it is fine for most search applications that we not capture the language difference between Java and Python for identifiers, additions, function calls, etc.

2.4.2 Designing Language-Specific Types

Now that a large part of a language is covered by generic types, the rest of the features need to be either represented by some language specific types or somehow converted into generic types types. This design decision affects how faithful the representation is to the source. Our approach is to devise specific types to capture the source as close as possible and then write equality rules reason about them. Take for example the python list assignment:

```
1   [a, b] = [1, 2]
```

We could let translator implementation unpack this into two separate assignments and therefore only need the language-generic types. However, we decide to design python-specific types $\mathsf{pyListLV}(v, vs)$ and $\mathsf{pyList}(x, xs)$ to represent unpackable lvalue and rvalue, respectively. (These are cons-list where the first argument is an element and the second is recursively another list or $\varnothing$.) These two types plug into the argument ports of assign like any lvalue and rvalue. This representation allows the Yograph to be relatively close to the source. A search pattern or an equality rule can target python list assignments specifically.

Much like custom abstraction node type, language specific types need to interoperate well with the rest of the representation. For one, they should fall into the existing categories (memory, lvalue, rvalue, or memory-value pair). pyListLV, for instance, is an lvalue and

can be plugged in wherever an lvalue is expected. Each new type should also have some specification of what it does. We can think of pyListLV as evaluating to some special memory address that is tied to the addresses of its arguments (`a` and `b` in the above example). Its partial specification is that the memory state after a list assignment is the same as if the assignment was unpacked into multiple consecutive assignments. To capture this, a python rule set might include this rule:

$$\mathsf{mem}(\mathsf{assign}(\sigma,\mathsf{pyListLV}(v, vs), \mathsf{pyList}(x, xs))) \Rightarrow \mathsf{mem}(\mathsf{assign}(\mathsf{mem}(\mathsf{assign}(\sigma, v, x)), vs, xs)) \quad \text{(PY-ASMT-UNPACK)}$$

In our current design, the Python specific types include types for list, dictionary, keyword arguments, and more. We also have a python-specific binary operation type that takes in a memory state along with the operands. This is used when the an operand is a reference to non-primitive objects. The translator implementation heuristically decides when this is the case. For Java, it turns out that the generic types cover most of the language features we currently want to support. We have a Java-specific type for `instanceOf`.

2.4.3 Abstraction Again

This organization of node types means that programs of any language are represented uniformly; Python and Java Yographs have the same structure. The last step toward bridging cross-language paraphrases is simply to apply the concept of abstraction on this representation. In this part, we refer back to the array frequency count loop example to show how abstraction works cross-languages.

As it stands, our discussion in Section 2.3 on abstraction is enough to get us a common representation for loop index, i.e., iterating over [0, 1, 2, ...], which is present in the Python programs in Figures 2 and in Java program in Figure 3b. This is because Java and Python use the same language-generic loop-related types. The next step is to have a common rep-

resentation for iterating over an array, where array is an abstract concept that encompasses Python and Java lists.

For that, the idea is to define an abstract rvalue type $\mathsf{get}(obj, key)$ to represent array element access. get looks like sel from Section 2.1.3, except it takes an actual array object (not object reference) and is a rvalue representing the array element (not lvalue). Then, we write the following language specific rules to bridge Python and Java list access through this abstraction.

$$\mathsf{Q}(\sigma, \mathsf{sel}(obj, key)) \Rightarrow \mathsf{get}(\mathsf{Q}(\sigma, \mathsf{at}(obj)), key) \qquad (\textsc{py-get})$$

$$\mathsf{val}(\texttt{list.get(key)}) \Rightarrow \mathsf{get}(\mathsf{Q}(\sigma, \mathsf{at}(\texttt{list})), \texttt{key}) \qquad (\textsc{java-get})$$

Rule PY-GET matches against Q-based object indexed access and rewrites that to a get-access. Notice that it uses at to fetch the object from the reference obj. (This rule is arguably language generic, although for now it's useful for Python examples.) Rule JAVA-GET, which we abbreviated, rewrites the value of a `get` function call to a get-access, similarly using at to fetch the object from a reference. Now the loop index and the get-access are both abstracted. We finally write a language-generic rule to finish our array-iteration abstraction:

$$\mathsf{get}(obj, \mathsf{iterV}_l(\mathsf{seq}(0, 1))) \Rightarrow \mathsf{iterV}_l(obj) \qquad (\textsc{array-iter})$$

With that, the search pattern for the array frequency count pattern is language-generic. Broadly speaking, such a pattern looks for some counter that increments when the array iterator is equal to some target value.

In summary, our multi-language approach is a combination of two powerful ideas. One is to represent languages under a consistent infrastructure and allow them to share resources while retaining their differences. The other is to use abstraction to bind multi-language paraphrases under the same representation. Good abstractions are reusable and composable, which makes them an efficient tool to build a large library of paraphrases.

3 Matching

This section details what matching is in a Yograph search tool, what sort of search Yograph is designed to support, and how patterns are written. We do not discuss matching algorithms here, as that is covered in Section 4 on the implementation.

Matching happens in two contexts. One is to serve an end-user's search query. That is, now that the program is represented in Yograph, the search tool has to find pieces that match the pattern the user is looking for. The other context is for rule execution. Each equality rule has a LHS pattern that has to match against a sub-Yograph to trigger the RHS.

These two types of matching are very similar. An end-user specifies the what they look for in a program with a *match pattern*. A match pattern, roughly speaking, is a pattern over Yograph along with some additional constraints. An equality rule's LHS is also a match pattern, although the ones we have shown do not have constraints. To avoid confusion, we will call a match pattern written as an end-user search query (rather than a rule LHS) a *search pattern*. We refer to *code pattern* or *program pattern* as a general description over the program source independent of Yograph, such as "a print statement" or "a function call inside an if-block". In a search application, the user has a general program pattern they want to look for, which they translate into a search pattern to be matched on Yograph.

Section 3.1 gives a more formal description of a structure and semantics of a match pattern, which we have been using rather informally. Section 3.2 characterizes what kinds of search can be expressed as a Yograph search pattern. The goal is to give end-users an intuition for what search applications Yograph is designed for. The rest of the sections propose additional match pattern constraints beyond graph expressions and why they are needed.

3.1 Match Pattern

So far we have been informally describing match patterns as expressions, like the LHS of these rules:

$$\mathsf{Q}(\mathsf{mem}(\mathsf{assign}(\sigma, \lambda, x)), \lambda) \Rightarrow x \qquad (\textsc{asmt-1})$$

$$\mathsf{cond}(_, x, x) \Rightarrow x \qquad (\textsc{cond-same})$$

Now we would like to give a more formal description of what a match pattern is. For that, we start by describing *node pattern*. A node pattern is a pattern describing a Yograph node. A node pattern has one of the two forms:

- Free Form: $\mathsf{nodeType}_{attribute}(arg1, arg2, ...)$
- Binding Form: $var \leftarrow \mathsf{nodeType}_{attribute}(arg1, arg2, ...)$

The attribute part is only applicable to types that have one. The attribute in a node pattern is either a value, a wildcard (_), or a variable. A wildcard matches any attribute value. A variable matches like a wildcard and is bound to the value it matches; later when we have multiple node patterns, this binding behavior enforces that the same variable matches the same attribute value. The arguments can be wildcards or variables. Analogous to the attribute, a wildcard matches any equivalence class, and a variable is bound to the equivalence class it matches. In a binding form node pattern, the variable *var* is also bound to the equivalence class containing the matched node. As we will soon see, this is allows us to write two node patterns where one is an argument to the other.

Here are some example node patterns and what they match, provided that the variables in each pattern are not constrained by other node patterns yet:

- $\mathsf{const}_0()$ matches only $\mathsf{const}_0()$.
- $\mathsf{const}_()$ matches any const node.

- $\mathsf{const}_a()$ matches any const node and binds a to the matched attribute
- $\mathsf{binop}_+(_, _)$ matches any addition binop node.
- $\mathsf{binop}_+(a, b)$ matches any addition binop node and binds a and b to the equivalence classes of the matched node's arguments.
- $\mathsf{binop}_+(a, a)$ matches any addition node whose two arguments are equivalent and binds a to the class of the arguments.
- $c \leftarrow \mathsf{binop}_+(a, b)$ is similar to $\mathsf{binop}_+(a, b)$ but also binds c to the equivalence class of the matched node.
- $a \leftarrow \mathsf{binop}_+(a, b)$ matches any addition whose value is equivalent to its first argument. For example, this is expected to match $+(1, 0)$ if there is a rule for addition identity. Binds a and b to the classes of the arguments.
- $\mathsf{Q}(_, _)$ matches any Q node. There is no attribute for Q.

A *match pattern* is a list of node patterns optionally followed by *constraints*:

nodePattern1, nodePattern2, ..., constraint1, constraint2, ...

We will ignore constraints until Section 3.3. For now, a match pattern just describes a list of Yograph nodes and how they are related. This very basic match pattern only has one node pattern, which matches every const node in the graph:

$$\mathsf{const_}()$$

Now consider this match pattern with three node patterns:

$$\mathsf{const_}(), \mathsf{const_}(), \mathsf{const_}()$$

This matches three, not necessarily distinct const nodes in Yograph. If the Yograph has N different const nodes, there will be N^3 matches. Usually a useful match pattern does not look for completely independent nodes like these, but rather nodes that are connected to form a subgraph. This is expressed using variables, which are enforced across the match pattern. This following pattern searches for an addition of some constant and zero, which is described as three node patterns connected by variables:

$$\mathsf{binop}_+(a, b), a \leftarrow \mathsf{const_}(), b \leftarrow \mathsf{const}_0()$$

As a matter of convenience, we adapt our notation so that node patterns can be inlined to form a nested expression. The pattern above is identical to the following pattern:

$$\mathsf{binop}_+(a \leftarrow \mathsf{const_}(), b \leftarrow \mathsf{const}_0())$$

Still a valid match pattern, and still consists of three node patterns, although we introduce the syntax sugar to write them as one expression. Finally, since we have inlined node patterns like that, the binding variable do not look useful anymore. Again as a syntax sugar, the binding forms can become free forms like so:

$$\mathsf{binop}_+(\mathsf{const_}(), \mathsf{const}_0())$$

This makes the binding variables implicit, but the match pattern still matches the same nodes as in the previous two patterns.

Now we can formally explain the LHS of an equality rule like $\mathsf{Q}(\mathsf{mem}(\mathsf{assign}(\sigma, \lambda, x)), \lambda) \Rightarrow x$, which is a match pattern with three node patterns in it for Q, mem, and assign, inlined into one nested expression. There is no requirement that a match pattern has to boil down to one expression. Totally disjoint node patterns, like the example pattern with three $\mathsf{const_}()$ patterns, are also allowed.

A note on equality rule notation. For an equality rule, when there are multiple possibly nested node patterns in the LHS match pattern, the top-level of the first expression is considered the root node of the LHS. This is the root that is asserted equivalent to the RHS.

The RHS pattern is different from the LHS pattern in that it describes nodes to be inserted rather than to be matched. As such, there are no wildcards. The variables in the RHS get their values from the LHS.

3.2 Type of Search

We do not claim that Yograph is a suitable representation for all types of code search applications. Given how match pattern works, this section characterizes what kinds of program pattern Yograph is designed to search. This will help draw a line for our evaluation, as well as help the end-user decide whether Yograph is the right approach for their search requirements.

Yograph matches semantically. In this work, semantic matching is to search for program values that can be described in a functional representation, i.e., some operators applied to some arguments. The syntax of a match pattern in Section 3.1 reflects this philosophy. This principle translates to Yograph being good at finding what the program computes not how the program computes something or what the program looks like.

With semantic matching, searching for values that can be described as a mathematical expression is natural. To search for squared euclidean distance, the end user can write search pattern: $+(pow(-(x1, x2), 2), pow(-(y1, y2), 2))$. (Binop nodes are abbreviated to their operator attribute.) With a good rule system, this pattern matches various programs that compute this expression, whether in a single program expression or in multiple statements to the same effect (e.g., with intermediate variables for subcomponents), or with self-multiplication in place of power-of-two.

The concept of "what the program computes" extends beyond mathematical expressions to include computations with function calls, conditionals, loop constructs, and everything

encoded as a combination of Yograph node types. The user can look for “a loop index”, the loop variable that starts with 0 and increments in each iteration, with a search pattern $i \leftarrow \mathsf{loop}_l(0, +(i, 1))$.

The distinction between finding what the program computes and not what the program looks like comes from the fact that Yograph does not encode strict textual or syntactic information. There is no way to search for “two assignments in a row, forbidding anything at all in between” or “a variable declaration as the first statement of a loop”. A search for a simple constant 0 may yield not only literal `0` in the program but also other expressions known to be equal to zero, such as `x * 0` and even `len([])`, depending on what rules are in place. If the user has a strict requirement on the surface form of the match results, the user will have to filter them against the AST outside of Yograph.

The distinction between finding what the program computes and not how the program computes something is exemplified with the following example. Suppose a user wants to look for something like this:

```
1  if p:
2    <some-code-1>
3    count += 1
4    <some-code-2>
5  else:
6    <some-code-3>
```

A user may want to describe this as some conditional event or action: “if `p` is true then increment `count`”. The Yograph representation, being a value representation, is not suitable to describe events or actions. The user has to look for some functional expressions as a proxy. The most straightforward one in this case is $\mathsf{cond}(p, +(x, 1), x)$. The search is framed as as looking for a value that is x if p is true or $x + 1$ otherwise. The pattern will match the code snippet if and only if it has proof that `<some-code-1/2/3>` do not affect the value of `count`. There is no obvious way to search for the conditional increment without regards to what happens around it.

Often times actions come in form of function calls, which can be matched. The pro-

gram pattern "make a directory d" translates to a search pattern looking for a function call like `os.mkdir(d)`. The limitation on action searching becomes apparent when context is involved. Suppose a user wants to find "make a directory d if it does not exist". A possible proxy is to look for the memory state:

$$\mathsf{cond}(\mathsf{fileExists}(\sigma, d), \sigma, \mathsf{mem}(\texttt{os.mkdir(}\sigma\texttt{, } d\texttt{)}))$$

which is a conditional state that evaluates to σ if d does not exist at state σ or, otherwise, the state after the `os.mkdir` function call. While this is precise, it yields false negatives if there are other state-changing operations in the if-block.

Due to our approach's reliance on abstraction to bridge paraphrases, the approach works well when the end-user writes a search pattern of the right size and has the right level of abstraction. If the search pattern is too small, such as finding for "make a directory d" which is a single function call node pattern, the user may not see immediate benefits over using grep or AST traversal search. A search pattern enjoys more potential matches if it is composed of multiple abstractions.

On the other hand, if the search pattern is large but contains mostly low-level or concrete node types, then there are likely many more paraphrases that the pattern does not capture. The issue at hand here is scalability. Larger search patterns represent longer code patterns, which likely have more paraphrases. The way to capture a large code pattern is to break it down into meaningful logical chunks, define an abstraction type for each chunk, write rules for each abstraction, then write a search pattern based on those abstractions. If a search pattern contains three abstractions, each with ten possible paraphrases, then the pattern encodes a thousand paraphrases. Plus, these abstractions may be useful for other match patterns. But if a search pattern is composed of only low-level types, each representing few concrete programs, then the pattern only captures a small space of possible paraphrases.

Another fundamental limitation is that Yograph is intraprocedural. It cannot reason

about code that spans across multiple function definitions.

Although we have framed the explanation of this section as relevant to search pattern, the very same principles and limitations apply to what match pattern we can write for equality rules.

Yograph and the matching semantics as we have explained give us a powerful framework to do multi-language code search. However, this turns out to not be enough. The following sections show examples of useful program patterns Yograph still miss and outline the additional matching primitives necessary to match them.

3.3 Independence Constraint

Suppose we have this program fragment: `arg = 0; x = y; use(arg)`, and our search pattern describes a `use` function call with argument `0`. To reason that `arg` remains `0` after `x = y`, we need a rule to say that if there is an assignment into lvalue λ, and that lvalue λ' is not related to λ, then λ' memory read is unchanged across that assignment. The condition that λ and λ' are not related cannot be expressed as node patterns.

We need a special *constraint* in the LHS match pattern. A constraint is an additional expression in a match pattern enforcing a property of its arguments. In this case, we propose the *independent* constraint:

$$\textsc{independent}(\lambda, \lambda')$$

The constraint checks that an assignment into lvalue λ does not affect the memory read of lvalue λ'. We say that λ is independent from λ'. λ and λ' are variables that must be used before in a node pattern. For example, different identifiers (e.g., `arg` and `x`) are independent from one another. Object identifier `obj` is independent from `obj.field` because assigning to a field does not change the object's reference, but not the other way around.

Now we can write another assignment rule:

$$\mathsf{Q}(\mathsf{mem}(\mathsf{assign}(\sigma, \lambda, x)), \lambda'), \textsc{independent}(\lambda, \lambda') \Rightarrow \mathsf{Q}(\sigma, \lambda) \qquad (\textsc{asmt-3})$$

With this rule, intervening assignments no longer affect the memory reads of unrelated lvalues. Similar rules can be written about what memory reads other memory operations affect. The rule set we use for our evaluation has such a rule for function calls, for example. These rules make it possible to find discontiguous matches across irrelevant statements, as in the fragment `arg = 0; x = y; use(arg)`

A complete and sound implementation of INDEPENDENT is challenging and out of our current scope, as it will take a more advanced analysis than we have. This is a well-carved out space for future works. We provide a heuristic implementation in Section 4.5.

3.4 Invariance Constraint

Recall Rule COUNTER-ITERV from Section 2.3:

$$i \leftarrow \mathsf{loop}_l(a, +(i, k)) \Rightarrow \mathsf{iterV}_l(\mathsf{seq}(a, k)) \qquad (\textsc{counter-iterv})$$

This rule is meant to recognize a loop variable that begins with a and increments by k to be equivalent to an iterator over the fixed-stride sequence $[a, a + k, ...]$. As it stands, this rule is not precise. Consider this Java fragment:

```
1   for (int i = 0; i < n; i += f()) { use(i) }
```

In this example, the loop variable `i` increments by `f()`, which could be different for each loop iteration. Therefore, `i` does not represent an iterator over a fixed-stride sequence. We need to check that the pattern should only match if the increment value remain constant across loop iterations.

We propose the invariance constraint:

$$\textsc{invariant}(l, v)$$

which checks that the value of v does not vary with the iteration of loop l. v is a variable bound to some equivalence class by some node pattern. l is a variable bound to some loop identity, which the attribute of loop-related types (e.g., loop_l).

In the semantic matching framework we have discussed, this is arguably a "how the program computes something" case, which violates our notion of semantics. However, we ran into several examples where checking this is important, so we decided to make an exception.

The improved version of Rule COUNTER-ITERV is:

$$i \leftarrow \mathsf{loop}_l(a, +(i, k)), \textsc{invariant}(l, k) \Rightarrow \mathsf{iterV}_l(\mathsf{seq}(a, k)) \qquad (\textsc{counter-iterv})$$

The invariance constraint does not enable more paraphrases to be matched. Rather, it makes matches more precise.

3.5 Multiple Bindings

Now suppose we have this program fragment: `arg = 0; while(p) {..}; use(arg)`, where the loop does not change the value of `arg`, and we are searching for program pattern `use(0)`. The value of `arg` as the argument to `use(arg)` is represented as a memory read of lvalue `arg` at the memory state right after the loop. Currently, we have no rule to carry the assigned value `0` of `arg` through the loop and out the end.

For now we will pretend that we do not have INVARIANT. We need a rule that says "if a memory read of an lvalue does not change inside a loop, then its value is the same as before and after the loop." More specifically, if there is a loop state $\sigma^{loop} \leftarrow \mathsf{loop}_l(\sigma^{init}, \sigma^{next})$, and the memory reads $\mathsf{Q}(\sigma^{loop}, \lambda)$ is equivalent to $\mathsf{Q}(\sigma^{next}, \lambda)$, then the memory read is also equivalent to $\mathsf{Q}(\sigma^{init}, \lambda)$ and $\mathsf{Q}(\mathsf{final}_l(_, \sigma^{loop}), \lambda)$. Writing the LHS requires checking that

$\mathsf{Q}(\sigma^{loop}, \lambda) = \mathsf{Q}(\sigma^{next}, \lambda)$.

To support this, we allow a match pattern to have multiple binding-form node patterns that bind to the same variable. Such node patterns must match the same equivalence class. With this modification, we can write the following two equality rule (with the same LHS):

$$q \leftarrow \mathsf{Q}(\sigma^{loop}, \lambda), q \leftarrow \mathsf{Q}(\sigma^{next}, \lambda), \sigma^{loop} \leftarrow \mathsf{loop}_l(\sigma^{init}, \sigma^{next}), \sigma^{final} \leftarrow \mathsf{final}_l(_, \sigma^{loop})$$

$$\Rightarrow \mathsf{Q}(\sigma^{init}, \lambda) \qquad (\textsc{loop-const-init})$$

$$\Rightarrow \mathsf{Q}(\sigma^{final}, \lambda) \qquad (\textsc{loop-const-final})$$

These two rules together link the memory read of `arg` before, inside, and after the loop. Now we can use Yograph to find program pattern `use(0)` in the fragment `arg = 0; while(p) {..}; use(arg)`

Caring about whether an lvalue's memory read remains constant seems to violate our principle of semantic matching, since this is caring about how a program computes. An alternative may be to use INVARIANT instead of adding another exception like this. However, our implementation of invariance happens to depend on being able to reason about memory reads across a loop in the first place, so this exception is necessary at least in our implementation.

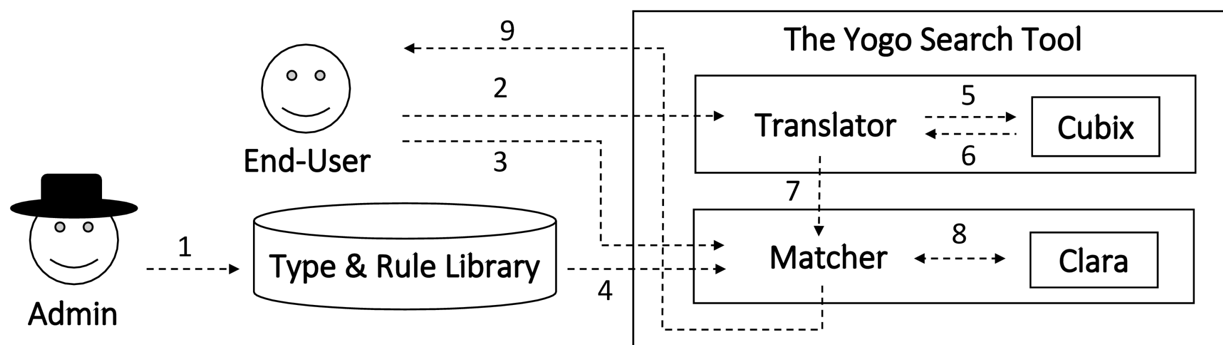


Figure 9: The organization of the Yogo Search Tool and its deployment. The system admin maintains a long-term library of rules and custom types (1), which are reused in every search session (4). Then for each search session, the end-user provides source files (2) and search patterns (3). The tool outputs match results to the end-user (9).

4 Implementation

Yogo ("You Only Grep Once") is a prototype multi-language code search tool based on the Yograph representation. Figure 9 shows the organization of Yogo and its deployment. The tool takes in source files (arrow 2 in the figure), search patterns (3), rules (4), and custom node type definitions beyond the standard types needed for translation (also 4); to the best of its ability, the tool outputs every way a search pattern can match the program (9). Each match is a mapping from node patterns in a search pattern to the corresponding AST nodes. Yogo currently supports Python and Java.

Yogo has two types of users: the system administrator and the end-user. The admin is an expert who writes rules and designs custom abstractions (arrow label 1), which go into a long-term library of rules and types that are used in every search session. Node type definitions and rules are written in our Clojure-based DSL. Internally, Yogo does not have any rules and only has standard node types. The admin must take a bare Yogo tool and make it ready for search.

The end-user is the one who does code search. The end-user has less knowledge about languages and how to design abstractions, and only wants to quickly search for something

in a codebase. For each search session, the end-user provides the source files (2) and search patterns (3). The search patterns are also written in our Clojure-based DSL.

At the start of a deployment when the type and rule library is small, the end-user and the admin must work closely to design necessary rules and abstractions for each search. As the library gets larger over time, new search patterns should be able to reuse existing abstractions, thus the admin's work per-session grows smaller. The per-session work becomes amortized to just writing search patterns (arrow 3).

Internally, Yogo has two major components: the translator and the matcher. The translator translates source files into Yographs. The matcher is a rule engine. The matcher that takes in the Yographs, type definitions, rules, and search patterns. It saturates the Yographs with equalities and reports matches.

Section 4.1 explains the Yogo domain specific language used by the users. Sections 4.2 and 4.3 explain translator and matcher implementation of the Yogo tool. Finally, Sections 4.4 and 4.5 show how we implemented the invariance and independence constraints.

4.1 Domain Specific Language

4.1.1 Node Type Definition

Node types must be defined before they are used in a Yograph or a rule. Node type definitions come from two sources: the translator's internally defined standard node types and the admin's custom node types. Both use the same DSL. Figure 10 contains example contents of a language-generic type definition file as would be generated by the translator.

A file starts with a namespace declaration `(ns 'namespace)`. Following that, each `(defnode type [param1 param2 ...])` expression defines a node type named `type` with parameters named `param1`, `param2`, and so on. Parameter names that begin with a dollar sign are attributes, such as `$value` for `const` and `$op` for `binop`. Parameter name `$depth` is reserved for loop-related types. Other parameters are equivalence-class parameters. There can be multiple node type definition files in a search session, as long as they have different

```
1  (ns 'generic)
2
3  (defnode const [$value])
4  (defnode binop [$op arg1 arg2] )
5  (defnode mem-genesis [] )
6  (defnode mem [src]
7  (defnode cond [p t f] )
8  (defnode loop [$depth init next] )
9  (defnode final [$depth loop])
10 (defnode foreach-v [$depth src])
11 (defnode foreach-p [$depth src])
```

Figure 10: Example contents of a language-generic node type definition file.

namespaces.

4.1.2 Search Pattern

In each search session, the end-user writes one or more search patterns to tell Yogo what to find. Our DSL for search patterns mirrors the syntax for match patterns from Section 3.1 very closely. The example search pattern `distance-2d` in Figure 11 shows an example rule.

A file containing search pattern begins with a namespace declaration `(ns 'namespace)`. A search pattern is defined using `(defsearch search-name DOC? MATCH-PATTERN)`. `DOC?` is an optional documentation string. `MATCH-PATTERN` is a list of node patterns and constraints. Node patterns may be nested following the discussion on inlining node patterns in Section 3.1. A free-form node pattern is written as `(namespace/node-type arg1 arg2 ...)`. Attributes and equivalance-class arguments are written together as one argument list; their positions follow the node type definition. Each argument can be a Clojure primitive value (for attributes), a wildcard (_), a variable, or another node pattern. A binding-form node pattern is written as `(var <- free-form-node-pattern)`.

The possible constraints are: `(independent lv1 lv2)` to check that `lv2` is independent from `lv1` and `(invariant d v)` to check that the value of class `v` is invariant with respect to loop depth `d`. We will come back to loop depth later.

The Yogo match results report the AST nodes that correspond to each variable in the search pattern. This includes variables bound by binding forms and in node pattern argu-

```
1  (ns 'rules)
2
3  ;; SEARCH PATTERN
4  (defsearch distance-2d
5    "(x1-x2)^2 + (y1-y2)^2"
6    (dx <- (generic/binop :- x1 x2))
7    (dy <- (generic/binop :- y1 y2))
8    (dx-sq <- (generic/binop :** dx (generic/const 2)))
9    (dy-sq <- (generic/binop :** dy (generic/const 2)))
10   (distances-sq <- (generic/binop :+: dx-sq dy-sq)))
11
12 ;; EQUALITY RULES
13 (defeqrule cond-dist
14   "Q(cond(p, t, f), lv) => cond(p, Q(t, lv), Q(f, lv))"
15   (q <- (engine/q (generic/cond p t f) lv))
16   =>
17   (generic/cond p (engine/q t lv) (engine/q f lv)))
18
19 (defeqrule assignment-indp
20   (engine/q mem-after lvalue2)
21   (mem-after <- (generic/mem (generic/assign mem-before lvalue1 _)))
22   (independent lvalue1 lvalue2)
23   =>
24   (engine/q mem-before lvalue2))
25
26 ;; TRIGGER
27 (deftrigger loop-mem-is-mem
28   (n <- (generic/loop _ init _))
29   (custom/is-memory init)
30   =>
31   (custom/is-memory n))
32
```

Figure 11: Example contents of a rule definition file. Typically search patterns are defined separately (by the end-user), but this is not required.

ments if they represent an equivalence class. Variables in attribute positions are reported as Clojure values. Free-form node patterns will not be part of the match report.

4.1.3 Equality Rule

An equality rule is defined with `(defeqrule rule-name DOC? LHS => RHS)`. See two examples `cond-dist` and `assignment-indp` in Figure 11. Note that the memory read node type `Q` is available from a special namespace `engine` and can be is referred to as `engine/q`.

In an equality rule, `LHS` has the same syntax as the `MATCH-PATTERN` part of a search. The first, outer-most node pattern is designated to be the root node of the LHS.

`RHS` can be arbitrary Clojure expressions, with the following syntax notes. An expression `(namespace/node-type arg1 arg2 ...)` inserts a node rather than matches one. Each argument can be a Clojure value (if it is an attribute) or an equivalence class. Variables that occur in the LHS will be bound to equivalence classes and can be used in the RHS. Each node insertion in the RHS also returns the node's equivalence class, which means insertion expressions can be nested similar to node patterns in the LHS or bound into Clojure variables with `let`. Finally, the RHS must evaluate to an equivalence class, which will be asserted equivalent to the LHS root's class.

The fact that we allow equality rule RHS to be arbitrary Clojure code is more a convenience than a necessity. All of our equality rules only insert nodes, and we could have constrained the RHS to only be Yograph node expressions, although the current implementation allows us and the user to insert debug print statements during development.

4.1.4 Trigger

A trigger is a type of rule that we have not mentioned before. A trigger is defined with `(deftrigger trigger-name DOC? LHS => RHS)`, and it is very similar to an equality rule in that it has a match pattern on the LHS and arbitrary Clojure code on the RHS. The difference is that the RHS does not have to evaluate to an equivalence class, and no equality

assertion takes place.

One use of trigger to annotate the graph with some fact, usually by inserting a node whose existence means something. Trigger `loop-mem-is-mem` in Figure 11 is an example. In our Yogo setup, we define a custom type `(defnode is-memory [node])` whose presence in a Yograph asserts that its argument `node` represents a memory state. Trigger `loop-mem-is-mem` essentially says that for any `loop` node whose initial value represent a memory state, the said `loop` node also represents a memory state.

Another use of trigger is to arrange memory read nodes to exist in certain locations. This usually happens when the program pattern is a conditioned event, like "insert k to map m if m does not have key k". Part of the search pattern is a conditional object used as a proxy for the conditioned event. (Recall the discussion in Section 3.2.) However, if for any reason the program actually computes that conditional object but never uses it after the conditional block, there may not be such a conditional object in Yograph. A work-around is to write a trigger that inserts `Q` nodes in the right places so that if the computation does happen, the search pattern can find it. `Q` nodes can be inserted liberally in Yogo; ones that do not correspond to real AST nodes will just record no AST correspondences.

4.2 Translator

The first implementation choice in building the translator is to choose the source representation to translate from. We decide to use the syntax tree representation of the Cubix multi-language transformation tool [9]. Cubix defines language fragments (e.g., identifiers, function calls, assignments) as modular data types that can be "summed" to form the signature of a language [16]. Some fragments are unique to a language, while others are generic. Yograph node types are organized in a similar philosophy: a Yograph for a language uses a set of node types, some language specific and some generic. This allows for a very modular translator structured as a mapping from Cubix fragments to Yograph node types.

We develop the translator in Haskell. The translator hands the source files to Cubix

(arrow 5 in Figure 9) and gets back a Cubix tree representation (arrow 6). Translation starts at the root of the Cubix tree. At a given tree node, a translation handler is invoked based on the language fragment that node is. Most handlers recursively translate the child nodes and then emit a Yograph node. Since Yograph is intraprocedural, some handlers, such as that for method declaration, do not output a node but rather start a new graph. A single Cubix tree thus results in possibly multiple Yographs.

As the translator traverses the source tree, it keeps track of the current memory state. Variable expressions are emitted as Q nodes reading from this memory state. For conditionals and loops, the translator emits cond and loop nodes only for memory states. Variable expressions inside a loop is just a memory read against a loop state, but itself not an explicit loop node. The admin will have to write rules to generate cond and loop nodes for other program values.

The translator internally defines Yograph node types as Haskell data types similar to how Cubix defines language fragments. These are standard types that are necessary to represent a language but do not include user's custom types for higher-level abstractions like seq. Each language has a "signature", which is a sum of Yograph node types used by that language. With this organization, much of the common translation logic is reused in multiple languages. A translation for, say, assignments can be written as a function that accepts any Cubix signature with an assignment fragment and outputs any Yograph signature with assign node type.

Aside from effort saving on our end, this translation approach benefits the downstream applications that depend on Yogo. Having search results be on Cubix tree means that downstream multi-language applications that use or manipulate syntax trees are able to also leverage the Cubix representation to save their development effort, avoiding building separate infrastructures for different languages.

We have so far implemented the translator for Java and Python, languages which we find to have a good balance of similarities and differences to demonstrate our approach's

multi-language capability. The current version of the translator understands a subset of the language features and treats what it does not understand as unknown values or unknown memory operations. For example, our current implementation treats advanced control flow `break`, `continue`, and `return` as unknown memory operations. A better source-to-graph translator is needed to generate a Yograph that takes these control flow statements into account. As of now, Yogo cannot match patterns like “return zero”. Yogo also cannot reason about equality of memory reads across these statements.

Another limitation is that Cubix depends on external parsers that do not handle all modern language features. The Java parser cannot parse diamonds (`<>`) or lambda (`_ -> _`), and fails to translate the whole file when encountering them. Our current workaround is to preprocess Java files before feeding it to Yogo, as we do in Section 5 on evaluation

The translator outputs two sets of files to the matcher (arrow 7). One is the files containing node-type definitions, which are generated from the Haskell data types and the language signature. The other is the files containing all the translated Yographs. Each node in the output Yographs is tagged with labels that point back to the source AST nodes that generate it.

4.3 Matcher

The matcher is essentially a rule engine that keeps applying the rules to the Yograph until saturation, and reports matches in the end. The matcher is largely language-agnostic; most of what it has to know about a language comes from the node type definitions and the rules given to it, although it has some internal heuristics to reason about memory, discussed in Section 4.5.

Our matcher is built around Clara, an open-source Clojure implementation of a RETE rule engine [5]. A Clara *session* contains *productions*, which are $LHS \Rightarrow LHS$ rules similar to Yogo triggers, where LHS is patterns over Clara *facts* (or *working memory elements*, in RETE’s terms) and RHS is arbitrary code to be executed when a match is found. A fact

is a key-value map, with one of the keys denoting the fact type. A simple fact asserting a whether condition might be:

```
1       {:fact-type :weather, :location ‘‘Boston’’, :temperature 68}
```

Having initiated a session with production rules, a Clara user inserts facts into the session and tells Clara to fire the rules. Activated rules may insert or retract facts and cause more rule activation. Clara terminates when the session is saturated.

With this framework, we define the matcher session as a thick wrapper around the Clara session, with the addition of various sidecar data structures and utility functions. At a high-level, the matcher interprets search patterns, equality rules, and triggers (arrows 3 and 4) into Clara production rules. The matcher also turns Yographs (from arrow 7) into Clara facts.

From that point, the matcher and Clara works closely together (two-way arrow 8). The production rules go into building a Clara session, and the Clara facts are inserted to the session. Clara production rule RHS is written in terms of callbacks to the Yogo matcher session, often to insert new nodes and assert equivalences. The matcher interprets these calls, updates its own state, and inserts appropriate facts into the Clara session. This circular process continues until saturation or timeout.

The following sections explains in more details how the matcher handles Yographs and rules in a matcher session.

4.3.1 Representing Yograph

Each Yograph node is turned into a Clara fact. A fact contains what node type it represents, the arguments, and also the ID of the equivalence class the node currently belongs to. A Yograph expression $+(1, 2)$ are represented as three Clara facts:

```
1       {:fact-type :generic/const, :value 1, :eid ’e.1}
2       {:fact-type :generic/const, :value 2, :eid ’e.2}
3       {:fact-type :generic/binop, :op :+, :arg1 ’e.1, :arg2 ’e.2, :eid ’e.3}
```

The parameter names are derived from the node type definitions (Figure 10), save for

`:eid`, which is a special parameter that denotes the node's equivalence class. Arguments that are EIDs indicate taking equivalence classes as input. Note that there is no direct edge between nodes. If the class `'e.1` has other members, then they are all interchangeable as argument `:arg1` to the `binop` node.

The translator gives a Yograph to the matcher (arrow 7) using an internal DSL that describes a sequence of node insertions like this:

```
1    (defgraph addition
2       ...
3       ;; let eid1 be the EID of const(1), which originates from an AST
           node labeled 187
4       eid1 (generic/const 1 [187])
5       eid2 (generic/const 2 [193])
6       eid3 (generic/binop :+ eid1 eid2 [214])
7       ...)
```

For each line, the matcher invokes a node insertion function, which generates an EID for the node, adds the corresponding fact into the Clara session, and returns the EID. The function uses hash-consing to avoid duplicate nodes. The matcher session also maintains a map from each EID to a set of *occurrences*, which in this case are AST labels. The occurrence (e.g., `[187]`) is added to the node's equivalence class regardless of whether the node is a duplicate. Note that Yogo saves no direct connection between nodes and their occurrences: whenever equivalence classes are merged by an equality rule, their occurrences are pooled together, and the occurrences of individual nodes inside a class are not tracked. This is in line with our philosophy that Yogo searches for what is computed, not how it is computed. If an equivalence class contains 0 and $*(0, x)$, a match for 0 will match both.

4.3.2 Interpreting Rules

The matcher interprets rules written in Yogo DSL into Clara production rules. Clara's rule language is also based on Clojure which allows us to take advantage of Clojure metaprogramming. On the LHS, the rule interpreter turns positional argument-based Yograph expressions into Clara fact patterns, which are based on named arguments, by consulting the node type

definitions. Nested node patterns are flattened into a list of fact patterns connected by EID binding variables.

On the RHS, Yograph node patterns are converted into node insertion function calls. For equality rules, the rule interpreter generates a function call to the matcher that asserts the equality between the EIDs of LHS and RHS roots.

For search patterns, the interpreter generates an RHS that records matches in a set maintained by the matcher session. A match is a map from named variables in the search pattern to either their corresponding EIDs in Yograph or some Clojure primitive values. At the end of the rule-firing, these matches are reported back to the user (arrow 9). In the report, the matcher expand each EID into its corresponding set of occurrences to denote what AST nodes it points to. As a result, the end-user sees a mapping from search pattern variables to AST node labels.

4.3.3 Asserting Equalities

Clara does not have a concept of classes or equivalence between facts. Keeping track of and updating equivalance classes are done by the matcher session. When two nodes are asserted equivalent by a rule, one of the two equivalence classes will be merged into the other. This means all the node facts in the former class will need to have their `:eid` attribute somehow changed to the latter class. The former class will become obsolete. All of the nodes that take the former class as arguments will need to update their arguments to the latter class. This section explains this process in details.

Information about equivalence classes are kept in the following data structures inside the matcher session but outside of the Clara session:

- `eid->nodes`: a map from each EID to the set of member nodes.
- `old-eid->eid`: a map from possibly obsolete EID to the current EID. Since equivalence classes can be merged into one another, this map indicates the "root" of those

merged-away classes. This is the same data structure as in the Quick-Find disjoint set algorithm.

- `eid->old-eids`: the reverse of `old-eid->eid` mapping each root class to classes merged into it.

- `eid->dependent-nodes`: a map from each EID to nodes that take it as argument.

When an equality assertion is made by a rule, the matcher looks up `eid->nodes` to determine which is the smaller class (call it `’eid-s`), which will get merged into the larger class (call it `’eid-l`). The merge process starts by updating `eid->old-eids` and `old-eid->eid` so that all member old-EIDs of the class `’eid-s` now point to `’eid-l`, and that the old-EIDs set of `’eid-l` includes these new old-EIDs.

The matcher proceeds to update Clara facts representing nodes in class `’eid-s` to reflect that they now belong to class `’eid-l`. Clara does not have fact updating mechanism, so the matcher has to retract and re-insert them. The matcher retracts all nodes in the small class (`eid->nodes[’eid-s]`) and re-inserts them with `:eid = ’eid-l`. The node insertion function inspects the argument of each node: for each EID argument, it looks up the `old-eid->eid` to make sure the node gets the updated EID. This is so that any argument pointing to `’eid-s` properly points to `’eid-l` instead.

Now that `’eid-s` is merged away into `’eid-l`, nodes that take `’eid-s` as arguments will need to update their arguments as well. The matcher retracts and reinserts all of the nodes in `eid->dependent-nodes[’eid-s]`.

Furthermore, since a node re-insertion may change some of the node arguments (from `’eid-s` to `’eid-l`), the updated node may collide with an existing node, causing a chain of more equality assertions. For instance, suppose there are these two Clara facts in the session:

```
1     {:fact-type :generic/sometype, :arg ’e.1, :eid ’e.3}
2     {:fact-type :generic/sometype, :arg ’e.2, :eid ’e.4}
```

and then some rule merges class `'e.1` into `'e.2`. The first fact will be retracted and re-inserted, now with argument `:arg = 'e.2`, colliding with the second fact. These two nodes initially belong to two different classes (`'e.3` and `'e.4`), but the collision reveals that these nodes – and so their classes – are in fact equivalent. The equality assertion process is a queue-based process that starts with one initial assertion, and when the node insertion function detects such a collision, more assertions are added to the queue.

4.4 Loop Depth and Invariance

Yogo provides an approximation for INVARIANT (Section 3.4) through the match pattern constraint `invariant(depth, eid)`. Instead of checking whether a value is invariant with respect to a particular loop, Yogo checks whether a value (represented by equivalence class `eid`) is constant with respect to loop of with depth `depth`. To unpack this, consider the following java example:

```
void fn() {
  while(..) { i += f(); j += 1; ... }
}
```

Suppose we would like match a loop value that increments at a fixed stride. In this case, `j` should match but `i` should not, as `f()` could return a different value each time.

Yogo's solution to this is that each loop-related type has a "loop depth" specified by a special `$depth` attribute. A node representing an outer-most loop (like the `while` loop above) has depth 1, and nested loops have increasing depths. Then, each node in Yograph is tagged with a "value depth", which is a conservative estimate on the largest depth at which its value could possibly vary. Constant like `1` never varies and thus has depth 0. A loop memory state has the same value depth as its loop depth, which is 1 in the example. The value of `f()` depends on the loop state, and therefore also has value depth 1. With these information, we know a node is invariant to a loop if its value depth is smaller than the loop's loop depth.

Loop depths are provided by the translator using the `$depth` attribute. Value depths are

inferred in a number of ways. Nodes with a `$depth` attribute have the same value depth as loop depth. Otherwise, a node's value depth is the maximum of the depths of its arguments. Nodes whose type has no EID parameters, like `const`, have value depth 0. Given a node type definition, the matcher uses these criteria to generate a Clara rule that determines its value depth. .

Since all nodes in an equivalence class have the same value, they also have the same value depth. Value depths are therefore asserted as facts about equivalence classes, not nodes. When multiple value depths are asserted to a class (because of the different rules from its member nodes), the smallest depth wins, which is sound given that value depths are conservative.

Value depths are stored as Clara facts with a special fact type. The `invariant(depth, eid)` constraint finds on the value depth fact for `eid`, and compares the value depth with `depth`. For example, here's rule COUNTER-ITERV from Section 3.4 in Yogo DSL:

```
1 (defeqrule counter-iterv
2   (i <- (generic/loop depth a (generic/binop :+ i k)))
3   (invariant depth k)
4   =>
5   (generic/iter-v depth (user/seq a k)))
```

The user can write a trigger that asserts value depth explicitly using `(s/make-value-depth eid depth)` function in the RHS. For example, the following trigger asserts that the value depth of a final node is one below its loop depth:

```
1 (deftrigger final-depth
2   (n <- (generic/final depth _ _))
3   =>
4   (s/make-loop-depth n (- depth 1)))
```

Yogo's loop depth analysis is designed with only structured control flows in mind. We do not have an approach to understand loops as a result of `goto` statements.

4.5 Independence

The user uses the `independent(lv1, lv2)` constraint to assert that assignment into lvalue `lv1` does not affect the memory read of lvalue `lv2`. (Discussion in Section 3.3.) Yogo's current implementation employs a simple set of heuristics, which naively assume that programmers tend not to use multiple aliases to refer to the same object. Suppose `lv1` and `lv2` are *simple dot lvalues*, which are lvalues made of one or more identifiers connected by dots, such as `x` and `obj.field.subfield`, here are the heuristics:

- `independent(lv1, lv2)` if `lv1` is not a prefix of `lv2`. For example, assigning to `a.b.c` may change the memory read of `a.b.c` and `a.b.c.d` but not `x`, `a.b.d`, or even `a.b`. Note that `independent(a.b.c, a.b)` is reasonable because in our memory model, a memory read of an object lvalue returns the object reference, not the actual object. Assignment into a field does not change the object's reference.

- `independent(lv1, at(Q(_, lv2)))` if `lv1` and `lv2` is not a prefix of one another. Node type `at` takes an object's reference and returns an lvalue that reads the whole object. In other words, assigning into `a.b.c` may affect the whole-object read for `a.b` and vice versa.

- `independent(sel(object, key), lv2)`. Assignment into any array or map entry does not affect the memory read of any simple dot lvalue.

The Yogo matcher interprets the `independent` constraint in a rule LHS into Clara expressions that encode these heuristics. Uses of `independent` that do not match these conditions will fail to match. These heuristics are simplistic although they work reasonably well in our evaluation. More sophisticated memory analysis is left for future works.

5 Evaluation

Our evaluation goal is (1) to test Yogo's ability to search for realistic patterns in realistic programs and find matches that contain paraphrases, discontiguity, and are correct in multiple languages, (2) to understand Yogo's limitations, both fundamental to the approach and contingent on the implementation.

To that end, we conducted two experiments. Section 5.1 describes the first evaluation, which tests Yogo's ability to perform realistic multi-language search by running Yogo on real-world codebases. Due to the rather small size of our evaluation codebases, Section 5.2 describes the second evaluation, which complements the first by testing Yogo on synthetic code fragments. In both evaluations, we analyze what Yogo finds and does not find to understand the limitations.

5.1 Evaluation 1: Searching in Real-World Codebases

The purpose of this evaluation is to characterize Yogo's ability to search realistic patterns over realistic codebases by running Yogo on real-world programs. It is also to explore what the admin and the end-user have to do to use Yogo. Below, we describe the codebases, how we design search patterns to be realistic, and different rules and heuristics that go into the our Yogo setup. We analyze the results to characterize Yogo's behavior and different failure modes. We also show that the found matches contain a variety of paraphrases and discontiguous matches in different language for our patterns.

5.1.1 Codebases

We use Yogo to search over three real-world open-source projects: PyGame, Cocos2D, and LitiEngine. All three are open-source frameworks for creating games. PyGame is written in Python with 49k LoC (commit ad681aee). Cocos2D is also written in Python with 53k LoC (commit 9bb2808). LitiEngine is written in Java with 59k LoC (commit c188504). We

choose game engines because we expect them to have plenty of meaningful intra-procedural computations, and two of the test patterns that guided our development are geometric computations, although the results on other patterns should convince the reader that our approach is not domain-specific.

5.1.2 Patterns

We use nine search patterns in this evaluation. The patterns come from two different sources. Five of the search patterns were derived from program patterns that guided the design and development of Yogo and Yograph. They were intended to touch different language constructs Yogo should support. Because we came up with these, and because they heavily influenced the design, we are aware that Yogo could be biased to perform well on them. In order to show that Yogo generalizes to new and realistic search patterns, we derive four more search patterns from Stack Overflow using the following procedure after the development:

1. Take 50 highest voted questions tagged with 'java' and 50 highest voted questions tagged with 'python.'

2. Of these questions, only consider questions that effectively ask "How to do X". This step discards questions like "Difference between wait() and sleep()" and "What does the 'yield' keyword do?".

3. Among the how-to-do-X questions, only consider ones where the operation X:

    - can be described semantically as values being computed, not what the code looks like or how the values are computed. (Section 3 describes semantic search.)
    - can be generalized to both Java and Python. This filters out language-specific questions about libraries, Android, conversions between types that do not make sense in both languages, etc.
    - is more than a single operation over the inputs in both languages. This is to avoid having a lot of search patterns that are too simple and have no structure to

highlight Yogo's multi-language capability. For instance, we ignore "how to concatenate two lists in Python" and "What's the simplest way to print a Java array". The latter is because its Python counterpart is just a single print statement.

- does not directly involve language constructs we do not support, such as breaks, list comprehensions, and lambdas.

4. Narrow down the remaining questions to those whose accepted answer contains an intraprocedural snippet that does X.

5. Rewrite X in a language-generic term (e.g., "Iterate through a HashMap" becomes "Iterating over a [language-generic] dictionary").

6. By this point, we are only left with two program patterns. We repeat Steps 2-5 for the next 25 questions of both languages (for the total of 50 additional questions) and get two more intents, for the total of four.

The search patterns are listed below. SP1-SP5 are our motivating patterns, and SP6-SP9 are patterns derived from Stack Overflow. Pretending to be an end-user, we state the *intent* of each pattern. (E.g., "SP1. Bound checking: checking whether ... bound.") This is an English description for the program pattern an end-user intends to find. We also narrate how an end-user thinks about structuring the Yogo search pattern for it and what additional abstractions and rules the end-user needs. The actual search patterns can be found in Appendix A.

- **SP1. Bound checking**: checking whether a numerical value falls out of a [a, b) bound. In mathematical notation, this is searching for the expression $or(<(x, lo), \geq (x, hi)))$, which can be written as a Yogo pattern straightforwardly.

- **SP2. Squared 2D distance**: calculating squared distance between two points in 2D space. In mathematical notation, the pattern is $+(pow(-(x1, x2), 2), pow(-(y1, y2), 2))$.

- **SP3. Map put-if-not-present**: inserting an entry into a map if the key is not present. This intent here describes a conditional event. As a proxy, we look for a map ϕ whose value is conditional on whether a key k exists in map m or not. If yes, than $\phi = m$. If not, $\phi = m[k \rightarrow v]$ for some value v. Because objects (such as maps) are manipulated by reference, the value representing the whole map object may not exist in Yograph even when the program computes it. We need a trigger to arrange appropriate memory reads of at-lvalues to aid this search pattern.

- **SP4. Array frequency count loop**: a loop iterating over a collection and counting the number of occurrences of a particular value. This is the motivating example used in the introduction. We need an abstraction for iterating over a collection, then the search pattern can describe for a node $x \leftarrow \mathsf{loop}(0, \mathsf{cond}(p, x, +(x, 1)))$ where p is a boolean expression comparing the iterator to the target value.

- **SP5. Time elapsed**: calculating difference between two program points. We define a language-generic abstraction for time read. With that, we write a search pattern that looks for a subtraction of two time reads.

- **SP6. Loop index**: a loop variable that starts with zero and increments by one in each loop iteration. This is $\mathsf{iterV}_l(\mathsf{seq}(0, 1))$, which we have discussed before.

- **SP7. Iterating over a dictionary**: a loop iterating over a dictionary's entries, accessing its keys and corresponding values. The pattern is structured as a combination of three abstractions: getting a key set from a dictionary, iterating over that key set, and accessing the corresponding values. We need will need language-specific rules for these abstractions.

- **SP8. MD5 hashing**: computing an MD5 hash of a message. We structure this pattern into three abstractions: creating an MD5 hasher, updating it with data, and getting the digest.

- **SP9. File writing**: writing data to a while. This patterns consists of creating a file-buffer and writing data to it.

5.1.3 Defining Rules

We take on the role of a system administrator to write a system of rules and abstractions necessary to support the search. Some are language-generic, such as arithmetic equality rules and equality rules that generate `loop`/`cond` nodes for values other than memory states. Some are language-specific, such as the equality rule that unpacks the value of Python `enumerate` function call or the equality rule that equate the value of Java `containsKey` function call to a language-generic "map-membership" abstraction.

Rules and abstractions are written with the search patterns to support in mind. This resembles the workflow where the admin and the end user work closely together, which is necessary in the early stage of Yogo deployment.

We anticipate that as Yogo grows its rulebase, more search patterns can be supported with few to no additional equality rules and triggers. In fact, much of the rules were written only with SP1-5 in mind. SP6-9 were added later in the development process and did require additional rules, such as rules that equate Java's `new FileOutputStream()` and Python `open()` to the same "file-buffer" abstraction. However, most rules written to support SP1-5 were useful for SP6-9 as well. The `iterV` abstraction alone is used in three patterns. This experience supports our anticipation that the marginal effort to prepare for a new search pattern should grow smaller over time.

We implement some heuristics about function calls. One is that a function may only modify its receiver argument. Any lvalue that is independent from the lvalue of the function receiver argument preserves its memory read value across the function call. For example `x` preserves its memory read across `y.f()`, since `x` is independent from `y`.

We also infer whether a function is pure based on its name. For example, Java functions named "size", "get*", or "contains*" are pure. Python functions named "strip" and "split"

are also pure. We define a custom node type `(defnode is-pure [fcall])`, which takes a function call node. The presence of an `is-pure` node indicates that the input function call is pure. We then define language-specific triggers that match on function calls, check the function name, and insert `is-pure` nodes. For instance, a trigger for Java looks similar to this:

```
(deftrigger fn-pure-heuristics
  "Java heuristics for pure functions."
  (n <- (generic/fcall _ fn _))
  (fn <- (engine/q _ fn-ident))
  (fn-ident <- (generic/ident name))
  =>
  (when (or (clojure.string/starts-with? name "get")
            (clojure.string/starts-with? name "contains")
            (clojure.string/starts-with? name "to")
            (= name "size")
            (= name "keySet")
            (= name "entrySet"))
    (custom/is-pure n)))
```

A language-generic equality rule asserts that memory after a pure function call is equivalent to the memory before the function call:

```
(defeqrule fn-pure
  "Memory is unchanged across a pure function call"
  (generic/mem (generic/fcall mem-before _ _))
  (custom/is-pure fcall)
  =>
  mem-before)
```

5.1.4 Preprocessing LitiEngine

For LitiEngine, we find that the Cubix's Java parsing is not up-to-date with the Java version used in the codebase. Cubix depends on two Java 3rd-party parsers. The primary parser hooks to an old version of JavaParser [18]. The `language-java` parser [1] then serves as a fallback. When the primary fails to parse a file (e.g., because of an unsupported language feature), a less-accurate fallback parser is used for the file. If that also fails, then Yogo ignores the file.

Because of this behavior, we write a preprocessor that puts each Java method declaration

Table 1: The number of matches found for each search pattern and codebase.

Pattern	Cocos2D	PyGame	LitiEngine
SP1. Bound checking	4	3	13
SP2. Squared 2D distance	8	8	10
SP3. Map put-if-not-present	3	4	11
SP4. Array frequency count loop	0	0	0
SP5. Time elapsed	1	20	1
SP6. Loop index	120	126	42
SP7. Iterating over a dictionary	17	6	9
SP8. MD5 hashing	1	0	0
SP9. File writing	8	7	0

into its own separate file. This is so that an instance of an unsupported feature only causes one function, rather than a whole original source file, to go to the fallback parser or fail to parse. The preprocessor works by stripping away comments, removing diamonds (`<>`, which tends to fail to parse and is ignored by the translator anyway), and then scanning for method declarations. The preprocessor finds text pattern `class ...{...}` in a source file to find a class declaration. Roughly speaking, inside a class body, it considers the text pattern `<modifier> ...(...)...{...;...}` to be a method declaration and inserts it into a dummy class in a new file. It also recursively looks for inner class declarations.

5.1.5 Timeout

A long source function generally results in a large graph, which may take the rule engine too long to saturate it with equalities. For both Java and Python, we set a time limit of three minutes per graph for Clara to execute the rules.

5.1.6 Results

Table 1 reports the number of matches that the matcher finds for each pattern and codebase. In the following paragraphs, we analyze different aspects of the results. First, we present an analysis on the correctness of the found matches to show that Yogo finds quite precise matches across languages and also to understand the reason behind some false positives. Then we

show example matches, which indeed contain various paraphrases and discontiguous matches. Lastly, we analyzes what Yogo misses (the false negatives) to understand the limitations of our approach.

Correctness We analyze the correctness of the matches at the granularity of a function. That is, a match is considered correct if the corresponding function contains code that implements the intent of the search pattern. Note that this is not the finest metric; the matcher reports the AST nodes each binding variable in the match pattern is equivalent to, and ideally we would be reporting the precision at level of AST nodes. However, verifying AST labels against the parse tree is still a laborious task under our current infrastructure. We only do this verification when the source function contains multiple potential matches to make sure each match aligns correctly.

We manually examine all found matches, except for SP6 where we sample 10 matches from each codebase. For SP3, SP5, SP6, SP7, SP8, and SP9, all found matches are correct. No matches were found for SP4 in any codebase.

For SP1, the only failure mode is that the pattern also matches when the code implements the opposite intent: checking whether a numerical value falls *inside* a bound [a, b). We found one case like this in PyGame and eight in LitiEngine. This is because some equality rules rewrite code fragment `x >= lo and x < hi` into `not (x < lo or x >= hi)`, and the search pattern matches the inner expression being negated. This inner expression is created by an equality rule and does not correspond to any AST nodes. Under the correctness criteria above, this is incorrect because nothing in the code implements the intent of the pattern, only its negation. However, the match results accurately report that the `or` node, which is the root of the matched inner expression, does not point to any AST nodes. This is an expected behavior, and the end-user is easily able to see and understand the match result

For SP2, it is worth noting that each code snippet results in two matches due to addition being commutative. More would have resulted if we had rules for associativity. In terms of

correctness, we find that four matches in PyGame (thus two snippets) actually implement 3D vector cross product rather than 2D point distance. The relevant snippets look like this, with the matched expression underlined:

```
1   cross = ((self.v1.y * v.z - self.v1.z * v.y) ** 2 +
2             (self.v1.z * v.x - self.v1.x * v.z) ** 2) +
3             (self.v1.x * v.y - self.v1.y * v.x) ** 2)
```

This highlights a downside of our approach, which is that it can be challenging to write a search pattern that precisely captures the intent. In this case, the pattern looks for any computation of the form `(x1 - x2)**2 + (y1 - y2)**2` without any constraint on `x1`, `x2`, `y1`, and `y2`. The cross product formula contains this computation. A more careful user may define abstractions and rules for 2D points and for getting their X and Y components. This requires knowing how points are usually represented and risks more false negatives.

For SP6, we found that a single match will point to the loop counter in multiple loops of the same depth. That is, this Python snippet:

```
1   for i in range(n):
2     use(i)
3   for j in range(m):
4     use(j)
```

will result in one SP6 match, with the match result pointing to both `i` and `j` as equivalent loop index. The explanation for this is that our loop node types do not remember loop identity, only its depth. Therefore, both `i` and `j` correspond to `(generic/iter-v 1 (custom/seq 0 1))` and get coalesced to the same Yograph node.

Example matches Figures 12, 13, and 14 show excerpts of functions that match SP2, SP5, and SP7, respectively. The excerpt boundaries are drawn at our discretion to concisely show general regions of code that match. This is because unlike several other search systems, Yogo does not output "snippets" or lines of code.

These examples support our claim that our approach is able to find matches in the presence of paraphrases, match discontiguity, and multiple languages.

Examples in Figures 12 contain different ways the same formula can be written. Figures

```java
double distSq = Math.pow((a.getCenterX() - b.getCenterX()), 2) +
    Math.pow((a.getCenterY() - b.getCenterY()), 2);
```

(a) LitiEngine (Java)

```java
return Math.sqrt((p1.getX() - p2.getX()) * (p1.getX() - p2.getX()) +
                 (p1.getY() - p2.getY()) * (p1.getY() - p2.getY()));
```

(b) LitiEngine (Java)

```python
xdistance = left.rect.centerx - right.rect.centerx
ydistance = left.rect.centery - right.rect.centery
distancesquared = xdistance ** 2 + ydistance ** 2
```

(c) PyGame (Python)

Figure 12: Excerpts from functions that matched SP2 Squared 2D distance.

```python
t1 = pytime.time()
poster.start()
going = True
while going:
  event_list = []
  event_list =
      event_module.get()
  for e in event_list:
    if e.type == QUIT:
      print (c.get_fps())
      poster.stop.append(1)
      going = False
    if e.type == KEYDOWN:
      if e.key == K_ESCAPE:
        print (c.get_fps())
        poster.stop.append(1)
        going = False
  if poster.done:
    print (c.get_fps())
    print (c)
    t2 = pytime.time()
    print ("total time:%s" %
        (t2 - t1))
    print ("events/second:%s"
        % (NUM_EVENTS_TO_POST /
        (t2 - t1)))
    going = False
```

(a) PyGame (Python). Lines 1, 20, 21, and 22 are directly related to the pattern. The matcher found one match, which points to both subtractions on line 21 and 22 as equivalent.

```java
while (!interrupted()) {
  // ...
  final long updateStart = System.nanoTime();
  if (this.getTimeScale() > 0) {
    ++this.totalTicks;
    this.update();
    this.executeTimedActions();
  }
  ++this.updateCount;
  final long currentMillis =
      System.currentTimeMillis();
  this.trackUpdateRate(currentMillis);
  final long lastUpdateTime = currentMillis;
  final long updateTime = (long)
      TimeUtilities.nanoToMs(
        System.nanoTime() - updateStart);
  // ...
}
```

(b) LitiEngine (Java). Lines 3 and 13 are directly related to the pattern.

Figure 13: Excerpts from functions that matched SP5 Time elapsed.

```python
for key, val in kwargs.items():
    if key in ['_use_update',
        '_time_threshold',
        '_default_layer']:
        if hasattr(self, key):
            setattr(self, key, val)
```

(a) PyGame (Python)

```python
for case in stats:
    res = stats[case]
    print("   %s:"%repr(case))
    print("       [%s, %s,
        %s,"%(repr(res[0]),
        repr(res[1]), repr(res[2])))
    print("       %s],"%repr(res[3]))
```

(b) Cocos2D (Python)

```python
for n, k in enumerate(self.tileset):
    s = pyglet.sprite.Sprite(self.tileset[k].image, y=y, x=x,
        batch=self.tiles_batch)
    # ...
```

(c) Cocos2D (Python)

```java
for (final Map.Entry<String, List<Consumer<Float>>> entry :
    this.componentReleasedConsumer.entrySet()) {
    for (final Consumer<Float> cons : entry.getValue()) {
        pad.onReleased(entry.getKey(), cons);
    }
}
```

(d) LitiEngine (Java)

Figure 14: Excerpts from functions that matched SP7 Iterating over a dictionary.

12a and 12b use explicit power-of-2, while 12b uses self-multiplication. These paraphrases are bridged by a Java-specific equality rule that rewrites the value of `Math.pow` function call to a generic power operation and a language-generic equality rule that rewrites `x * x` to `x ** 2`. Figure 12c uses intermediate variables for sub-components of the formula, but we have rules that reason about equality across assignments. These rules are hardly specific to any application. They assert semantic equivalence between small, application-agnostic paraphrases. Their combinations allow a single search pattern to cover exponentially many paraphrases.

Similarly, Figures 14 show a different way one can iterate over key-value entries of a map. Our experience with SP7 is that, although we had to add language-specific rules to understand iterating over the key set vs the entry set of a map, we reuse a lot of rules and abstractions from SP3, which also does map access, and SP4, which involves iterating over a collection. SP7 is fairly concise (see Appendix) with only 5 node patterns

Discontiguous match is best exemplified by match results for SP5 in Figures 13. In both examples, Yogo is able to find the matches despite a large interleaving chunk of code. The only way discontiguity affects Yogo is that a large code region is likelier to contain statements Yogo does not understand well (e.g, `raise` or `return`), which makes Yogo unable to reason that the variable storing the first time read (`t1` and `updateStart` in Figures 13a and 13b, respectively) remains unchanged until the difference is computed. If given a perfect memory oracle, our semantic matching approach is unaffected at all by interleaving irrelevant code.

Misses To understand the limitations of our approach, we manually try to find false negatives in the codebases, mainly using grep. For SP4 and SP8, we could not find any code that should match except for the one instance (SP8) that Yogo also found. This manual process is certainly not exhaustive. However, we did find some false positives that give us a sense of Yogo's failure modes.

Not enough translation. This refers to failures as a result of the translator, being in its prototype phase, not translating every language feature fully. This is the most frequent failure mode and causes misses in three ways. First, code that should have matched was not translated properly. Some Python file writing instances are not matched because they open a file using a with-statement, which is currently translated as an unknown statement. We found at least four such instances in PyGame and one in Cocos2D. Similarly, Java code inside a lambda is not translated.

Second, potential matches that rely on understanding advanced control flows are not matched. For example, Yogo cannot match this snippet from LitiEngine with SP3 because it lacks understanding of `return`:

```
1 if (loadedCustomEmitters.containsKey(emitterData.getName())) {
2     return loadedCustomEmitters.get(emitterData.getName());
3 }
4 loadedCustomEmitters.put(emitterData.getName(), emitterData);
5 return emitterData;
```

Third, some poorly translated language constructs inhibit equality analysis. Although

this snippet (PyGame) matches SP7:

```
1 for name,bytes in _pngsuite.items():
2     if name[3:5] not in ['n0', 'n2', 'n4', 'n6']:
3         continue
4     it = Reader(bytes=bytes)
```

the match results only point to AST nodes on line 1. The matcher does not find that `bytes` on line 4 is an iterator over the map values because `continue` on line 3 is treated still as an unknown operation, which could have done anything including reassigning `bytes`.

Not enough rules. We found code paraphrases that should have matched but there are no rules to account for them. There are two reasons for this. One is that we failed to anticipate certain paraphrases of implementations of an abstraction. The other is that rule-writing for an abstraction has diminishing returns; the first few rules cover most common paraphrases, and after that the benefits of writing more rules diminish, so we stop at some point.

For example, despite getting no match for SP9 in LitiEngine, we manually found that LitiEngine has an instance of image file writing using `ImageWriter`:

```
1  Iterator<ImageWriter> iter = null;
2  // ...
3  final ImageWriter writer = iter.next();
4  // ...
5  try (final FileImageOutputStream output = new
       FileImageOutputStream(file.getAbsoluteFile())) {
6    writer.setOutput(output);
7    final IIOImage outimage = new IIOImage(image, null, null);
8    writer.write(null, outimage, iwp);
9    writer.dispose();
10 }
```

Yogo fails to find this match for two reasons. One is that Java try-with-resources is not properly translated, but in addition to that, we do not have rules that recognize that `writer` is a "file writer". Our Java rules only account for objects created by `FileOutputStream`, `FileWriter`, and `BufferredWriter` constructors, which we believe to be the more common cases. We did not spend the effort accounting for all possible file-writing types.

Another interesting case is this 2D distance calculation in PyGame that Yogo did not find:

```
diff = self.v1 - self.v2
# ...
self.assertAlmostEqual(self.v1.distance_squared_to(self.v2),
                       diff.x * diff.x + diff.y * diff.y)
```

This technically requires interprocedural mechanism to know what the overridden subtraction does. However, this pattern of 2D point representation is arguably common enough to warrant rules that expect its behavior.

These examples illustrate a fundamental limitation of our approach: Yogo can only make inferences based on the rules it is given. Its ability to match paraphrases depend on rule writer's designing composable abstractions and accounting for as many paraphrases for each abstraction as possible.

Timeout. Recall that we set a time limit of three minutes per graph (that is, per source function) for Clara to run the rules. Some longer functions time out before their graphs are saturated with equalities. We found at least one case in PyGame where time-out causes false negatives. The function body is 110 SLoC, with four-level nested loops and a long if-elif chain. The function contains six time subtractions; only five are found.

5.2 Evaluation 2: Synthetic Fragments

Because our example codebases in the first evaluation are small, patterns SP4, SP8, and SP9 do not occur often or at all. This evaluation shows that if the patterns did occur, Yogo would find it as well as other patterns. We write synthetic code fragments that should be matches or near misses for three rare search patterns, then run a Yogo search over these tests. We make test fragments differ in ways that demonstrate Yogo's capability and limitations. Below we describe the test fragments and results for each pattern.

SP4. Array frequency count. We write test cases in both Python and Java, which include snippets in Figures 1a, 2, and 3. All of them are matched successfully. Additional test cases iterate over Python `enumerate`, contain interleaving code, or flip if-else (i.e., `if`

`p X Y` becomes `if not p Y X`.) We find misses when the interleaving code has the same problem as discussed in the previous evaluation. We also try a case where the counter variable is an array cell (not the same array as the one being counted) and a case where it is an object's field. This array-cell version results in a false negative, while the object-field version works. This is because Yogo's INDEPENDENT implementation works better when the lvalue is a simple dot lvalue. We also include test cases that should report no matches by changing small pieces of the should-match cases. A notable false positive is in a for-each loop where the array being iterated is modified in the loop. Our for-each constructs assume, incorrectly in this case, that the array is not modified.

SP8. MD5 hashing. Our test cases for both Python and Java have a similar structure. Each case first gets the hasher object (`hashlib.md5()` in Python and `MessageDigest .getInstance("MD5")` in Java). The hasher is updated, possibly many times, with some data. Then the digest function is called to get the hash. Variations of the test cases include: updating happens in a loop; for Python, the data is given to the `md5` function instead of in a separate update call; in Java, the data is an argument to the digest function; all or parts of this pattern are inside a try-catch block. Yogo is able to successfully match all of them. We include negative test cases where the hasher is assigned to null before getting the digest, or when the hasher is from a wrong algorithm (e.g., SHA-1 instead of MD5). Yogo appropriately reports no matches for these cases.

SP9. File writing. We only write Java tests for SP9, as we already see a number of Python matches in the previous evaluation. Each test has two parts: creating a file writer, and writing to it. We limit the tests to only using `FileOutputStream`, `FileWriter`, and `BufferredWriter` to create a writer, and only `write` function to write, since these are what we have written rules for. Variations include writing multiple times or in a loop, and putting all or parts of the pattern in a try-catch block (but not try-with-resources). As expected, Yogo is able to match them.

6 Related Work

6.1 Code Search

There is an abundance of studies on code search engines. Several systems use natural language processing techniques to match free-form English search queries and/or give meanings to code. Gu et al. [6] used a deep neural network that encode code snippets and natural language descriptions into a vector space such that the vector distance represents the similarity between descriptions and code. Chatterjee et al. [2] proposed SNIFF, which uses the documentation of well-documented library methods to index otherwise undocumented code that uses such methods. Mcmillan et al. [12] combined natural language processing and indexing with PageRank and spreading activation network to search and rank functions from free-form queries. Kim at al. [8] proposed code-to-code search approach FaCoY, which expands the code query to multiple queries by finding similar code snippets on StackOverflow. Their similarity is based on syntax and also natural language similarity of the Q&A posts. NLP-based approaches work well when humans consume search results. However, their correctness is difficult to reason about by applications, since matching is guided by often informal descriptions.

Outside of search engines, other systems have code search as a part of their pipelines. Martin et al. (2005) developed a Program Query Language to find a code excerpt that matches the query and perform some actions on it [11]. The primary application of PQL is error detection, where a user describes pad patterns in PQL and what actions to take when found. PQL matches on Java execution event rather than source code. This is not suitable to match control flows like loops and conditionals, which are necessary for applications like automatic documentation or refactoring. It also does not work on multiple languages.

In the refactoring landscape, LASE [13] is able locate and edit code snippets given two or more examples of methods that are manually edited. LASE takes AST edit operations that are common among examples, abstracts or ignores edits that are different, and looks

for common unchanged statements that that form the context of the edits. These edit operations and context form a *partially abstract, context-aware edit script* that describes the refactoring and where to apply it. The main drawback of LASE is that it is a syntactic approach and works directly on the AST. Its ability to abstract away details is limited to types and identifiers. As a result, LASE's search technique would not perform well across multiple languages and under syntax variation.

Highly influential to this work is the Programmer's Apprentice project [14]. The system stores a library of *clichés*, which are commonly used combinations of program elements at various abstraction levels, such as stack push operation, successive approximation, and device driver. A cliche is represented with a Plan Calculus, which is a mix of flowcharts, dataflow diagrams, and abstract data types. This representation abstracts away the syntax details of a particular language, making it language-independent. The Plan Calculus can be used to recognize cliches in code in order to generate documentation and reason about the program. However, unlike an AST, the Plan Calculus does not lend itself to refactoring. Such downstream applications must work with the source code and, if they wish to support multiple languages, cannot take advantage of the plan's language-independence.

6.2 Program Representation

The notion of augmenting a graph with an equivalence relation to represent a large number of programs was first introduced by Downey et al. [4], who proposed an algorithm to compute the congruence closure of any equivalence relation on the graph's vertices. This technique was made popular as the E-graph data structure by Nelson in applications such as theorem proving and superoptimization [7, 3]. These early works use E-graph to represent and reason about expressions and low-level operations such as load and store.

Yograph is most inspired by the PEG and E-PEG representations of Tate et al. [17]. A PEG is a graph representation where nodes represent values of program expressions, and an E-PEG is an E-graph that groups equivalent PEG nodes into the same equivalence class. E-

PEGs represent a wider range of program constructs than previous E-graphs, including loops, conditionals, and memory state. Combined with a system of equality rules, this approach is extremely suitable for encoding paraphrases and normalizing syntactic variations within a language. However, E-PEG does not have a concept of higher-level abstractions to bridge across language differences, and is not designed for multiple languages to work under the same infrastructure.

For multi-language representation, we look at the Cubix multi-language transformation framework of Koppel et al. [9]. A language consists of language-specific language fragments, some of which can be generalized to language-generic fragments. They proposed the *Incremental parametric syntax*, which allows both types of fragments to coexist in a syntax tree, thus capturing common features of languages without losing language-specific information. The concept of *sort injection* makes language-generic and language-specific fragments interoperate by asserting "is-a" relationship between fragments. For example, a Python assignment is an (generic) assignment, which is an expression, etc. The incremental parametric syntax inspired our organization of node types and abstractions. We also used the Cubix framework to develop our translator.

7 Conclusion

We have presented Yograph, a multi-language program representation designed for source-level semantic code search. Yograph's node type organization supports multiple languages under the same infrastructure, allowing them to share language-generic resources while retaining language-specific information. Yograph's equivalence system, equality rules, and abstraction pack a large number of paraphrases into one efficient representation. We have also presented Yogo, a Yograph-based multi-language code search tool which currently supports Java and Python. We evaluated Yogo on real-world open-source codebases and some synthetic code fragments. Our evaluation shows that Yogo is able to find quite precise matches that vary in how they look and are scattered among irrelevant statements but ultimately compute the same thing. Our experience confirms that Yogo's rules and abstractions are reusable and should make Yogo easier to use over time. From these results, we believe that our approach is a good foundation on which to build multi-language programming tools that depend on code search.

For future works and improvements, much of Yogo's limitations will be alleviated by continuing the development. This includes improving the translator to support more language features and writing more rules. More work is needed to translate and represent advanced control flow statements properly. The implementation of INDEPENDENT and INVARIANT constraints could also use better program analysis techniques. Lastly, we look forward to building actual programming tools on top of Yogo to gain deeper insights on how to improve on our approach.

Bibliography

[1] Niklas Broberg. language-java: Manipulating Java source: abstract syntax, lexer, parser, and pretty-printer. http://hackage.haskell.org/package/language-java.

[2] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. SNIFF: A Search Engine for Java Using Free-Form Queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, pages 385–400. York, UK, 2009.

[3] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, may 2005.

[4] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the Common Subexpression Problem. *Journal of the ACM*, 27(4):758–771, oct 1980.

[5] Charles L Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[6] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep Code Search. In *Proceedings of the 35th International Conference on Software Engineering*, volume 12, Gothenburg, Sweden, 2018. ACM.

[7] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation - PLDI '02*, volume 37, page 304, New York, New York, USA, 2002. ACM Press.

[8] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. FaCoY A Code-to-Code Search Engine. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, pages 946–957, New York, New York, USA, 2018. ACM Press.

[9] James Koppel, Varot Premtoon, and Armando Solar-Lezama. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. In *Proc. ACM Program. Lang*, volume 2, page 36, 2018.

[10] Fan Long and Martin Rinard. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178, Bergamo, Italy, 2015. ACM.

[11] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, San Diego, CA, 2005.

[12] Collin Mcmillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding Relevant Functions and Their Usages. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, Waikiki, Honolulu, HI, 2011.

[13] Na Meng, Miryung Kim, and Kathryn S Mckinley. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 35th International Conference on Software Engineering*, pages 502–511, San Francisco, CA, 2013.

[14] Charles Rich and Richard C Waters. The Programmer's Apprentice. *Computer*, 21j(11):10–25, 1988.

[15] Charles Rich and Linda M. Wills. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, 7(1):82–89, 1990.

[16] Wouter Swierstra. Data type a la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[17] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: a New Approach to Optimization. In *Proceedings of the Symposium on Principles of Programming Languages*, Savannah, GA, 2009.

[18] Danny van Bruggen. JavaParser: Process Java code programmatically. http://javaparser.org.

A Appendix: Search Patterns

Here are the search patterns for our evaluation.

```
1  (ns matches)
2
3  (defsearch sp1-bound-checking
4    (root <- (generic/binop :or
5                            (generic/binop :< x x-lo)
6                            (generic/binop :>= x x-hi))))
7
8  (defsearch sp2-squared-distance
9    (dx <- (generic/binop :- x1 x2))
10   (dy <- (generic/binop :- y1 y2))
11   (distance <- (generic/binop :+
12                               (generic/binop :** dx (generic/const 2))
13                               (generic/binop :** dy (generic/const 2)))))
14
15
16 (deftrigger sp3-map-put-if-not-present-trigger
17   (map-mem <- (concept/membership map-before k))
18   (mem-cond <- (generic/cond map-mem _ _))
19   (map-before <- (engine/q _ (at <- (generic/at m))))
20   (as <- (generic/assign mem-in (generic/sel m k) default))
21   (concept/is-memory mem-cond)
22   =>
23   (engine/q mem-cond at)
24   (engine/q mem-in at))
25
26 (defsearch sp3-map-put-if-not-present
27   (map-final <- (generic/cond map-mem map-before map-after))
28   (map-mem <- (concept/membership map-before k))
29   (as <- (generic/assign mem-in (generic/sel m k) default))
30   (map-before <- (engine/q mem-in (at <- (generic/at m))))
31   (map-after <- (engine/q (mem-out <- (generic/mem as)) at)))
32
33 (defsearch sp4-array-element-count
34   (answer <- (generic/final d bound counter))
35   (counter <- (generic/loop d (generic/const 0) next))
36   (next <- (generic/cond p inc counter))
37   (inc <- (generic/binop :+ counter (generic/const 1)))
38   (p <- (generic/binop :== e k))
39   (e <- (concept/iter-v d coll))
40   (bound <- (concept/iter-p d coll))
41   (invariant d k))
42
43 (defsearch sp5-time-diff
44   (start <- (custom/read-time _))
45   (end   <- (custom/read-time _))
46   (diff  <- (generic/binop :- end start)))
47
48 (defsearch sp6-loop-index
49   (i <- (concept/iter-v _ seq))
50   (seq <- (concept/seq (generic/const 0) (generic/const 1))))
51
52 (defsearch sp7-iter-map
53   (v <- (concept/get m k))
54   (k <- (concept/iter-v _ (concept/keys m)))
55   (m <- (engine/q _ (generic/at m-ref))))
```

```
56
57 (defsearch sp8-hashing
58   (digest <- (generic/val (concept/hash-digest _ hasher)))
59   (update <- (concept/hash-update _ hasher data))
60   (concept/is-hasher "MD5" hasher))
61
62 (defsearch sp9-file-writing
63   (file-write <- (concept/file-write _ file data))
64   (file <- (concept/file-handler filename)))
```