Object Graph Programming
Anonymous Author(s)

ABSTRACT
We introduce Object Graph Programming (OGO), which enables reading and modifying an object graph (i.e., the entire state of the object heap) via declarative queries. OGO models the objects and their relations in the heap as an object graph thereby treating the heap as a graph database: each node in the graph is an object (e.g., an instance of a class or an instance of a metadata class) and each edge is a relation between objects (e.g., a field of one object references another object). We leverage Cypher, the most popular query language for graph databases, as OGO’s primary query language. Unlike LINQ, which uses collections (e.g., List) as a source of data, OGO views the entire object graph as a single “collection”. OGO is ideal for querying collections (just like LINQ), introspecting the runtime system state (e.g., finding all instances of a given class or accessing fields via reflection), and writing assertions that have access to the entire program state. We prototyped OGO for Java in two ways: (a) by translating an object graph into a Neo4j database on which we run Cypher queries, and (b) by implementing our own in-memory graph query engine that directly queries the object heap. We used OGO to rewrite hundreds of statements in large open-source projects into OGO queries. We report our experience and performance of our prototypes.

KEYWORDS
Object graph, graph database, query, Cypher

1 INTRODUCTION
Declarative programming [46], focusing on the what rather than the how, has grown into the predominant way of programming in an increasing number of domains. For instance, Structured Query Language (SQL), a canonical example of the declarative paradigm, is the primary query language for most relational database management systems [6, 24, 59, 60]. At the same time, NoSQL databases have been gaining traction. In particular, the package graph databases [4, 44, 80, 81] is growing at a rapid pace, as they have been shown to be a great fit for tasks such as fraud detection, drug discovery [88], recommendation engines, and data visualization [20, 50]. Graph databases store data as property graphs [3, 39, 55], which emphasize relationships between data.

A property graph (graph for short) contains nodes \( N \) and edges \( E \) denoting relationships between nodes. Each node is assigned a label \( \mathcal{L} \) and contains an arbitrary set of properties \( \mathcal{P} \): mappings from nodes to values. Edges also have a label (sometimes called type in the literature) and an arbitrary set of properties. Querying, updating, and administering of such a graph is performed with a graph query language. Cypher [31, 40], initially developed as part of the Neo4j project [63], is currently the most popular graph query language [86]. Cypher is a declarative language, in many ways similar to SQL, which emphasizes simplicity and expressivity. As an example, to get the values of all nodes in a graph database we could run the following query: \texttt{match(n: Node) return n.val}. Although graph databases have been used for various tasks, the power of property graphs and graph query languages has yet to be used to enhance developers’ experience.

Our key insight is that an object graph [34], i.e., in-memory program state available at the execution time, can be seen as a property graph. We believe that being able to query object graphs during development, testing, and debugging will substantially extend the power of programming languages and tools.

We present Object-Graph Programming (OGO) that enables querying and updating an object graph via declarative queries. OGO treats a given object graph as a graph database: each node in the graph is an object (e.g., an instance of a class or an instance of a metadata class) and each edge is a relation between objects (e.g., a field of one object references another object). We leverage Cypher as OGO’s primary query language. This gives rise to endless opportunities to leverage OGO for programming, analyses, and tool development. We describe several use cases where OGO can be applied.

First, OGO provides a powerful and expressive way for writing assertions and program invariants [61]. Assertions written using OGO not only can access the local program state, but they can also check any aspect of the dynamic state of a program. For example, we could assert that there is never more than one instance of a specific (singleton) class.

Second, OGO can facilitate dynamic program analyses. For instance, a common task for tools that detect flaky tests [7, 77, 95] (due to test order dependency) is to check that the program state is the same at the beginning of each test. A common subtask is to find all object reachable from static variables. Using OGO, we can write a query to get all reachable objects starting from roots: \texttt{query("MATCH (n: [\{1\}]\-[*]->(m) RETURN m", roots)}.

Third, OGO can be used to introspect the system, such as finding all objects of a given class that satisfy a desired set of properties. Unlike reflection, which is frequently used to discover relations among objects and metadata via imperative traversal of object graphs, e.g., serialization code [9, 11, 69, 78, 79], OGO can help find these relations via queries over instances in memory and instances of metadata (assuming they are available as part of the object graph like in Java).

Finally, like LINQ, OGO could be used for querying collections of data and even implementing these collections. Unlike LINQ, OGO, at the moment, requires developers to cast their results to appropriate type, as we do not guarantee type safety, which is similar to working with the java.io package. At the same time, OGO can query any collection (e.g., an n dimensional array) without requiring a user to implement any interface.

We prototyped OGO for Java in two ways: (a) by translating an object graph into a Neo4j database on which we run Cypher queries (\texttt{OGONeo4j}), and (b) by implementing our own in-memory graph query engine that directly queries the object graph (\texttt{OGOInMem}). The former enables us to harvest the full power of a mature graph database, including the highly optimized query engine and visualization capabilities. However, the translation cost from object graph can be high even with a number of optimizations that we developed, and
We demonstrate the expressive power of declarative queries for
we developed a second prototype that works on the object graph
in memory. This approach requires close to zero extra memory, but
comes with significant engineering effort.

We evaluate the applicability and robustness of OGO by rewrit-
ing 230 assertions from 10 popular open-source projects available
on GitHub. We report execution time for both prototypes. Our results
highlight substantial performance benefits of in-memory
implementation.

The key contributions of this paper include:

- **Idea.** OGO introduces a new view of the runtime state of a pro-
gram and provides a novel way by which such a state can be
queried and modified. OGO offers developers a blend of imper-
ative and declarative programming abstractions to manipulate
the program state, increasing the expressivity of a programming
language which implements OGO’s paradigm. Although OGO
can be used to replace many statements (even a single field ac-
cess), it is best suited for tasks that include traversal of objects
and metadata, such as introspecting system state and writing
assertions and invariants.

- **Formalization.** We formalize OGO by giving a small-step oper-
tional semantics to Featherweight Java [43] in terms of property
graphs. This formalization captures the core of our translation
and can form the foundation for future projects that require
reasoning about correctness, such as query optimizations.

- **Implementation.** We implement two prototypes of OGO by
translating Java’s object graph to both an off-the-shelf graph
database and to an engineer-from-scratch in-memory database.
Although our focus was on features supported by OGO and not
on its performance, we describe several optimizations for both
translation and in-memory traversal.

- **Evaluation.** We evaluated the robustness of our prototypes and
compared their performance by rewriting a large number of asser-
tions that are already available in popular open-source projects.
Focusing on assertions simplified the selection of target state-
ments for the evaluation and enabled us to scale our experiment.
OGO is publicly available at [URL].

2 EXAMPLES

We demonstrate the expressive power of declarative queries for
analyzing program state by using two examples, such that each
example illustrates a different aspect of the framework: (1) creation
of instances, relations between instances and object graph pattern
matching; (2) implementing instance methods (containsKey) of
Java Collections framework class (java.util.HashMap).

2.1 Creating and Querying Instances and
Relations

The binary tree is a rooted ordered tree with each of its nodes
having at most 2 children. We demonstrate the versatility of OGO
by leveraging declarative queries to construct a binary tree and
query it for complex patterns. We also use this example to introduce
the syntax of the Cypher query language.

A Java implementation of the binary tree is given in Figure 5a.
An instance of BinaryTree contains a reference field root of type
Node (short for BinaryTreeNode), the root node of the tree and a
primitive int field size that tracks the total number of nodes
in the tree. An instance of Node contains references to its left
and right child nodes also of type Node and stores an integer value in
its primitive int field value.

Constructing a binary tree using OGO is given in Figure 1a.
We use the queryObject method of OGO to execute the passed in
Cypher query string. Based on Cypher grammar, the query contains
6 clauses. The first 2 are CREATE clauses (write to database/object
graph), the next 3 are MERGE clauses (write or read from database/object
graph) and finally, a RETURN clause (define expressions to re-
turn).

The first CREATE clause (lines 2-3) creates 5 Node instances. These
instances are assigned variable names a-e for referencing in followup
clauses. The expansion of the positional arguments @1, @2 is described
in Table 1 and are replaced with the fully qualified class name
(BinaryTreeNode and BinaryTree) respectively of the arguments following the query string. In Cypher syntax, these
are termed LABELS. We use LABELS to specify instance types for
graph nodes, consequently the CREATE clause creates instances of
the specified type. The LABEL is followed by property key-value
pairs ({{prop_name>:prop_value,...}}) which map to primi-
tive/String fields and their values. In CREATE, they assign the fields
of the created instance to the specified values. Thus, the value field
of instances a-e is assigned with values 1-5 respectively. The second
CREATE (line 4) creates an instance of BinaryTree (assigned f, this
can be subsumed into the first CREATE but is divided for clarity).
The MERGE clauses (line 5) are used to create relationships among
the a-e. Since the binary tree is a directed graph, we create directed
relationships. LABELS can also be specified for relationships. In our
graph model of the heap, the relationship between a referrer and a
referrer instance is labelled by the reference field corresponding to
the referrer instance in the referrer instance’s class. We use the refer-
ence fields left and right as labels for the relationships between
a-e. For e.g., (b $ $ $ $ left $ $ $ $ c) $ (c $ $ $ $ right $ $ $ $ d) translates
to assigning b and d as the left and right child of c. The RETURN
clause returns a reference to the BinaryTree instance f to prevent
the objects from being garbage collected.

We next describe querying the binary tree by discussing two pat-
terns. The first pattern investigates general reachability of objects
from a given object. Such queries are of importance to the problem
domains of Aliasing, Confinement and Ownership. For instance, if a
node in the binary tree was owned by another instance outside the
confinement of the binary tree instance then the aliased node could
potentially be mutated, leading to undesirable outcomes. Figure 1b
shows a query which returns all objects reachable from the root
Node instance. The query contains a MATCH (line 3) and RETURN
clauses. The MATCH clause matches all paths in the heap’s object
graph satisfying the given pattern (n$1) $ [+] $ m. The position
argument $1 expands to a unique identifier belonging to the
first argument following the query and is used to uniquely identify
an object in the heap. The matched root Node instance is assigned
variable n. The pattern neither specifies a label nor a direction for

The Java collections framework provides a rich collection of data structures supported natively by the Java platform. We show how OGO can be used to manipulate these objects by considering the example of implementing methods native to the Java collections' class java.util.HashMap. The HashMap stores data as key-value pairs where every stored value is mapped to a unique key.

A snippet of the HashMap class is given in Figure 2a. The reference field table contains all the entries in the map. The method containsKey of the HashMap class checks if a given key is present in the map. A purely imperative implementation of the containsKey method is given in Figure 2b. OGO implementation is shown in Figure 2c. The Cypher query used contains 2 MATCH clauses (line 3), the first clause matches all instances reachable from table that correspond to the reference field key (defined in the static nested class Node in HashMap) and refers to these set of instances as m, the second clause matches the instance passed in as an argument to the containsKey method. The WHERE clause is used to filter the set m based on the result of the equals method (overridden or inherited from java.lang.Object). If the equals method evaluates to true for atleast one instance in m and for m then the cardinality of set m is non-zero after this clause is completed and hence the RETURN clause returns true.

The examples show a glimpse of the potential of OGO: it provides access to any object in memory, at any point regardless of access specifiers, through a declarative API. Similar to this example, one could envision numerous other potential applications of OGO, a point which we return to in Section 4.

2.2 Implementing Java Library Methods
The Java collections framework provides a rich collection of data structures supported natively by the Java platform. We show how OGO can be used to manipulate these objects by considering the example of implementing methods native to the Java collections’ class java.util.HashMap. The HashMap stores data as key-value pairs where every stored value is mapped to a unique key.

A snippet of the HashMap class is given in Figure 2a. The reference field table contains all the entries in the map. The method containsKey of the HashMap class checks if a given key is present in the map. A purely imperative implementation of the containsKey method is given in Figure 2b. OGO implementation is shown in Figure 2c. The Cypher query used contains 2 MATCH clauses (line 3), the first clause matches all instances reachable from table that correspond to the reference field key (defined in the static nested class Node in HashMap) and refers to these set of instances as m, the second clause matches the instance passed in as an argument to the containsKey method. The WHERE clause is used to filter the set m based on the result of the equals method (overridden or inherited from java.lang.Object). If the equals method evaluates to true for atleast one instance in m and for m then the cardinality of set m is non-zero after this clause is completed and hence the RETURN clause returns true.

The examples show a glimpse of the potential of OGO: it provides access to any object in memory, at any point regardless of access specifiers, through a declarative API. Similar to this example, one could envision numerous other potential applications of OGO, a point which we return to in Section 4.

3 FRAMEWORK
We first present OGO’s API design (Section 3.1) followed by a high-level overview of the workings of OGO (Section 3.2). Next, we formally describe mapping of the object graph in the JVM heap memory to a property graph, as well as translation to Neo4j (Section 3.3). Lastly, we conclude the section with description of our implementation and optimizations details, sections 3.4 and 3.5, respectively.

3.1 API
We begin by describing queries—their type, arguments, and return values—followed by a discussion of our API design choices.

Queries. We show (the most important parts of) OGO’s API in Figure 3. The design choice of keeping the API minimal is intentional, similar to that available for working with relational databases such as java.sql. This allows developers acquainted with both Java and Cypher to be able to use OGO with ease.
framework using OGO. (a) The nested static class Node

(a) Snippet of java.util.HashMap class definition.

(b) Imperative implementation of HashMap's containsKey method.

(c) OGO implementation of HashMap's containsKey method.

Figure 2: Implementing methods from Java collections framework using OGO. (a) The nested static class Node stores a key-value pair and the reference field Table stores all the entries in the map (b) The imperative implementation of containsKey uses the getNode method (c) OGO can also be used to invoke instance methods as shown in the WHERE clause.

Figure 3: OGO' API available via the OGO class. Bounded queries (lines 1, 3 and 5) contain an additional root argument that constrains the query execution to a subgraph (limited to only objects reachable from root argument under transitive closure) of the JVM heap object graph.

Table 1: Positional arguments supported by OGO in the query formatting string.

Arguments. Both query methods in our API share the remaining arguments: formatting string (fmt) and values. The formatting string in its simplest form is just a Cypher query such as "MATCH (n:java.util.ArrayList) RETURN n". To enable constraining queries by embedding runtime values, we introduce several kinds of positional arguments; the values are provided from the third argument onward. We show kinds of positional arguments in Table 1. For each kind, we show an example and the expansion once the formatting is complete. We support embedding the unique id of an instance ($), fully qualified class name of an instance (@), or doing a union of query results when we run a query on each element of a (Iterable) collection (\[]). Return value. The result of each query is an instance of ResultSet. ResultSet [74] is available in Java as an interface and a common structure for storing the results of queries; similar structure is used in other programming languages. Via the resulting instance one can

```
1 public class HashMap<K,V> extends AbstractMap<K,V>
2 implements Map<K,V>, Cloneable, Serializable {
3     transient Node<K,V>[] table;
4 static class Node<K,V> implements Map.Entry<K,V> {
5 final int hash; final K key; V value; Node<K,V> next;
6 }
7 }

(a) Snippet of java.util.HashMap class definition.

1 public boolean containsKey(Object key) {
2 return getNode(hash(key), key) != null;
3 }
4 final Node<K,V> getNode(int hash, Object key) {
5 Node<K,V>[] tab; Node<K,V> first; e; int n; k;
6 if ((tab = table) != null && (n = tab.length) > 0 &&
7     (first = tab[(n - 1) & hash]) != null) {
8     if (first.hash == hash && // always check first node
9         (k = first.key) == key || (key != null && key.equals(k)))
10     return first;
11     if ((e = first.next) != null) {
12         if (first.instanceof TREEhive)
13         return ((TreeNode<K,V>)first).getTreeNode(hash, key);
14     do
15         if (e.hash == hash && (k = e.key) == key)
16         (key != null && key.equals(k))) return e;
17     while ((e = e.next) != null);
18     return null;}

(b) Imperative implementation of HashMap's containsKey method.

1 public boolean containsKey(Object key) {
2 return queryBool(
3 "MATCH ([\$1])-[\$2]->(\$3) \nWHERE n.'equals'(k) = true RETURN COUNT(n) = 0", this, key, 5)
4 }

(c) OGO implementation of HashMap's containsKey method.

1 public static ResultSet query(Object root, String fmt, Object... values);  
2 public static ResultSet query(String fmt, Object... values);  
3 public boolean queryBool(Object root, String fmt, Object... values);  
4 public static long queryLong(Object root, String fmt, Object... values);  
5 public static long queryLong(String fmt, Object... values);  
6 // similar for other primitive types

Figure 3: OGO' API available via the OGO class. Bounded queries (lines 1, 3 and 5) contain an additional root argument that constrains the query execution to a subgraph (limited to only objects reachable from root argument under transitive closure) of the JVM heap object graph.

The highlight of the API are two variadic query methods (lines 1 and 2 in Figure 3), which we call bounded query and unbounded query, respectively. We describe each one in turn.

In case of a bounded query, the first argument of the method takes an object (root) that constrains query execution only to objects reachable (under transitive closure of reference fields) from this root. Having the ability to specify only a subgraph of the entire object heap enables two things: (a) making localized queries, e.g., like we did in Figure 1c, and (b) improve performance of OGO, as

```
access columns (e.g., `getArray(int columnIndex)`), get the current row number (e.g., `getRow()`) etc. Anybody already familiar with working with relational databases from Java would be familiar with the structure.

For convenience, we introduced several query methods that return a specific type (lines 3-6). The only difference is that these methods cast/extract the result as a single value from the `ResultSet`; many assertions or queries that use `COUNT` would end up benefiting from this shorter version. As for the naming, we followed similar convention as the `Unsafe` class [75].

**Design decisions.** Similar to working with SQL strings and Java `sql`, OGO does not statically type check expressions. Thus, a dynamic `CastException` might be raised if a wrong value is passed to one of the query calls. Alternatively, rather than specifying the Cypher query as a string explicitly, the Cypher DSL [57] could be used instead. We leave this integration for future work.

Moreover, our API is not designed to be thread safe. Namely, the developer is responsible to ensure that appropriate locks are held when querying the (sub) object graph. This approach offers more flexibility without being different than implementing any code snippet imperatively.

### 3.2 Overview of OGO Steps

Figure 4 shows the high level overview of the working of OGO. The figure illustrates steps taken by both of our implemented prototypes, and highlights the differences between the two.

OGO flow starts once a query method is invoked, as described in the previous section. In addition to the query, OGO implicitly takes as input the entire state of the program. In the first step, OGO processes the formatting string and builds the actual query to be executed. This step is straightforward and includes simple string manipulations and object id discoveries (if the user has any collection in the formatting string, e.g., `[ ]`, OGO builds a batch query). In the second step, OGO uses a JVMTI agent (and the root object if given) to identify objects that are in the (sub)graph of interest. Note that a highly optimized system would not traverse the objects before analyzing the given query. Once the second step is done, the execution for OGO\textsubscript{Neo} and OGO\textsubscript{Mem} diverge.

OGO\textsubscript{Mem}, in step three, builds intermediate representations of the query (AST among others) and executes the query as per the Cypher semantics [30]. In the final step, OGO\textsubscript{Mem} collects the results into a `ResultSet`, which is the final result of the query execution.

Unlike OGO\textsubscript{Mem}, OGO\textsubscript{Neo}, in step three, translates (Section 3.3) objects of interest and their relations into a format accepted by Neo4j for batch data loading. In step four, OGO\textsubscript{Neo} passes the query to a Neo4j database running on a separate JVM and takes the result of the execution, which are so called node ids. In the final step, it processes these node ids to build the final result, which are the values known to the JVM. Primitive values remain as they are returned by Neo4j. On the other hand, non-primitive values are mapped to object ids, which are then used to fetch objects and build the final `ResultSet` returned to the user. OGO\textsubscript{Neo} and the Neo4j database are implemented as RMI client and server applications respectively. This prevents polluting the JVM heap with irrelevant Neo4j database objects.

### 3.3 Translation

In this section we will describe our translation from the JVM heap to a Neo4j graph database, by formalizing graph databases and presenting an operational semantics for Featherweight Java in terms of this formalization.

**Graph Databases, Formally.** A graph database is a directed multigraph: a pair $N \times R$ of nodes, the main entities of the graph, and relationships, the edges of the graph denoting directed connections between nodes. A node is a pair $L \times P$ of a label, drawn from some abstract domain that serves as the type of the node, and properties, a map from string keys to string values. A relationship corresponds to the edges of the multigraph: it has a start and end node, a label$^2$, and a key-value property map.

**The Java Heap as a Graph Database, by Example.** Following the binary tree example presented in Section 2, consider a simple `BinaryTree` instance in Figure 5b, which defines a `Node` 1 with a value field 4, and a `BinaryTree` b whose root is a `Node` with 1 as its left subtree, no right subtree and a value field of 5. A pictorial representation of the property graph at the marked `POINT` is shown in Figure 5c.

- **Object instances** Every object instance that has been allocated corresponds to a node in the property graph. It’s label is the name of the object’s class, and it’s property set contains the values of any primitive fields or strings. In our example, the node corresponding to 1 has the label `Node` and a singleton property set that maps `value` to 4, while the node corresponding to b has the label `BinaryTree` and a singleton property set that maps size to 2.

- **Fields** Each reference field of an object corresponds to a relationship whose label is the name of the field, its origin is the node of the graph that corresponds to the object it belongs to, its destination is the object corresponds to the field’s value, and its property set is empty. The `root` field of b is then an edge in the graph going from b to the nameless node corresponding to the second allocation, whose `left` field in turn points to 1.

- **Local variables** Each local variable we introduce (such as n and l), gives rise to both a node in the graph whose label is `Local` and whose property set is empty, and to a relationship whose

\[\text{Corresponds to the fully qualified class name of the instance represented by the node}\]
class BinaryTree {
  private Node root;
  private int size;
  static class Node {
    private Node left;
    private Node right;
    int value;
    int mtype;
    Node() {
      value = 0;
      mtype = "m";
    }
    Node(int val) {
      value = val;
      mtype = "m";
    }
  }
  void test() {
    Node l = new Node();
    Node r = new Node();
    Node root = new Node(l, r);
  }
}

(a) BinaryTree class definitions.

(b) An example test class.

(c) Graph database at "POINT".

Figure 5: An example class, its test class and the corresponding object graph shown as a property graph. (c) instances of Node, BinaryTree and Java.lang.Class are shown colored black, blue and grey. The local variable references are shown as dashed edges. Reference fields (left, right) are mapped as node relations whereas primitive (value, size) and String fields are mapped as node properties.

Figure 6: Syntax of Featherweight Java.

label is the name of the variable from that node to the object corresponding to the local variable’s value.

- **Class information** Each object is also related to a static definition of its class via an instanceof relationship. This allows us to capture, for example, static fields belonging to the class at its property set.

**Featherweight Java.** Featherweight Java, as introduced by Igarashi et al. [43], constitutes the object-oriented core of Java. Its syntax is shown in Figure 6; it consists of: class declarations such as class C extends D (C, F, K), which introduce a class C with D as its superclass, F as its fields, K as its constructor, and H as its methods; constructor declarations C(F)(super(X); this.x=F; x); which initialize the fields of an instance of C; and method declarations C.m(F)(e) which define a method m with arguments X of types C whose body e returns a type T. This setup allowed Igarashi et al. to resolve field (fields), method type (mtype), and method body (mbody) lookups from a fixed class table in a straightforward manner, which we will assume in the rest of the presentation.

Unlike Featherweight Java, where the bodies e where a single return expression, we consider method bodies in assignment normal form (ANF), as our aim is to formalize the small-step impact of expressions as a graph rather than their big-step reductions and their interactions with (sub)typing. Method bodies are therefore either return expressions that return a variable, or sequences of commands c that either allocate a new object, assign to a field, or invoke a method call.

What we’re ultimately interested in is modelling the shape of the object graph after each command. To that end, we introduce two (mutually recursive) judgements: a small-step judgement that relates a command c and an input graph G to the resulting graph G’ and a big-step judgement that relates an entire expression e and an input graph G to the resulting graph G’:

\[
\frac{c \Rightarrow G \rightarrow G'}{e \Rightarrow G \rightarrow G''}
\]

Figure 7: Big-step reduction for Featherweight Java expressions.

\[
\begin{aligned}
(N, R) &\Rightarrow n_x = R(x), 3 \quad n_y = R(y), 3 \\
r &= (f, n_x, n_y, 0) \\
x.f &= (N, r \cup \{ R(f, n_x, 0) \}) \\
x.m &= x \Rightarrow G \\
mbody(m) &= \{ y, e \} \Rightarrow G \\
\end{aligned}
\]

Figure 8: Small-step reduction for Featherweight Java commands.

\[
\begin{aligned}
\text{fields}(C) &= \{ f \} \\
C &= \{ y \} \Rightarrow \text{this}, \{ f \} \Rightarrow \text{this} \\
\text{mkFields}(n_c, C(\{ x \})) &= F \Rightarrow G \\
\end{aligned}
\]

Figure 9: Calculation of fields relationships.
graph to include this new relationship while removing the previous relationship corresponding to the \( f \) field of \( x \).

To invoke a method \( m \) of a variable \( x \) with some arguments \( \bar{x} \), we first lookup the body of the method (using \( mbody \) as defined in Igarashi et al.), which is an expression \( e \) parameterized by arguments \( \bar{y} \). We then substitute \( \bar{x} \) for \( \bar{y} \) and \( x \) for \( \text{this} \) in \( e \) and use the big-step judgment for expressions to construct the resulting object graph \( G' \).

Finally, to create a new object \( C \) by invoking its constructor with some arguments \( \bar{x} \) and assigning this new object to a fresh local variable \( x \), we need to extend the graph with two new objects, one corresponding to \( x \) (whose label is \( Local \) and whose property map is empty), and one corresponding to the newly created object (whose label is \( C \) and whose property map is empty). Then we need to create a collection of field relationships (Figure 9) to account for the initialization of the new object. We do that via a helper meta-function \( mkFields \), identifying two cases:

- If \( C \) does not extend another class, then its constructor does not involve a call to \( super \) and is just a sequence of field initializations \( \text{this.}\bar{\bar{x}} = \bar{\bar{y}} \). In this case, we find the object \( n_x \), corresponding to each argument \( x_i \) passed into the constructor (it is the third component of the relationship whose label is \( x_j \)), and construct a new relationship with an empty property set \( (f, nc, N(x_j), 3, \emptyset) \).
- If \( C \) does extend some other class \( D \), then the first \( k \) arguments (denoted as \( \bar{\bar{x}} \)) to the constructor will be used to initialize \( D \), while the rest will be used to initialize \( C \)'s fields. To construct the full set of new relationships, we recursively call \( mkFields \) for \( D \) with the first \( k \) arguments, and then augment the resulting set with the initializations for \( C \)'s fields, calculated as in the base case.

### 3.4 Implementation

We now describe the implementation of OGO by considering the execution of the Cypher query in Figure 1c. The execution can be divided into the following 4 steps.

1. **Query Pre-processing**. Semantically, the query aims to find patterns that contain instances referring to other instances that are 2 hops away, referenced through fields named \( \text{left} \) or \( \text{right} \) and those which contain a primitive \( \text{int} \) field named value with value \( 1 \). The first step involves processing the query format string to expand the positional arguments using the expansion described in Table 1. The positional arguments \( \$1 \) and \( \$i \) are expanded into a unique identifier and the fully qualified class name for \( \text{binary} \).root respectively. Any function that maps objects bijectively to a set comprised of either, one of the Java primitive data types or \( \text{java.lang.String} \) elements can be used to generate the unique identifier in the expansion of \( \$ \). This constraint is based on the design of our property graph model where Java primitive and \( \text{String} \) fields are embedded as node properties. We use the hashcode computed using the \text{hashCode} method from the \( \text{java.lang.System} \) package as the unique identifier in our implementation. This function is bijective except for some pathological scenarios [72].

2. **GraphTriggerException**. The next step is the identification of a subgraph of objects in the JVM heap memory, of relevance to the query. Since a significant number of objects in the heap may be irrelevant to the query, it is more efficient to index once and store a subgraph of the complete object graph than to re-traverse it for every additional clause in the query. We store the subgraph in semantically corresponding lightweight \( C++ \) classes (for e.g., a Java class definition is stored in \( C++ \).ClassInfo etc).

We use a native agent developed using the JVMTI [70] framework to identify the subgraph. Native agents can be triggered using JVMTI events based on certain actions from within a Java application. The general approach is to perform an action in the Java domain of OGO after pre-processing the query that triggers an event inside JVMTI. The callback provided by JVMTI to service the event can then be used as the entry point to the native agent. We use the exception event, generated when an exception (any instance of \( \text{java.lang.Throwable} \)) is thrown by a Java method to trigger the native agent. We created a dummy exception class \( \text{GraphTriggerException} \). This exception is thrown from a dummy method (\( \text{setupGraph} \)) after the query pre-processing step. In the JVMTI callback for handling exception events, we monitor if the signature of the method from which this exception is thrown matches that of \( \text{setupGraph} \). If a match is found then the native agent proceeds with identification of the subgraph.

3. **Native agent.** OGO uses JVMTI to identify relations between objects and their field properties (name, value, type, etc.) for identifying the subgraph. The native agent is implemented in the \( C++ \) programming language. The following are the major steps involved in identifying the subgraph.

- **Tag zero.** JVMTI callbacks use tags which are primitive long types to refer to objects in the heap. These tags can be modified inside certain JVMTI callbacks and is the preferred way of identifying and tracking relations across objects. However, these tags can also be modified by JVMTI internal processes. Therefore, as an initialization step, we iterate through all objects in the heap and assign their tags to 0. Optionally, our optimization—Force Garbage Collection—can be enabled to force a GC event prior to tagging to reduce the total number of objects in the heap consequently, reducing the overhead incurred by all the following JVMTI callbacks.

- **Loaded classes.** The next step involves identifying the types of the objects potentially relevant to the query. This helps limit the number of objects to be considered for inclusion into the subgraph. OGO provides an optimization Whitelist which can be used to flag certain user specified classes whose instances are guaranteed to be included in the subgraph. In addition, OGO also provides a blacklist for specifying classes whose instances are to be definitively excluded from the subgraph. Furthermore, in addition to each object being referred to by tags, JVMTI framework also uses tags to identify classes. By initializing all objects to 0 in the Tag zero step, we also initialize the tags of all classes to 0 (instances of \( \text{java.lang.Class} \)). When a class is blacklisted, it remains untagged (tag=0) and hence its instances will not be reported in any of the following JVMTI callbacks since all JVMTI callbacks support filters to filter out untagged objects and classes.

- **Iterate heap assign unique tag.** In this step we assign a unique tag to every instance of every tagged class. We also allocate memory to certain bookkeeping \( C++ \) classes required for storing information about the subgraph. These unique tags help identify relations between objects in the next step.
Follow references. This JVMTI callback traverses the object graph in the JVM heap. It first reports the referrer and referee instances followed by the primitive fields, String fields and array primitive fields of the referrer instance before doing the like for the referee instance. We once again limit the objects reported by this callback by applying filters ensuring that the reported objects as well as their classes are tagged. By default, if no root object is specified, the traversal is started from a set of system classes, JNI globals and other objects used as roots for garbage collection. The optimization Fix Root Objects can be used to start the traversal from the specified object. This reduces the overhead of traversing paths irrelevant to the specified query.

Write graph to CSV. This step applies to NEO exclusively. In this step, we serialize the subgraph into CSV files such that they can be batch imported into a Neo4j database.

(4) Executing the query. For the NEO implementation, the exported CSV files are loaded into a Neo4j database and the given query is executed using Neo4j’s Cypher engine. For queries involving the return of a primitive/String fields (stored as properties of Neo4j nodes), the result of Neo4j’s Cypher engine is the result of the specified query. For queries that involve the return of an object, we use the semantically equivalent Neo4j node returned by Neo4j’s Cypher engine to obtain the unique identifier. Followed by this, we use the JNI framework to retrieve the corresponding instance from the JVM heap.

For the OGO implementation, we used the ANTLR4 [5, 76] parser generator to generate the parser and visitor for the Cypher query language. We use the visitor pattern to deduce the semantics and execute the query. There is no overhead of writing the subgraph to CSV or setting up and creating a Neo4j graph mirroring the state of the JVM’s heap. The query is executed directly on the subgraph and like former, JNI is used to report the result back to Java.

3.5 Optimizations

To improve the performance of OGO, we introduced 3 optimizations, Whitelist (WL), Force Garbage Collection (FGC) and Fix Root Objects (FRO).

Whitelist (WL). Limits the size of the subgraph by specifying the type of instances to be definitively included (instances reachable from the specified instance types under transitive closure are also included).

Force Garbage Collection (FGC). Force a garbage collection event to reduce the number of objects in the JVM’s heap before performing the steps to identify the subgraph. This decreases the overhead incurred during the JVMTI callbacks.

Fix Root Objects (FRO). Limits JVMTI FollowReferences callbacks to reporting instances that are transitively reachable from the provided root object. The root object is passed as an argument to the query API call (bounded query).

4 EVALUATION

We evaluated OGO in two parts: (1) by rewriting existing assertion statements available in tests in open-source projects and, (2) by implementing methods from Java data structure libraries. The first part demonstrates the robustness of our system and ease of its integration with large open-source projects while the second part describes its expressive power over a purely imperative approach. Most of the selected projects are supported by large software organizations, such as Apache or Google. This section describes the experiment setup, chosen subjects and our findings.

4.1 Experiment Setup

The OGO queries were benchmarked on a 64-bit Ubuntu 18.04.1 desktop with a 11th Gen Intel(R) Core(TM) i7-8700 @ 3.20GHz and 64GB RAM. We use Java 11.0.16 and Neo4j 4.4.0 for all experiments.

4.2 Results

Re-written Assertions. The selected projects, their lines of code (LOC) and OGO query execution times under different introduced optimizations are given in Table 2. All executions are averaged over 50 runs except for the Naive case which is averaged over 3 runs due to long execution time. The reported times are all end-to-end and limited to the scope of OGO API methods (includes the object graph construction time, the query time itself and in case of NEO, serialization and clearing of Neo4j database). We do not consider the total test time, as long running tests would then mask the actual cost of queries. All times are in milliseconds.

Columns 5 through 9 report the query times for NEO prototype whereas those for OGO are given in columns 10 through 12. The final column shows the speedup of OGO over NEO. For NEO and OGO we show time with different optimization levels, which were described in Section 3.5. To compute the values in the last column, we use NEO, +WL+FRO+FGC and +WL+FGC. The last two rows in the table show the average (Avg.) and total time (Σ) across all the assertions from all projects.

OGO:Naive is not usable since this attempts to include every object in the heap into the graph; there is even an instance when the entire run crashed as the VM ran out of memory (Geometry). Each individual optimization provides substantial reduction in the execution time over Naive. +WL+FGC provides the biggest reduction as it dramatically reduces the number of objects to be translated. Finally, combining all the optimizations together (NEO,+WL+FRO+FGC) gives the best performance in most cases. Speedup of NEO over NEO is up to 94%. Looking at the third column, we find that NEO:Naive is substantially faster than even the most optimal NEO. Furthermore, using +WL+FGC improve times of NEO by half.

In summary, we find that our optimizations are effective. Furthermore, NEO is, at the moment, better suited for writing program statements due to its low cost. However, we still see NEO being very much usable in a debugging environment, where moderate overhead with a large number of features provided by Neo4j can be effectively used.

Implementing library Methods. We re-wrote methods of classes from Guava, the Java Collections Framework (JCF) and the Java Universal Network/Graph Framework (JUNG) using OGO. Table 3 shows the average and total lines of code (LOC) and number of characters (NOC) for implementing the selected methods using OGO and a purely imperative approach. We see that on average,
Table 2: End-To-End execution time in milliseconds of assertions re-written using OGO with different optimizations for external database and in-memory query executions.

<table>
<thead>
<tr>
<th>Project</th>
<th>SHA</th>
<th>LOC</th>
<th>#Assert</th>
<th>OGO&lt;sub&gt;Neo&lt;/sub&gt;</th>
<th>OGO&lt;sub&gt;Mem&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Naive - +FRO +FGC</td>
<td>Naive - +FRO +FGC</td>
</tr>
<tr>
<td>Assertj-db</td>
<td>8aeaf0f</td>
<td>7855</td>
<td>9</td>
<td>832652 7063 7092 4799 4861</td>
<td>851 860 248 95</td>
</tr>
<tr>
<td>CLI</td>
<td>0d06c4b</td>
<td>9256</td>
<td>27</td>
<td>99798 1987 1982 1900 1880</td>
<td>224 226 160 91</td>
</tr>
<tr>
<td>Csv</td>
<td>1c551d9</td>
<td>9910</td>
<td>12</td>
<td>448403 2867 2713 2291 2171</td>
<td>319 318 178 92</td>
</tr>
<tr>
<td>Lang</td>
<td>dded8fd</td>
<td>86960</td>
<td>23</td>
<td>255390 1718 1724 1510 1437</td>
<td>287 286 148 90</td>
</tr>
<tr>
<td>Geometry</td>
<td>6c6d046</td>
<td>79996</td>
<td>95</td>
<td>1407 1397 1321 1267 1267</td>
<td>258 257 145 89</td>
</tr>
<tr>
<td>Guava</td>
<td>0ca124d</td>
<td>367861</td>
<td>5</td>
<td>175658 2804 2784 2458 2408</td>
<td>217 218 130 95</td>
</tr>
<tr>
<td>JFreechart</td>
<td>7ceb1be</td>
<td>134399</td>
<td>10</td>
<td>383694 1811 1824 1625 1548</td>
<td>256 234 134 91</td>
</tr>
<tr>
<td>JSON-Java</td>
<td>31110b5</td>
<td>14829</td>
<td>10</td>
<td>99798 1456 1423 1483 1400</td>
<td>155 155 126 91</td>
</tr>
<tr>
<td>TCases</td>
<td>2fccf74</td>
<td>482581</td>
<td>26</td>
<td>323006 39736 39713 2120 2068</td>
<td>768 796 199 90</td>
</tr>
<tr>
<td>Zip4j</td>
<td>fc3a258</td>
<td>15525</td>
<td>18</td>
<td>79658 3528 3548 3322 3157</td>
<td>244 242 174 94</td>
</tr>
</tbody>
</table>

Avg. 269806 6419 6420 2283 2219 358 359 164
Σ 2698057 64197 64200 22829 22197 3579 3592 1642

Table 3: Comparison of LOC and NOC (Number of Characters) between OGO and Imperative Implementation of selected instance methods.

<table>
<thead>
<tr>
<th>Class</th>
<th>OGO LOC</th>
<th>OGO NOC</th>
<th>Imp. LOC</th>
<th>Imp. NOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guava</td>
<td>ArrayTable</td>
<td>9</td>
<td>315</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>HashBiMap</td>
<td>6</td>
<td>226</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>LinkedListMultiMap</td>
<td>6</td>
<td>202</td>
<td>48</td>
</tr>
<tr>
<td>JUNG</td>
<td>SparseGraph</td>
<td>7</td>
<td>303</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>UndirectedSparseGraph</td>
<td>6</td>
<td>281</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>DirectedSparseGraph</td>
<td>6</td>
<td>281</td>
<td>36</td>
</tr>
<tr>
<td>JCF</td>
<td>ArrayList</td>
<td>6</td>
<td>212</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>HashMap</td>
<td>6</td>
<td>214</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>LinkedList</td>
<td>6</td>
<td>226</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>ArrayDeque</td>
<td>3</td>
<td>99</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Vector</td>
<td>6</td>
<td>213</td>
<td>20</td>
</tr>
<tr>
<td>Avg.</td>
<td>6</td>
<td>234</td>
<td>31</td>
<td>842</td>
</tr>
<tr>
<td>Σ</td>
<td>67</td>
<td>2572</td>
<td>337</td>
<td>9264</td>
</tr>
</tbody>
</table>

OGO requires 5 and 4 times as less LOC and NOC than its counterpart. Furthermore, we benchmarked the implemented methods for our two prototypes with different optimizations and the results are shown in Table 4. The reported execution times are averaged over 200 runs. We see that OGO<sub>Mem</sub> once again outperforms OGO<sub>Neo</sub>, furthermore, we also observe certain instances where test execution times of OGO<sub>Neo</sub> is significantly larger. Based on the profiled data, this stems from the unpredictability in execution times of node and relationship creations.

5 DISCUSSION AND FUTURE WORK

Although OGO is already production ready, there are endless opportunities for improvements and applications. We document several directions in this section.

We have focused our work on implementing various features rather than OGO’s performance, so far. Future work should implement various optimizations that most graph databases already include, such as graph compression techniques [27, 28], indexing [17, 97], memory-efficient custom data structures [38, 92, 93].

At the moment, OGO uses Cypher as the primary query language. There are several other popular alternatives that can be supported in the future, including Gremlin [84], SPARQL [21], PGQL [91], GSQL [96], and GraphQL [29]. We chose Cypher as it is the most dominant language at this point. Additionally, we were very much familiar with Neo4j.

In this paper, we focused solely on having OGO being used to program an application. Another direction is to bring OGO to support development tools. Specifically, OGO could be integrated with debugging tools, e.g., jdb [71]. This integration would enable developers to navigate the entire state of a program in an easy way (by writing unbounded queries) and discover interesting values and relations. Additionally, having data in a graph database provides data visualization capabilities with off-the-shelf tools; existing graph visualization libraries are way more advanced than any existing visual debugger [2, 32, 33, 65].

In-memory databases (such as H2 and HSQLDB) are widely used in industry. We are excited about the possibility to develop an in-memory graph database on top of OGO. An in-memory graph database would be a subgraph (of the object-graph) and nodes would be instances of dynamically generated classes.

OGO allows developers to break one of the core software engineering principles: encapsulation. While the power of OGO enables various ways to treat the system, responsible use has a great potential. Furthermore, there are many ways in which encapsulation in Java (and other languages) is already being broken (e.g., Unsafe [41, 56]) when it comes to designing program analysis tools. Having another, more effective way to implement analyses tools, is a plus.

OGO idea is applicable beyond Java and integration with other languages, especially those that are dynamically typed, is a planned future work.
6 RELATED WORK

We cover the most closely related work in this section by comparing our work with the following groups: (1) program analysis using query and domain specific languages (DSL) and, (2) minimizing impedance mismatch between imperative programming languages and database systems.

Program analysis. Closely related to OGO are Fox [82] and Datalog [15]. Fox uses a DSL query language to enable analyzing object graph in the JVM for aliasing, confinement and ownership. Datalog and its applications to (static) program analysis have been explored in numerous studies [13, 42, 48]. Most prominently, the Doop framework [10] and followup work [87] express various forms of pointer analysis in Datalog by exploiting its expressive power. We focus on dynamic program analysis and our key insight that the entire program heap can be seen as a single graph database which can be queried via popular graph query languages. The expressive power of Cypher as a query language enables concise descriptions in applications as shown in Section 2.

Impedance mismatch. The friction between imperative programming languages and their interfacing with database systems stemming primarily from their independent development has been coined the term Impedance Mismatch [54]. Efforts to identify and categorize [22, 23, 45] this mismatch in an attempt to reduce it have been achieved through object-oriented databases, object-relational mappers, data access APIs, embedded query languages [22] and language integrated queries.

Call level interface (CLI). are API’s such as JDBC [35] and ADO.Net [1, 8, 14, 16] that abstract away the generation of the query language through API methods. However, it is difficult [51] to ensure the efficiency of the generated queries.

Embedded query languages. API’s such as SQLJ [25, 26], XJ [36, 37] allow embedding the query language to query external databases. Although OGO shares similarities in that it allows writing queries in Cypher inside Java, however, OGO allows the in-memory object graph to be queried.

Relational object mappers. Relational object mappers [12, 19, 52, 68, 90], like Hibernate, enable conversion of data between type systems. They convert objects to (relational) database by automatically grouping properties and enable loading and updating these values. OGO is about querying object graphs not about persistence. The closest connection with object mappers is our translation from an object graph into a graph database.

Language integrated queries. (LINQ) [58], developed by Microsoft, is a technology that adds native data querying capabilities to .NET languages. As a data source, LINQ can use in-memory data, i.e., any collection that implements IEnumerable (e.g., List, SortedSet). Although powerful LINQ provides no support to query arbitrary objects and their relations. Language integrated queries have seen renewed interest [18, 47, 49, 53, 62, 66, 67, 83, 85, 89] since the release of the LINQ framework. Similar frameworks for Java include SBQL4J [94] and Quaere [64].

7 CONCLUSION

We introduced object graph programming (OGO), a novel paradigm that combines imperative (object-oriented) programming and declarative queries. OGO treats the program state (i.e., object graph) as a graph database that can be queried and modified using graph query language(s); OGO currently uses Cypher as the primary query language. Each object in an object graph is a node, each primitive, String and primitive array field is a property, and each reference field forms a relation between two nodes. OGO is ideal for querying collections (similar to LINQ), introspecting the runtime system state (e.g., finding all instances of a given class or accessing fields via reflection), and writing assertions that have access to the entire program state. We prototyped OGO for Java in two ways: (a) by translating the JVM heap object graph into a Neo4j database on which we run Cypher queries, and (b) by implementing our own in-memory graph query engine that directly queries the object graph. We used OGO to rewrite hundreds of statements in large open-source projects into OGO queries. Our evaluation shows the wide applicability of our approach and good first results with in-memory implementation. OGO enables an entirely different view of objects and data, which will move programming experience to the next level.
REFERENCES


[27] Wenfei Fan, Jianzhong Li, Xín Wang, and Yinhui Wu. 2012. Query Preserving Graph Compression. In SIGMOD.


