.QL: Object-Oriented Queries Made Easy

Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev
Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Julian Tibble

Semmle Limited

Abstract. These notes are an introduction to .QL, an object-oriented query language for any type of structured data. We illustrate the use of .QL in assessing software quality, namely to find bugs, to compute metrics and to enforce coding conventions. The class mechanism of .QL is discussed in depth, and we demonstrate how it can be used to build libraries of reusable queries.

1 Introduction

Software quality can be assessed and improved by computing metrics, finding common bugs, checking style rules and enforcing coding conventions that are specific to an API. Many tools for these tasks are however awkward to apply in practice: they often detract from the main task in hand. Above all, it is tough to customise metrics and rules to one's own codebase, and yet that is where the greatest benefit lies.

These lectures present a new approach, where all these tasks related to software quality are phrased as *queries* over a relational representation of the code base. Furthermore, the language for expressing these queries is object-oriented, encouraging re-use of queries, and making it easy to tailor them to a specific framework or project. While code queries have been considered before (both in industry and academia), the object-oriented query language (named .QL) is unique, and the key to creating an agile tool for assessing software quality.

As an advance preview of .QL, let us briefly consider a rule that is specific to the *Polyglot* compiler framework [46]. Every AST node class that has children must implement a method named "visitChildren". In .QL, that requirement is checked by the query:

```
class ASTNode extends RefType {
  ASTNode() { this.getASupertype+().
                hasQualifiedName("polyglot.ast","Node") }
  Field getAChild() {
     result = this.getAField() and
     result.getType() instanceof ASTNode
  }
}
from ASTNode n
where not(n.declaresMethod("visitChildren"))
select n, n.getAChild()
```

Of course this may appear rather complex to the reader for now, but the example still serves to illustrate a couple of important points. First, this is a very useful query: in our own research compiler for *AspectJ* called *abc* [4], we found no less than 18 violations of the rule. Second, the query is concise and in a syntax resembling mainstream languages like SQL and Java. Third, as we shall see later, the class definition for *ASTNode* is reusable in other queries.

.QL has been implemented as part of an Eclipse plugin named *SemmleCode*. SemmleCode can be used to query any Java project. It provides an industrial-strength editor for .QL, with syntax highlighting, autocompletion and so on (as shown in Figure 1). Furthermore, the .QL implementation itself is quite efficient. Java projects are stored in relational form in a standard database system. That database system can be a cheap-and-cheerful pure Java implementation such as H2 (which is distributed with SemmleCode), or a dedicated system such as PostgreSQL or SQL Server. With a proper database system in place, SemmleCode can easily query projects that consist of millions of lines of code. That scalability is another unique characteristic that sets SemmleCode apart from other code querying systems.

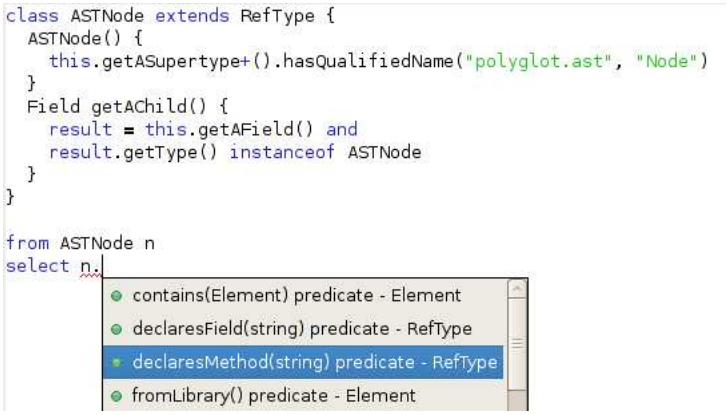

Fig. 1. The SemmleCode editor

.QL is in fact a general query language, and could be thought of as a replacement for SQL — the application to software quality in these notes is just an example of its power. Like SQL, it has a simple and intuitive syntax that is easy to learn for novices. In SQL, however, that simple syntax does not carry over to complex constructs like aggregates, while in .QL it does. Furthermore, recursion is natural in .QL, and efficiently implemented. Compared to the direct use of recursion in SQL Server and DB2, it can be orders of magnitude faster. Finally, its object-oriented features offer unrivalled flexibility for the creation of libraries of reusable queries.

The structure of these lectures is as follows:

- First we shall consider simple queries, using the existing library of classes that is distributed with SemmleCode. An important concept here is the notion of *non-deterministic methods*, which account for much of the conciseness of .QL queries. We shall also examine features such as casts and instance tests, which are also indispensable for writing effective queries in .QL.
- In the second part of these lectures, we take a close look at the object-oriented features of .QL. First we illustrate the ideas with a number of motivating examples, and then zoom in on a number of subtle issues in the design of .QL's class mechanism. As indicated above, our notion of classes somehow must be tied to a traditional database, and we outline how that is done by appropriate annotation of a database schema.

1.1 Exercises

Exercises for the reader have been sprinkled throughout these notes. Most of the exercises involve writing a new query in .QL, and it is strongly recommended that readers follow along with SemmleCode running on a computer. For full instructions on how to install SemmleCode, visit the Semmle website [53].

The Java project used in the exercises is JFreeChart 1.0.6 [33]. We have chosen JFreeChart because it is a well-written piece of Java code, and its developers already make extensive use of *checkstyle* [12], the most popular Eclipse plugin for checking coding rules. Nevertheless, as we shall see, there are still several problems and possible improvements that are easily unearthed with SemmleCode.

There is a special web page accompanying these notes that takes you through the steps required to load JFreeChart in Eclipse, and populate the database with facts about the project [54].

Each exercise has an indicator of its difficulty at the end: one heart is easy (less than five minutes), two hearts is medium (requiring at most ten minutes), and three hearts is a tough exercise (requiring up to fifteen minutes). Full answers can be found in an appendix to these notes.

2 Program Queries

2.1 A Simple Query

Program queries allow programmers to navigate their source code to identify program fragments with arbitrary semantic properties. As a simple example of a program query in .QL, let us attempt to find classes which violate Java's `compareTo` / `equals` contract. The Java documentation for the `compareTo` method states:

> *The natural ordering for a class* `C` *is said to be consistent with* `equals` *if and only if* `(e1.compareTo((Object)e2) == 0)` *has the same boolean value as* `e1.equals((Object)e2)` *for every* `e1` *and* `e2` *of class* `C` *... It is strongly recommended (though not required) that natural orderings be consistent with equals.*

The following .QL query identifies those classes that only implement the `compareTo` method without implementing the `equals` method. This is likely to indicate a bug, though it is not necessarily erroneous:

```
from Class c
where c.declaresMethod("compareTo")
      and
      not(c.declaresMethod("equals"))
select c, c.getPackage()
```

This query consists of three parts. First, the **from** statement declares the variables of interest (in this case just the class *c* that we are looking for) together with their types. The second part of the query is the **where** clause imposing some conditions on the results. In this query, the condition is that the class *c* should declare a method called `compareTo` but not a method called `equals`. The final part of the query is the **select** statement, to choose the data to return for each search result, namely the package in which the offending class `c` occurs, together with `c` itself. The order of the **select** items is chosen so that results are presented grouped by the package in which they occur in the source.

The type *Class* is an example of a .QL class. This type defines those programs elements which are Java classes, and defines operations on them. For instance, *declaresMethod* is a test on elements of type *Class*, to select only those Java classes declaring a particular method. We will be describing .QL types and classes in more detail in Section 3, but examples will appear throughout.

Exercise 1. Run the above query on JFreeChart. You can do that in a number of ways, but here the nicest way to look at the results is as a table, so use the run button marked with a table (shown below) at the top right-hand side of the Quick Query window. You will get two results, and you can navigate to the relevant locations in the source by double-clicking. Are both of them real bugs? ♡

2.2 Methods

Predicates such as *declaresMethod* are useful, but can only filter results. Another common task is to compute some properties of an element. This is achieved by more general .QL methods, which may return results. Let us illustrate this with an example query. Unlike the previous query, which attempted to detect violations of Java's style rules, and therefore could easily be hard-coded into a development environment, the next query is domain-specific.

Suppose that we are working on a compiler, and would like to identify the children of nodes in the AST, for instance to ensure that appropriate methods for visiting children are implemented. To code this as a query, we declare three variables: *child* for the field, *childType* for type of that field, and *parentType* for the parent class:

```
from Field child, ASTNode childType, ASTNode parentType
where child.getType() = childType
      and
      child.getDeclaringType() = parentType
select child
```

The *ASTNode* class is an example of a user-defined class, picking out those types that are AST nodes, and described further in Section 3. The methods *getType* and *getDeclaringType* are defined in the class *Field*, and are used to find the declared type of a field and the type in which the field declaration appears, respectively. The *ASTNode* types appearing in the **from** clause serve to restrict the range of values for the variables they qualify, so that values of the wrong type are simply ignored.

This query is concise, but not terribly satisfactory. In the **from** clause, we define variables *childType* and *parentType* to denote the types of the field and its containing class respectively. However we are not really interested in these types, and indeed they do not appear in the **select** clause. To avoid polluting queries with such irrelevant types, local declarations can be introduced through the **exists** statement:

```
from Field child
where exists(ASTNode childType | child.getType() = childType)
      and
      exists(ASTNode parentType | child.getDeclaringType() = parentType)
select child
```

An advantage of the resulting query is that the scopes of the variables representing the types of the field and the container are made explicit. There is a further improvement to be made, however. These fields are only used to restrict the types we are looking for, as we are only interested in AST nodes. We do not need to know the exact types of the child and parent, and so it would be better not to introduce variables to hold these types. .QL offers an **instanceof** construct to achieve this, and we can finally rewrite the query as:

```
from Field child
where child.getType() instanceof ASTNode
      and
      child.getDeclaringType() instanceof ASTNode
select child
```

Exercise 2. Write a query to find all methods named `main` in packages whose names end with the string `demo`. You may find it handy to use the predicate *string.matches*("%demo") (as is common in query languages, % is a wildcard matching any string). ♡

2.3 Sets of Results

Methods in .QL are a convenient way of finding properties of elements, as well as a powerful abstraction mechanism in conjunction with classes. The methods we

have seen so far define attributes of elements, such as the declaring type of a field. This is only represent a special case, however, since the data model behind .QL is relational and thus allows methods to define arbitrary *relationships* between elements.

As an example, we will consider a query to find calls to the `System.exit` method. This method terminates the Java virtual machine, without offering the opportunity to clean up any state. This should therefore usually be avoided, and identifying calls to this method allows potentially fragile code to be found. The query is:

```
from Method m, Method sysexit, Class system
where system.hasQualifiedName("java.lang", "System")
  and sysexit.getDeclaringType() = system
  and sysexit.hasName("exit")
  and m.getACall() = sysexit
select m
```

The first line of the **where** clause identifies the `java.lang.System` class, while the second and third lines find the `exit` method in this class. The last line is of more interest. The expression *m.getACall*() finds *all* methods that are directly called by *m*. This method returns a result for each such call, and any logical test on the result is performed for each possible result. In this case, each method called by *m* is compared to the `exit` method. If one of the calls matches (*i.e.*, *m* calls `exit`), then the equality succeeds and *m* is returned. Otherwise, this value of *m* is not returned. The query thus singles out just those methods that (directly) call `exit`.

Methods returning several results can be chained arbitrarily. In the following example, we search for calls between packages, that is all the calls from any method in one package to any method in another package. This may be used to construct a call graph representing dependencies between packages, and identify potential problems such as cycles of dependencies between packages.

```
from Package caller, Package callee
where caller.getARefType().getACallable().calls(
            callee .getARefType().getACallable())
      and caller.fromSource()
      and callee.fromSource()
      and caller != callee
select caller , callee
```

The expression *caller.getARefType*() finds any type within the package *caller*, so that *caller.getARefType*().*getACallable*() finds any method or constructor (referred to as a *callable*) within some type in the *caller* package. The use of methods returning several values greatly simplifies this expression, and avoids the need to name unimportant elements such as the type or callable, focusing only on the pairs of packages that we are searching for. As this expression (and its analogue for *callee*) return all callables in the package, the query succeeds exactly for those pairs of packages in which any callable of *caller* calls some callable in

callee. The predicate *fromSource*(), which holds of program elements defined in a source file (as opposed to a Java class file), serves to exclude results from library code. Finally, the last line of the **where** statement removes trivial dependencies of packages on themselves.

The use of sets of results is sometimes called *nondeterminism*, and a method that possibly has multiple results (like *getARefType* above) is said to be *nondeterministic*. Nondeterminism can sometimes be a bit subtle when used inside a negation. For instance, consider the .QL method *getACallable* that returns any callable (constructor or method) of a class. We could find classes that define a method named "equals" with the query

from Class c
where c.getACallable().hasName("equals")
select c

In words, for each class c, we try each callable, and test whether it is named "equals"; if one of these tests succeeds, c is returned as a result. Now consider the dual query, where we wish to identify classes that do *not* have a method named "equals". We can do that just by negating the above condition, as in

from Class c
where not (c.getACallable().hasName("equals"))
select c

The negated condition succeeds only when *none* of the tests on the callables of c succeeds.

Exercise 3. The above queries show how to find types that define a method named "equals", and how to find types that do not have such a method. Write a query picking out types that define at least *one* method which is not called "equals". ♡

Exercise 4. Continuing Exercise 1 about `compareTo`. You will have found that one class represents a real bug, whereas the other does not. Refine our earlier query to avoid such false positives. ♡♡

Exercise 5. Write a query to find all types in JFreeChart that have a field of type *JFreeChart*. Many of these are test cases; can they be excluded somehow? ♡

2.4 .QL Type Hierarchies and Casts

In the previous section we defined a query to find all dependencies between packages, by looking for method calls from one package to another. However, such calls are only one possible way in which a package may depend on another package. For instance, a package might use a type from another package (say with a field of this type), without calling any methods of this type. This is intuitively a dependency which we would like to record, and indeed there are many more ways

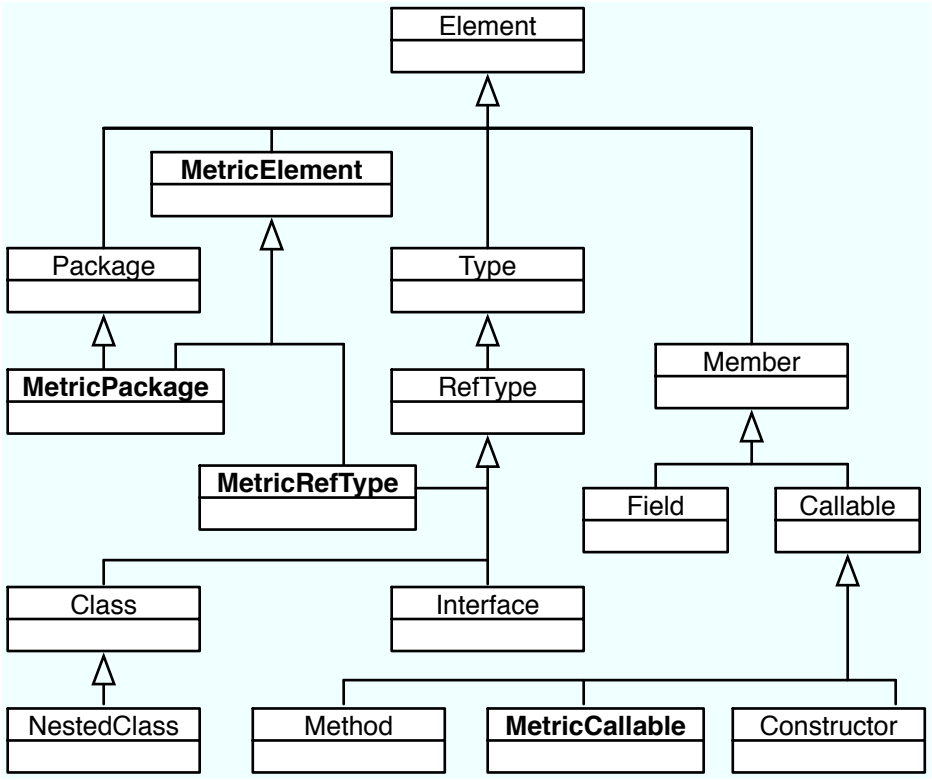

Fig. 2. Standard Library: Inheritance Hierarchy (excerpt)

in which a package may depend on another. This is encapsulated in a method *getADependency*, defined as part of the *metrics library* for Java programs.

The metrics library, which we shall be using throughout these notes, extends the basic .QL class definitions for Java programs with additional methods to compute information about dependencies in source code, and to evaluate various quantitative metrics to analyse the code. In order to separate these definitions from the basic classes, some .QL classes representing program elements, *e.g. RefType*, are extended by counterparts in the metrics library, in this case *MetricRefType*, which contains all methods for computing dependencies and metrics on reference types, in addition to the standard methods defined in *RefType*. The class *MetricRefType* does not, however, restrict the set of elements that it contains — any *RefType* is also a *MetricRefType*, and the metric class merely provides an extended view of the same object. Figure 2 describes the inheritance hierarchy for (part of) the standard .QL library for Java programs, with the metrics classes highlighted. The metrics library makes crucial use of multiple inheritance for .QL classes (described later in Section 3) — a *MetricPackage* is both a *Package* and a *MetricElement*.

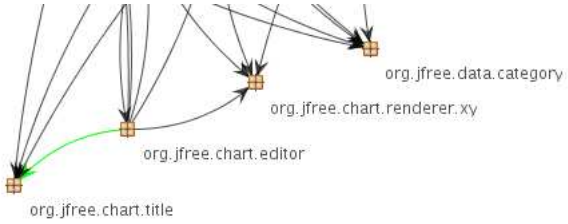

Fig. 3. A fragment of the graph showing inter-package dependencies in JFreeChart

Using the metrics library it is straightforward to find precise dependencies between packages, as the class *MetricElement* defines the methods *getADependency* to find dependent elements, and *getADependencySrc* to find dependencies from source. The query is shown below:

```
from MetricPackage p
select p, p.getADependencySrc()
```

This query finds all packages p, and for each p finds those packages defined in source that depend on p. The results of this query form a dependency graph, part of which is shown in Figure 3. Suppose, now, that we do not want to inspect just the other packages that p depends on, but instead also the types that inhabit such packages. At first you might want to write a query that looks like this:

```
from MetricPackage p
select p, p.getADependencySrc().getARefType() // incorrect!
```

However, that is in fact not type correct, because the result of the method *getADependencySrc* is a *MetricElement*, and *MetricElement* does not have the method *getARefType*. The .QL compiler therefore rejects the above query. We must amend it by casting the result of *getADependencySrc* to a *Package*:

```
from MetricPackage p
select p, ((Package) p.getADependencySrc()).getARefType()
```

The cast here will always succeed because when given a package as the receiver, *getADependencySrc* always returns another package. Similarly, starting from a *MetricRefType*, it will always return a *RefType*.

Casts in .QL also behave like **instanceof** tests, limiting results to those of a certain type. For instance, this query will filter out all the types that are not an instance of *Class*:

```
from MetricPackage p
select p, (Class) ((Package)p.getADependencySrc()).getARefType()
```

It follows that casts in .QL never lead to runtime exceptions as they do in languages like Java: they are merely a test that a logical property (in this case a reference type being a class) is satisfied.

2.5 Chaining

The queries that we have seen so far find relatively local properties of program elements, such as the declaring type of a field, or the relationship of one method directly calling another. However, many properties of interest are highly nonlocal, justifying the introduction of *chaining*, also known as transitive closure.

As an example, we shall write a query to find all types that represent AST nodes in a compiler (in this case the Polyglot compiler framework [46]), as suggested previously by our use of the *ASTNode* class. In Polyglot, AST nodes must implement the `Node` interface, and so we are interested in all subtypes of this interface. The standard .QL library for Java provides a convenient *hasSubtype* method to find subtypes of a type, but this only finds *immediate* subtypes, in this case all classes that implement `Node` directly. As we are also interested in classes that are indirect descendents of `Node`, we must use chaining, written in .QL using the + postfix operator:

```
from RefType astNode, RefType rootNode
where rootNode.hasQualifiedName("polyglot.ast","Node")
      and (rootNode.hasSubtype+(astNode)
             or
           astNode = rootNode)
select astNode
```

The method *hasSubtype+* picks out all direct and indirect subtypes of a type (in this case the `Node` interface). AST nodes are defined as subtypes of `Node`, together with `Node` itself. As this pattern is extremely common, simpler notation is provided for possibly empty chains, as the query is equivalent to:

```
from RefType astNode, RefType rootNode
where rootNode.hasQualifiedName("polyglot.ast","Node")
      and rootNode.hasSubtype*(astNode)
select astNode
```

The $*$ operator (known as *reflexive transitive closure* in mathematics) defines possibly empty chains from given relationships, such as the subtype relationship. The symbols +, $*$ may be familiar from repetition in regular expressions where `a*` denotes any number of occurrences of `a`, while `a+` denotes at least one occurrence of `a`, justifying the intentional similarity in notation.

Exercise 6. There exists a method named *getASuperType* that returns *some* supertype of its receiver, and sometimes this is a convenient alternative to using *hasSubtype*. Uses of methods such as *getASuperType* that return an argument can be chained too. Using $x.getASuperType*()$, write a query for finding all subtypes of `org.jfree.chart.plot.Plot`. Try to use no more than one variable. ♡

Exercise 7. When a query returns two program elements plus a string you can view the results as an edge-labelled graph by clicking on the graph button (shown below). To try out that feature, use chaining to write a query to depict the hierarchy above the type `TaskSeriesCollection` in package `org.jfree.data.gantt`.

You may wish to exclude `Object` from the results, as it clutters the picture. Right-clicking on the graph view will give you a number of options for displaying it. ♡♡

2.6 Aggregates

We have so far seen .QL used to find elements in a program with certain properties. The language also offers powerful features to aggregate information over a range of values, to compute numerical metrics over query results. These features are substantially more expressive than their SQL counterparts, and allow a wide range of metrics to be computed straightforwardly. As a first example, the following query computes the number of types in each package in a program:

from Package p
select p, **count**(RefType c | c.getPackage() = p)

The **count** expression in this query finds those elements c of type *RefType* (all reference types) satisfying the condition $c.getPackage() = p$. The value of the expression is just the number of results, that is the number of reference types in p.

The above query is a simple example of the aggregate constructs in .QL. Aggregates in .QL adopt the *Eindhoven Quantifier Notation* [21, 34], an elegant notation introduced by Edsger W. Dijkstra and others for the purpose of reasoning about programs. The general syntax for aggregates is

aggregateFunction (localVariables | condition | expression)

The aggregateFunction is any function for aggregating sets of values. The functions provided in .QL are **count**, **sum**, **max**, **min** and **avg** (average). The localVariables define the range of the aggregate expression, namely the variables over the values of which the aggregation is computed. The condition restricts the values of interest. In our previous example, the condition was used to restrict counting to those types in the appropriate package. Finally, the expression defines the value to be aggregated. In our above example, the expression was omitted. This is always possible when counting, as the value of each result in the aggregation is irrelevant. The expression becomes very useful in other aggregates such as summation, however. As an example, the following query computes the average number of methods per type in each package:

from Package p
where p.fromSource()
select p, **avg**(RefType c | c.getPackage() = p | c.getNumberOfMethods())

This aggregate finds all reference types in the appropriate package, finds the number of methods for each such type (which itself is easily defined as an aggregate), and averages these numbers of methods.

Exercise 8. Display the results of the above query as pie chart, where each slice of the pie represents a package and the size of the slice the average number of methods in that package. To do so, use the *run* button marked with a chart (shown on the next page), and select '*as a pie chart*' from the drop-down menu. ♡

Aggregates may be nested, as the expression whose value is being aggregated is often itself the result of an aggregate. The following example computes the average number of methods per class over an entire project:

```
select avg(Class c
         | c.fromSource()
         | count(Method m | m.getDeclaringType()=c))
```

This query contains two aggregates. The outermost aggregate computes an average over all classes c that are defined in source files. For each such class, the value of the innermost aggregate is computed, giving the number of methods in the class, and the resulting values are averaged. This example does not include **from** or **where** clauses, as only one result is returned, so it is not necessary to define output variables.

Metrics An important use of aggregates in program queries is to compute *metrics* over the code. Such metrics may be used to identify problematic areas of the program, such as overly large classes or packages, or classes that do not encapsulate a single abstraction. It is not our aim here to describe the vast library of software metrics that have been proposed (see, for instance, [8, 14, 18, 36, 41, 56]), but we shall use such metrics as examples of the use of aggregates in .QL.

Many of these metrics are provided as a library, and use the object-oriented features of .QL to achieve encapsulation and reusability, as illustrated in Figure 2. However, as we discuss these features in Section 3, we shall simply express metrics as standalone queries for now.

Instability Instability is a measure of how hard it is to change a package without changing the behaviour of other packages. This is represented as a number between 0 (highly stable) and 1 (highly unstable). Instability is defined as follows:

$$Instability = \frac{EfferentCoupling}{AfferentCoupling + EfferentCoupling}$$

where the *efferent coupling* of a package is the number of types outside the package that the package depends on, while the *afferent coupling* is the number of outside types that depend on this package. Typically a package that has recently been added and is still experimental will have high instability, because it depends on many more established packages, while few other packages depend

on the new package. Conversely, a package with many responsibilities that is at the core of an existing project will have low instability, and indeed such packages are hard to modify.

It is easy to define queries to compute efferent and afferent coupling. As these are similar, we present afferent coupling only:

```
from Package p
select p, count(RefType t
              | t.getPackage() != p and
                exists(RefType u |
                         u.getPackage() = p and
                         depends(t, u)))
```

where the *depends* predicate, part of the metrics library, is fairly straightforward but lengthy, and so is omitted.

We now aim to define the instability metric. This is a clear case for the expressiveness of .QL classes. Without encapsulation mechanisms, there is no easy means of reusing definitions such as afferent coupling. In section 3 we shall see how definitions such as afferent coupling can be defined as methods. These definitions are in fact part of the metrics library and we can write the instability metric in a straightforward way:

```
from MetricPackage p, float efferent, float afferent
where efferent = p.getEfferentCoupling()
      and
      afferent = p.getAfferentCoupling()
select p, efferent / (efferent + afferent)
```

Without methods, the aggregate expressions for efferent and afferent coupling would have to be inlined, leading to a far less readable query. The above definition of instability is in fact itself available as a method named *getInstability* on *MetricPackage*, so a shorter version is

```
from MetricPackage p select p, p.getInstability()
```

Exercise 9. Not convinced that metrics are any good? Run the above query and display the results as a bar chart—the chart icon mentioned earlier for creating pie charts (shown below) is also used to create bar charts by selecting the appropriate option from the drop-down menu. It will be convenient to display the bars in descending order. To achieve that sorting, add "**as** s **order by** s **desc**" at the end of the query. Now carefully inspect the packages with high instability. Sorting the other way round (using **asc** instead of **desc**) allows you to inspect the stable packages. ♡

Abstractness Abstractness measures the proportion of abstract types in a package, as a number between 0 (not at all abstract) and 1 (entirely abstract). Packages should be abstract in proportion to their incoming dependencies, and concrete in proportion to their outgoing dependencies. That way, making changes is likely to be easy. There is therefore a relationship between abstractness and instability: the more abstract a package is, the lower its instability value should be. A highly abstract, highly stable package is well designed for its purpose and represents a good use of abstraction; conversely, concrete packages may be unstable as nothing depends on concrete packages. Abstract and unstable packages, however, are likely to be useless and represent design flaws.

Abstractness is easy to define: it is just the ratio of abstract classes in a package to all classes in this package. For a package p this may be written as:

```
from Package p, float abstract, float all
where all = count(Class c | c.getPackage() = p)
      and abstract = count(Class c
                           | c.getPackage() = p and
                             c.hasModifier("abstract"))
      and abstract > 0
      and p.fromSource()
select p, abstract / all
```

This query computes the number of types in the variable *all* and the number of abstract types in *abstract*, and for nonempty packages returns the ratio of the two. Again we gave this definition merely for expository reasons, as a method named *abstractness* has already been defined on *MetricPackage*; therefore an alternative query (which also sorts its results in descending order) is:

```
from MetricPackage p where p.fromSource() and p.abstractness() > 0
select p, p.abstractness() as a order by a desc
```

As in the previous exercise, this is a suitable query for viewing as a bar chart. The result is shown in Figure 4.

Semantics of Aggregation Aggregates in .QL are extremely general constructs, and while their use is largely intuitive as our above examples have shown, it is worth describing the exact meaning of aggregate queries in a little more detail. This section may be omitted on first reading, but forms a useful reference for the semantics of aggregate expressions.

An aggregate query of the form

$$\mathsf{aggregate}\ (\ T_1\ x_1,\ T_2\ x_2, \ldots,\ T_n\ x_n \mid \mathsf{condition} \mid \mathsf{expression}\ )$$

ranges over all tuples $(x_1, \ldots, x_n)$ of appropriately-typed values satisfying condition. The condition is a conditional formula in which the variables x_i may appear, and which allows some of the tuples to be excluded. Variables defined outside the aggregate may appear in the condition — the value of such variables is computed outside the aggregate, and the aggregate is evaluated for each possible assignment of values to external variables.

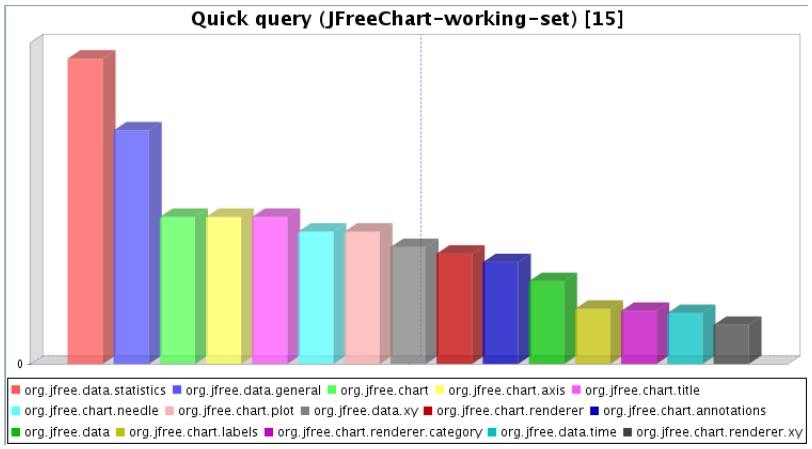

Fig. 4. A bar chart of the *abstractness* of packages in JFreeChart

For each tuple $(x_1, \ldots, x_n)$ making the condition true, the expression is evaluated. The values of the expression are then collected and aggregated (counted, added, ...). It is important to note that these values are not treated as a set, but allow duplicates. As an example, consider the following expression:

sum (**int** i | (i=0 **or** i=1) | 2)

Evaluation of this proceeds as described above: the set of integers i satisfying the condition $i = 0$ **or** $i = 1$ is collected, giving just the set $\{0, 1\}$. The expression has a constant value of 2, so the values to be summed are *two* copies of 2 — one for the assignment $i = 0$ and the other for the assignment $i = 1$. The result of the aggregate is therefore $4 = 2 + 2$.

As another example, consider the following:

sum (**int** i, **int** j
| (i=3 **or** i=4) **and** (j=3 **or** j=4)
| i∗i + j∗j)

This sum ranges over four tuples: $(3, 3)$, $(3, 4)$, $(4, 3)$ and $(4, 4)$. The result of the sum is thus $18 + 25 + 25 + 32 = 100$.

This notation is convenient, but it would be cumbersome to have to include all parts of the aggregate, including the term and condition, when these are not needed. A number of shorthands are therefore provided:

1. Counting: the expression can always be omitted in a **count** aggregate, as it is irrelevant
2. Numerical values. For other aggregates, such as **sum**, the expression can be omitted in exactly one case, namely if the aggregate defines one local variable of numerical type. For instance, the aggregate

   sum (**int** i | i=0 **or** i=1)

is simply equivalent to

sum (**int** i | i=0 **or** i=1 | i)

and thus adds the values of i matching the condition. This obviously cannot be extended to non-numerical variables — it does not make sense to add classes together!

3. Omitting condition: if the condition is not required, it may be omitted altogether. For instance, adding the number of types in each package may be written:

   sum (Package p | | p.getNumberOfTypes())

   This is particularly simple for counting, as both condition and expression can be omitted. Simply counting the number of packages can be achieved with

   count (Package p)

Exercise 10. The following questions are intended to help reinforce some of the points made above; you could run experiments with SemmleCode to check them, but really they're just for thinking.

1. What is **sum**(**int** $i \mid i = 0$ **or** $i = 0 \mid 2$)?
2. Under what conditions on p and q is this a true equation?

$$\textbf{sum}(\textbf{int } i \mid p(i) \textbf{ or } q(i)) = \textbf{sum}(\textbf{int } i \mid p(i)) + \textbf{sum}(\textbf{int } i \mid q(i))$$

♡

3 Object-Oriented Queries

So far we have merely written one-off queries, without any form of abstraction to reuse them. To enable reuse, .QL provides classes, including virtual methods and overriding, making it easy to adapt existing queries to new requirements. We present these features in a top-down fashion. First, we discuss some motivating examples, to give the reader a general feel for the way classes are used in practice. Next, we take a step back and examine the semantics of classes and virtual dispatch in some detail through small artificial examples. Finally, we demonstrate how a class hierarchy in .QL can be built on top of a set of simple primitive relations, of the kind found in traditional databases.

3.1 Motivating Examples

Classes A class in .QL is a logical property: when a value satisfies that property, it is a member of the corresponding type. To illustrate, let us define a class for 'Visible Instance Fields' in Java, namely fields that are not static and not private. Clearly it is a special kind of normal Java field, so our new class is a subclass of *Field*:

```
class VisibleInstanceField extends Field {
  VisibleInstanceField() {
    not(this.hasModifier("private")) and
    not(this.hasModifier("static"))
  }

  predicate readExternally() {
    exists(FieldRead fr |
             fr.getField()=this and
             fr.getSite().getDeclaringType() != this.getDeclaringType())
  }
}
```

This class definition states that a *VisibleInstanceField* is a special kind of *Field*. The constructor actually makes the distinguishing property of the new class precise: this field does not have modifier `private` or `static`. The conjunction of the constructor with the defining property of the supertype is called the *characteristic predicate* of a class. It is somewhat misleading to speak of a 'constructor' in this context, as nothing is being constructed: it is just a predicate, and naming it the *character* might have been more accurate. However, we adopt the terminology 'constructor' because it is familiar to Java programmers.

The above class also defines a predicate, which is a property of some *VisibleInstanceField*s. It checks whether this field is read externally. In order to make that check, it introduces a local variable named *fr* of type *FieldRead*: first we check that *fr* is indeed an access to **this** field, and then we check that the read does not occur in the host type of **this**. In general, a predicate is a relation between its parameters and the special variable **this**.

Newly defined classes can be used directly in select statements. For instance, we might want to find visible instance fields that are *not* read externally. Arguably such fields should have been declared private instead. A query to find such offending fields is:

```
from VisibleInstanceField vif
where vif.fromSource() and
      not(vif.readExternally())
select vif.getDeclaringType().getPackage(),
       vif.getDeclaringType(),
       vif
```

It should now be apparent that all those predicates we have used in previous queries were, in fact, defined in the same way in classes as we defined *readExternally*. We shall shortly see how methods (which can return a result as well as check a property) are defined as class members. It follows that while at first it may appear that .QL is specific to the domain of querying source code, in fact it is a general query language — all the domain-specific notions have been encoded in the query library.

Classless Predicates Sometimes there is no obvious class to put a new predicate, and in fact .QL allows you to define such predicates outside a class. To illustrate, here is a classless predicate for checking that one Java field masks another in a superclass:

```
predicate masks(Field masker, VisibleInstanceField maskee) {
    maskee.getName()=masker.getName() and
    masker.getDeclaringType().hasSupertype+(maskee.getDeclaringType())
}
```

In words, the two fields share the same name, but the *masker* is defined in a subtype of the *maskee*, while the *maskee* is visible. Such field masking is often considered bad practice, and indeed it can lead to confusing programming errors. Indeed, most modern development environments, including Eclipse, provide an option for checking for the existence of masked fields. In .QL, any such coding conventions are easily phrased as queries. In particular, here is a query to find all the visible instance fields that are masked:

```
from Field f, VisibleInstanceField vif
where masks(f,vif)
select f, vif
```

Exercise 11. Queries can be useful for identifying refactoring opportunities. For example, suppose we are interested in finding pairs of classes that could benefit by extracting a common interface or by creating a new common superclass.

1. As a first step, we will need to identify *root definitions:* methods that are not overriding some other method in the superclass. Define a new .QL class named *RootDefMethod* for such methods. It only needs to have a constructor, and no methods or predicates.
2. Complete the body of the following classless predicate:

   ```
   predicate similar(RefType t, RefType s, Method m, Method n) { ... }
   ```

   It should check that m is a method of t, n is a method of s, and m and n have the same signature.
3. Now we are ready to write the real query: find all pairs (t, s) that are in the same package and have more than one root definition in common. All of these are potential candidates for refactoring. If you have written the query correctly, you will find two types in JFreeChart that have 99 root definitions in common!
4. Write a query to list those 99 root definitions in a table.

♡♡

Methods Often the introduction of a classless predicate is merely a stepping stone towards introducing a new class. Wrapping predicates in a class has several advantages. First, your queries become shorter because you can use method dispatch and so there is no need to name intermediate results. Second, when typing

queries you get much better content assist, so you do not need to remember details of all existing predicates.

To illustrate, we introduce a class *MaskedField* as a subclass of the class *VisibleInstanceField* defined earlier:

```
class MaskedField extends VisibleInstanceField {
  MaskedField() { masks(_,this) }
  Field getMasker() { masks(result,this) }
  string getIconPath() { result = "icons/semmle−logo.png" }
}
```

The constructor for this .QL class consists of the property *masks*(_, **this**) stating that **this** is being masked by some other field. Here, as in many other logic languages, we use the underscore to represent a fresh variable whose value is not relevant. Next the class introduces two methods. The *getMasker*() method returns the masker of **this**. In general, the body of a method is a relation between two special variables named **result** and **this**; the relation may also involve any method parameters. Our new class also defines a method *getIconPath*, which is used to determine the icon that is displayed next to a program element in the results views provided by an implementation. In fact this overrides a method of the same signature in *Field*, and so from now on masked fields will be displayed differently from other fields. Somewhat frivolously, we have decided to give them the Semmle icon.

A query that uses the above class might read:

```
from MaskedField mf select mf,mf.getMasker()
```

and the results will be displayed with the new icon we just introduced.

Note that predicates in a class are really just a special kind of method that returns no result; indeed one could think of them as analogous to `void` methods in Java. Also note, once again, that methods may be nondeterministic. Indeed, in the above example, it is possible that one field in a Java class C is masked by several fields in different subclasses of C. Nondeterminism is a natural consequence of the fact that the method body is a relation between **this**, **result** and the method parameters. There is *no* requirement that **result** is uniquely determined.

Framework-specific Classes It is often worthwhile to define new classes that are specific to a particular framework, and we already encountered an example of that earlier, namely *ASTNode* (in Section 2.2). Now we have all the machinery at hand to present the definition of *ASTNode*. We assume the context of the *Polyglot* compiler framework [46], which is intended for experimentation with novel extensions of the Java language. In Polyglot, every kind of *ASTNode* is an implementation of the interface `polyglot.ast.Node`. This can be directly expressed in .QL:

```
class ASTNode extends RefType {
  ASTNode() { this.getASupertype+().
```

```
                hasQualifiedName("polyglot.ast","Node") }

  Field getAChild() {
    result = this.getAField() and
    result.getType() instanceof ASTNode
  }
}
```

Note the use of nondeterminism in the constructor: effectively it says that there exists *some* supertype that implements the *Node* interface. The method *getAChild* returns a field of an AST class, that is itself of an AST type. Of course it can happen that no such field exists (if the class represents a terminal in the grammar), or there may be multiple such fields.

In Polyglot, there is a design rule which says that every AST class that has a child must implement its own *visitChildren* method. We now aim to write a query for violations to that rule: we seek AST classes that do *not* declare a method named *visitChildren*, yet a child exists:

```
from ASTNode n
where not(n.declaresMethod("visitChildren"))
select n, n.getAChild()
```

At first it may appear that the condition that a child exists has been omitted, but in fact we do attempt to get a child in the **select** part of the query. If no such child exists then *n.getAChild*() will fail, and so the query will return no results for this value of *n* — exactly what we intended.

This type of coding convention is extremely common in non-trivial frameworks. Normally the conventions are mentioned in the documentation, where they may be ignored or forgotten. Indeed, in our own use of Polyglot in the *abc* compiler, there are no less than 18 violations of the rule. Interestingly, there are no violations in any of the code written by the Polyglot designers themselves — they do as they say. By making the rule explicit as a query, it can be shipped with the library code, thus ensuring that all clients comply with it as well.

As another typical example of a coding convention, consider the use of a factory. Again in Polyglot, all AST nodes must be constructed via such a factory; the only exceptions allowed are super calls in constructors of other AST nodes. Violation of this rule leads to compilers that are difficult to extend with new features.

Definition of a class that captures the essence of an AST node factory in Polyglot can be expressed in .QL as follows:

```
class ASTFactory extends RefType {
  ASTFactory() { this.getASupertype+().
                   hasQualifiedName("polyglot.ast","NodeFactory")
  }
  ConstructorCall getAViolation() {
      result.getType() instanceof ASTNode and
      not(result.getCaller().getDeclaringType()
```

```
                    instanceof ASTFactory) and
      not(result instanceof SuperConstructorCall)
  }
}
```

The constructor is not interesting; it is just a variation of our earlier example in *ASTNode*. The definition of *getAViolation* is however worth spelling out in detail. We are looking for an AST constructor call which does *not* occur inside an AST factory, and which is also not a super call from an AST constructor. Again, we successfully used this query to find numerous problems in our own code for the *abc* compiler.

Exercise 12. We now explore the use of factories in JFreeChart.

1. Write a query to find types in JFreeChart whose name contains the string "Factory."
2. Write a class to model the Java type `JFreeChart` and its subtypes.
3. Count the number of constructor calls to such types.
4. Modify the above query to find violations in the use of a factory to construct instances of `JFreeChart`.
5. There are 53 such violations; it is easiest to view them as a table. The interesting ones are those that are not in tests or demos. Inspect these in detail — they reveal a weakness in the above example, namely that we may also wish to make an exception for *this* constructor calls. Modify the code to include that exception. Are all the remaining examples tests or demos?

♡♡♡

Default Constructors New .QL classes do not have to define a constructor; when it is not defined, the default constructor is the same as that of the superclass. A .QL class with no constructor of its own does not define a new logical property, but this can often be handy when we want to define a new method that did not exist in the superclass, but which really belongs there.

For instance, suppose that we wish to define a method named *depth* that returns the length of a path from Object to a given type in the inheritance hierarchy. That method is not defined in the standard library definition of *RefType*, but it really is a property of any reference type. In .QL, we can add it as such via the definition

```
class RT extends RefType {
    int depth() {
        (this.hasQualifiedName("java.lang", "Object") and result=0)
        or
        (result = ((RT)this.getASupertype()).depth() + 1)
    }
    int maxDepth() {
        result = max(this.depth())
```

```
    }
}
```

That is, the depth of *Object* itself is 0. Otherwise, we pick a supertype, compute its depth and add 1 to it. In the recursive step, we cast a *RefType* to a *RT*, just so we can call depth on it. That cast will always succeed, because the characteristic predicates of *RT* and *RefType* are identical. Because Java allows multiple inheritance for interfaces, there may be multiple paths from a type to *Object*, and therefore we also define a method for finding the maximum depth of a type. This example was just for illustration and the same result can be obtained via *MetricRefType.getInheritanceDepth*().

3.2 Generic Queries

To conclude our introduction to object-oriented queries, we consider the definition of a metric that exists both on packages and on reference types: the Lakos level [36]. This metric, which was first introduced by John Lakos, is intended to give insight into the *layers* of an application: at the highest level, are the most abstract parts of the program, and at the bottom, utility elements. The Lakos metric is part of the metrics library, and we shall describe how many of the methods from this library that have already been used in earlier sections may be defined.

To appreciate the level metric, consider the well-known drawing framework *JHotDraw*. When arranging packages according to level, the highest point is a package containing sample applications, and a low point is a package of utility classes for recording user preferences. When arranging reference types according to level, most of the high level types are classes containing a `main` method. An example of a low reference type is again a utility class, this time for recording information about a locale.

As illustrated by these examples, Lakos's level metric is useful in sorting the components of a program (be it packages or types) in a top-down fashion, to ease exploration and to gain a bird's-eye view of the structure of a system.

Formally, an element has no level defined if it is cyclically dependent on itself. Otherwise, it has level 0 if it does not depend on any other elements. It has level 1 if it depends on other elements, but those occur in libraries. Finally, if it depends on another element at level n then it has level $n+1$.

Now note that this definition is truly generic: it is the same whether we are talking about dependencies between packages or dependencies between types. Consequently we can define an abstract class, which is a superclass both of reference types and packages. All we need to do to use the metric on particular examples is override the abstract definition of dependency, once in *MetricRefType* and once in *MetricPackage*.

The abstract class (named *MetricElement*) is a subclass of a common supertype of *Package* and *RefType*, namely *Element*. The first method we define is *getADependency*: this returns another element that **this** depends on; and the definition needs to be overridden both in *MetricPackage* and in *MetricRefType*.

Next, we define the notion of a *Source Dependency*, simply restricting normal dependency to source elements. We impose that restriction because it does not make sense to trace dependencies through all the libraries: we are interested in the structure of the source itself. It remains to fill in the dots in the class definition below by defining the level metric itself, and we shall do that below.

```
class MetricElement extends Element {

  MetricElement getADependency() {
      result=this // to be overridden
  }

  MetricElement getADependencySrc() {
      result = this.getADependency() and result.fromSource()
  }
   ...
}
```

We only define the level of elements in the source. Furthermore, as stated in the above definition, if an element participates in a dependency cycle, then it does not have a level. Here we test that by taking the transitive closure of *getADependencySrc*: in other words, we only consider cycles through source elements. Next come three cases: first, if an element depends on no other elements, it has level 0. Second, if it depends on some other elements but none of those are in source, it has level one. Finally, if it depends on level n, it has level $n+1$:

```
  int getALevel() {
        this.fromSource() and
        not(this.getADependencySrc+()=this) and
        (   (not(exists(MetricElement t | t=this.getADependency()))
             and
             result=0)
        or (not(this.getADependency().fromSource()) and
             exists(MetricElement e | this.getADependency() = e) and
             result=1)
        or (result = this.getADependency().getALevel() + 1) )
      }
```

Our definition of the Lakos level metric is now almost complete. The above definition of *getALevel* possibly assigns multiple levels to the same element. Therefore, we take the maximum over all those possibilities, and that is the metric we wished to define:

```
  int getLevel() {
      result = max(int d | d = this.getALevel())
  }
```

Exercise 13. The above definition of *getLevel* is in the default library; write queries to display barcharts. Do the high points indeed represent components that you would consider high-level? For types, write a query that calculates how many classes that have maximum level do not define a method named "main". ♡♡

3.3 Inheritance and method dispatch

We have introduced the class mechanism of .QL through a number of motivating examples; it is now time to take a step back and examine more closely what the precise semantics are. In this subsection we shall use minimal examples; they are artificial, but intended to bring out some subtle points in the language design.

Inheritance A class is a predicate of one argument. So for example, we can define a class named *All* that is true just of the numbers 1 through 4:

```
class All {
  All() { this=1 or this=2 or this=3 or this=4}
  string foo() { result="A"}
  string toString() { result = ((int)this).toString() }
}
```

Note that *All* does not have a superclass. Any such class that does not have an ancestor must define *toString*, just to ensure that the results of queries can be displayed. We have also defined a method named *foo*, for illustrating the details of method overriding below. The query

```
from All t select t
```

will return 1, 2, 3 and 4.

Defining a subclass means restricting a predicate by adding new conjuncts. For instance, consider the class definition below:

```
class OneOrTwo extends All {
  OneOrTwo() {this=1 or this=2 or this=5}
  string foo() { result="B"}
}
```

This class consists just of 1 and 2. That is, we take the conjunction of the characteristic predicate of *All* and the constructor. While 5 is mentioned as an alternative in the constructor, it is not satisfied by the superclass *All*. Consequently the query

```
from OneOrTwo t select t
```

returns just 1 and 2. More generally, the predicate corresponding to a class is obtained by taking the conjunction of its constructor, and the predicate corresponding to its superclass.

Because classes are logical properties, they can overlap: multiple properties can be true of the same element simultaneously. For instance, here is another subclass of *All*, which further restricts the set of elements to just 2 and 3.

```
class TwoOrThree extends All {
  TwoOrThree() {this=2 or this=3}
  string foo() { result="C"}
}
```

Note that the element 2 is shared between three classes: *All*, *OneOrTwo* and *TwoOrThree*. The overlap between subclass and superclass is natural, but here *OneOrTwo* and *TwoOrThree* are siblings in the type hierarchy. Overlapping siblings are allowed in .QL, but they can lead to nondeterminism in method dispatch, and we shall discuss that further below.

Summarising our account of classes so far, classes are predicates, and inheritance is conjunction of constructors. It is easy to see what multiple inheritance means in this setting: it is again conjunction. So for example, the following class is satisfied only by the number 2, because that is the only element that its superclasses have in common:

```
class OnlyTwo extends OneOrTwo, TwoOrThree {
  OnlyTwo() { any() }
  string foo() { result = "D" }
}
```

As remarked previously, in cases like this where the constructor is just *true*, its definition may be omitted.

A precise definition of what the predicate corresponding to a class definition can now be stated as: that predicate is the conjunction of all constructors of all its supertypes in the type hierarchy. It is not allowed to define a circular type hierarchy in .QL, so this notion is indeed well-defined. Figure 5 summarises the example so far, showing for each class what elements are satisfied, and what the value returned by *foo* is.

Method Dispatch Let us now consider the definition of method dispatch. A method definition m of class C is invoked on a value x if x satisfies the defining property of C, and there is no subclass D of C which defines a method m of the same signature, and x also satisfies D. In words, we always apply the most specialised definition.

In the above example, the query

```
from All t select t.foo()
```

returns "A", "B", "C" and "D". It returns "A" because 4 satisfies *All*, but none of the other classes. It returns "B" because 1 satisfies *OneOrTwo* but none of the other classes. Next, "C" is returned because 3 satisfies *TwoOrThree*, but not any of its subclasses. Finally, "D" appears because *OnlyTwo* is the most specific class of 2.

What happens if there are multiple most specific types? This can easily occur, as illustrated by

```
class AnotherTwo extends All {
```

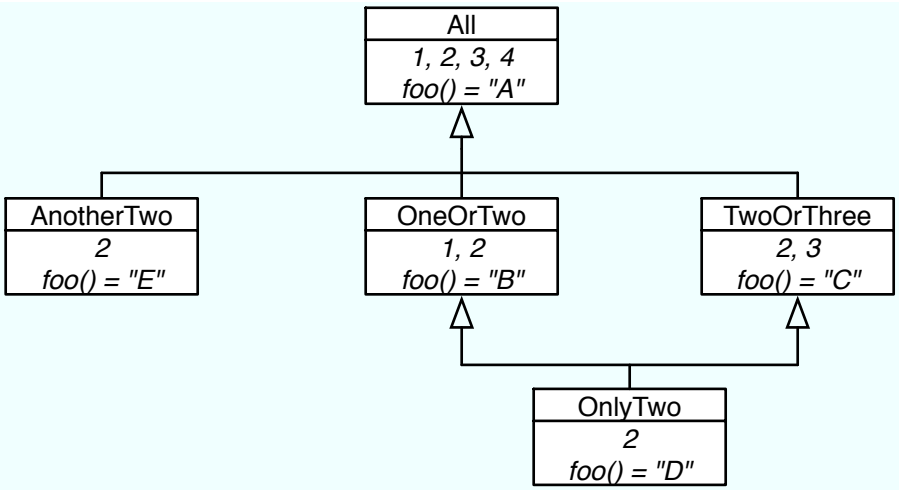

Fig. 5. Example Classes: Inheritance Hierarchy

```
  AnotherTwo() {this=2}
  string foo() { result="E" }
}
```

Now the number 2 has two most specific types, namely *OnlyTwo* and *AnotherTwo*. In such cases *all* most specific implementations are tried. In particular the query

```
from OneOrTwo t select t.foo()
```

returns "B", "D", and "E". It is quite rare for such nondeterminism to be intended, and it is therefore important to take care when designing a class hierarchy that few unintended overlaps between siblings occur. Of course it is always possible to resolve the nondeterminism by introducing another subclass that simultaneously extends all the overlapping subclasses.

Programmers who are familiar with object-oriented programming in Java may find it at first disconcerting that dispatch is entirely based on logical properties. The inheritance hierarchy is used only to build up those logical properties via conjunction and more primitive predicates. The semantics of runtime dispatch is however entirely in terms of the semantics of classes as predicates. Upon reflection, that is analogous to the way method dispatch works in Java, based on the runtime type of objects, and not at all influenced by static typing. The design of .QL is thus consistent with traditional notions of object-orientation, in that static type-checking and runtime semantics are not intertwined.

There is one small exception to the principle that dispatch is entirely a runtime phenomenon, to avoid unwanted confusion between method signatures. In deciding what method definitions to consider as candidates for dispatch, at compile time the compiler inspects the static type of the receiver (*i.e.* x in a call $x.bar(..)$) and finds the root definitions of the corresponding method: those are

definitions (of *bar*) in supertypes of the receiver type that do not override a definition in another superclass themselves. All definitions of *bar* in subtypes of the root definitions are possible candidates. As said, this is just a device to avoid accidental confusion of method names, and it is not a key element of the semantics of .QL.

In summary, method dispatch occurs in two stages, one static and one dynamic. To resolve a call $x.bar(..)$, at compile-time we determine the static type of x, say T. We then determine all root definitions of *bar* above T (methods with the same signature that do not themselves override another definition). This is the set of candidates considered for dispatch at runtime. At runtime itself, each of the candidates applies only if the value of x satisfies the corresponding predicate, and there is no more specific type that x also satisfies.

Exercise 14. Suppose the class *OnlyTwo* does not override *foo*. Does that make sense? What does the .QL implementation do in such cases? ♡

Exercise 15. Construct an example to demonstrate how dispatch depends on the static type of the receiver. ♡♡

3.4 Database Schema

.

We have claimed earlier that .QL is a general query language, which is specialised to a particular application by constructing a class hierarchy. Indeed, it is our claim that .QL can be used on *any* relational database. A key ingredient of that argument is still missing, however, and that is how the class mechanism interacts with information stored in such a relational database, and that is explained now.

Column types The primitive relations store information about the type hierarchy, class definitions, method definitions and so on. The schema for these relations is just like that found in a normal relational database, giving field names and types. The twist needed to create class definitions is that every field in fact has two types: one for the use of the underlying database (a *representation* type), and one for .QL (a *column type*).

For example, here is the schema for the table that represents method declarations.

```
methods(int id: @method,
        varchar(100) nodeName: string ref,
        varchar(900) signature: string ref,
        int typeid: @type ref,
        int parentid: @reftype ref,
        int cuid: @cu ref,
        int location: @location ref);
```

In words, we store a unique identifier for each method, a name, a signature, the return type, the declaring type, the compilation unit it lives in, and its location. The first type for each field (set in teletype font) is its representation type. For example, the unique method identifier happens to be an integer. Representation types describe the values stored in the database, but are not exposed to .QL programs, since it is undesirable to leak such low-level implementation details. As a result, each field has another type (the column type) for use in .QL, shown in italics above. Conventionally, column types start with the character '@', except for primitive types such as *string* or *int*.

The declaration of the *methods.id* field doubles as the declaration of the type *@method*: we define that type to be any value occurring in this column of the methods table. Such a type defined simultaneously with a field is called a *column type*. All the other fields have types that are references to column types that already exist elsewhere. For instance, the *cuid* field (short for Compilation Unit IDentifier) is a reference to the *@cu* type; and that type is defined in the table that represents compilation units.

Not all column types are introduced via a field declaration, however. Some of these are defined as the union of other types. For example:

```
@reftype = @interface | @class | @array | @typevariable;
```

This defines the notion of a reference type: it is an interface, or a class, or an array, or a type variable.

3.5 From Primitives to Classes

Now suppose we wish to write a new class for querying Java methods. As we have seen, there is a primitive relation *methods* one can build on. Furthermore, classes can extend column types, and this is the key that makes the connection between the two worlds. The characteristic predicate of a column type is just that a value occurs in its defining column. We can therefore define

```
class MyMethod extends @method {
  string getName() {methods(this,result,_,_,_,_)}
  string toString() {result=this.getName()}
}
```

Note how we can refer to primitive relations in the same way as we refer to classless predicates.

It should now be apparent that the design of the .QL language is independent of its application to querying Java code, of even querying source code more generally. There is a collection of primitive relations that comes with the application, and those primitive relations have been annotated with column types. In turn, those column types then form the basis of a class library that is specific to the application in hand. In principle, any existing relational database can be queried via .QL.

Of course annotating the database schema and constructing a good class library is not a trivial exercise. In the case of querying Java, the current distribution of .QL has a schema that consists of about forty primitive relations, and approximately fifty column types (there are more column types than relations because some column types are unions of others). The corresponding library of classes contains 70 class definitions, and amounts to 941 lines of .QL code (excluding white space and comments).

Exercise 16. Extend the above class definition with *getDeclaringType*. ♡

4 Implementation

In earlier sections we have seen the .QL query language, providing a convenient and expressive formalism in which to write queries over complex data. We then discussed the object-oriented features of .QL, which allow complex queries to be packaged up and reused in a highly flexible fashion. These features are essential to build up a library of queries over programs, but this begs the question of how .QL may be implemented, and it is the aim of this last section to describe the implementation strategy. We first describe the intermediate language used for .QL queries, a deductive query language known as Datalog. We then sketch the translation of .QL programs into Datalog, before briefly outlining the implementation of Datalog queries over relational databases.

4.1 Datalog

.QL is based on a simple form of logic programming known as Datalog, originally designed as an expressive language for database queries [26]. All .QL programs can be translated into Datalog, and the language draws on the clear semantics and efficient implementation strategies for Datalog. In this section we describe the Datalog language before outlining how .QL programs may be translated into Datalog. Datalog is essentially a subset of .QL, and as such we shall be using .QL syntax for Datalog programs.

Predicates A Datalog program is a set of *predicates* defining logical relations. These predicates may be recursive, which in particular allows the transitive closure operations to be implemented. A Datalog predicate definition is of the form:

predicate p(T1 x1, ..., Tn xn) { formula }

This defines a named predicate p with variables $x_1, \ldots, x_n$. In a departure from classical Datalog each variable is given a type. These restrict the range of the relation, which only contains tuples $(x_1, \ldots, x_n)$ where each x_i has the type T_i.

The body of a Datalog predicate is a logical formula over the variables defined in the head of the clause. These formulas can be built up as follows:

$$\begin{aligned} \mathit{formula} ::= {} & \mathit{predicate}(\mathit{variable}, \ldots, \mathit{variable}) \\ | \quad & \mathit{test}(\mathit{variable}, \ldots, \mathit{variable}) \\ | \quad & \mathit{variable} = \mathit{expr} \\ | \quad & \textbf{not}(\mathit{formula}) \\ | \quad & \mathit{formula}\ \textbf{or}\ \mathit{formula} \\ | \quad & \mathit{formula}\ \textbf{and}\ \mathit{formula} \\ | \quad & \textbf{exists}(\mathit{Type}\ \mathit{variable} \mid \mathit{formula}) \end{aligned}$$

That is, a formula is built up from uses of predicates through the standard logical operations of negation, disjunction, conjunction and existential quantification. In addition to predicates, *tests* are allowed in Datalog programs. A test is distinct from a predicate in that it can only be used to test whether results are valid, not generate results. An example of a test is a regular expression match. The test X **matches** "C%" is intended to match all strings beginning with "C". Evidently such a test cannot be used to generate strings, as there are infinitely many possible results, but may constrain possible values for X. In contrast, a predicate such as $\mathit{depends}(A, B)$ may generate values — in this case, the variables A and B are bound to each pair of elements for which A depends on B. In a manner of speaking, variable occurrences in a test are non-binding: such variables must also occur in a predicate.

Arguments to predicates are simply variables in Datalog, but *expressions* allow the computation of arbitrary values. Expressions are introduced through formula such as $X = Y + 1$ defining the value of a variable, and include all arithmetic and string operators. In addition, expressions allow aggregates to be introduced.

$$\begin{aligned} \mathit{expr} ::= {} & \mathit{variable} \\ | \quad & \mathit{constant} \\ | \quad & \mathit{expr} + \mathit{expr} \\ | \quad & \mathit{expr} \times \mathit{expr} \\ | \quad & \cdots \\ | \quad & \mathit{aggregate} \end{aligned}$$

Our definition of Datalog differs from usual presentations of the language in several respects. The first difference is largely inessential. While we allow arbitrary use of logical operators in formulas, most presentations requires Datalog predicates to be in *disjunctive normal form*, where disjunction can only appear at the top level of a predicate and the only negated formulas are individual predicates. However, any formula may be converted to disjunctive normal form, so this does not represent a major departure from pure Datalog. Expressions, on the other hand, are crucial in increasing the expressiveness of the language. In

pure Datalog expressions are not allowed, and this extension to pure Datalog is nontrivial, with an impact on the semantics of the language.

Datalog Programs A Datalog program contains three parts:

1. A *query*. This is just a Datalog predicate defining the relation that we wish to compute.
2. A set of user-defined, or *intensional* predicates. These predicates represent user-defined relations to be computed to evaluate the query.
3. A set of *extensional* predicates. These represent the elements stored in the database to be queried.

The general structure of a Datalog program therefore mirrors that of a .QL program. The query predicate corresponds to the query in a .QL program, while classes and methods may be translated to intensional predicates. Finally, in the context of program queries the extensional predicates define the information that it stored about the program. Examples may include the inheritance hierarchy, for instance represented as a table *hasSubtype* of each type and its direct subtypes; or the set of classes in the program.

Semantics and Recursion The semantics of Datalog program are very straightforward, in particular in comparison to other forms of logic programming such as Prolog. A key property is that termination of Datalog queries is not an issue. The simplicity of the semantics of Datalog programs (and by implication of .QL programs) is an important factor in its choice as an intermediate query language, as it is straightforward to generate Datalog code. It is worth exploring the semantics in a little more detail, however, as a few issues crop up when assigning meaning to arbitrary Datalog programs.

For our purposes, the meaning of a Datalog program is that each predicate defines a relation, or set of tuples, between its arguments. Other, more general, interpretations of Datalog programs are possible [58], but this will suffice for our purposes. An important feature is that these relations should be finite, so that they may be represented explicitly in a database or in memory. It is customary to enforce this through *range restriction*, that is to say ensuring that each variable that is an argument to a predicate should be restricted to a finite set. In our case, this is largely straightforward, as each variable is typed. Types such as @*class* or @*reftype* restrict variables to certain kinds of information already in the database, in this case the sets of classes or reference types in the program. As there can only be finitely many of these, any variable with such a type is automatically restricted. However, primitive types such as **int** are more troublesome. Indeed it is easy to write a predicate involving such variables that defines an infinite relation:

predicate p(**int** X, **int** Y) { X = Y }

This predicate contains all pairs (X, X), where X is an integer, which is infinite and therefore disallowed. As a result, the type system of .QL ensures that any

variable of primitive type is always constrained by a predicate, restricting its range to a finite set.

In the absence of recursion, the semantics of a Datalog program is very straightforward. The program can be evaluated bottom-up, starting with the extensional predicates, and working up to the query. Each relation, necessarily finite by range-restriction, can be computed from the relations it depends on by simple logical operations, and so the results of the query can be found.

The situation is more interesting in the presence of recursion. Unlike other logic programs in which evaluation of a recursive predicate may fail to terminate, in Datalog the meaning of a recursive predicate is simply given by the least fixed point of the recursive equation it defines. As an example, consider the recursive predicate

predicate p(**int** X, **int** Y) { q(X, Y) **or** (p(X,Z) **and** q(Z,Y)) }

where q denotes (say) the relation $\{(1,2),(2,3),(3,4)\}$. Then p denotes the solution of the relation equation $P = q \cup P; q$, in which ; stands for relational composition. This is just the transitive closure of q, so the relation p is simply

$$p = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$$

This least fixed point interpretation of Datalog programs makes it easy to find the value of any predicate. For instance, consider

predicate p(**int** X) { p(X) }

This predicate would be nonterminating as a Prolog program. However, in Datalog this is just the least solution of the equation $P = P$. As every relation satisfies this equation, the result is just the empty relation!

More precisely, the meaning of a Datalog program can be defined as follows. First, break the program up into components, where each component represents a recursive cycle between predicates (formally, a strongly-connected component in the call graph). Evaluation proceeds bottom-up, starting with extensional predicates and computing each layer as a least fixed point as above.

There are two technical restrictions to the use of recursion in Datalog. The first is known as *stratification*, and is necessary to deal with negation properly. The problem can be illustrated by this simple example:

predicate p(@class X) { **not**(p(X)) }

What should this predicate mean? It is defined as its own complement, so a class lies in p iff it does not lie in p. There is no relation satisfying this property, so we cannot assign a simple relational interpretation to this program. To avoid this issue, we only consider stratified Datalog. In this fragment of Datalog, negation cannot be used inside a recursive cycle. That is, a cycle through mutually recursive predicates cannot include negation. This is not a problem in practice, and stratification is not a substantial obstacle to expressiveness.

A similar problem is posed by our use of expressions, which does not lie in the scope of classical Datalog. While expressions increase the power of the

language, their interaction with recursion is problematic. For instance, consider the following:

predicate p(**int** Y) { Y = 0 **or** (Y = Z+1 **and** p(Z)) }

Clearly 0 lies in p. Therefore 1 must also lie in p from the recursive clause, and in this manner every number n lies in p. The use of expressions in recursive calls may therefore lead to infinite relations, and thus nontermination. In .QL this may also lead to nonterminating queries, and so care must be used when using arithmetic expressions in recursive calls — if, as in the above example, the expression can create new values for each recursive call, then the query may be nonterminating.

4.2 Translating .QL

The precise semantics of .QL programs are defined by their translation into Datalog programs. The outline of this translation is quite straightforward, as the overall structure of .QL programs closely mirrors that of Datalog programs. In particular, the query in a .QL program is translated into a Datalog query, while methods and classless predicates are translated to Datalog intensional predicates.

Translating Queries The general form of a .QL() query (ignoring **order by** clauses, which merely amount to a post-processing step) is:

from T1 x1, T2 x2, ..., Tn xn
where formula
select e1, e2, ..., ek

where each e_i is an expression, and each x_i is a declared variable of type T_i.

It is straightforward to translate this to a Datalog query, which is just a standard predicate. The resulting relation has k parameters (one for each selected expression), and so the query predicate has k parameters. The variables x_1 through x_n can be introduced as local variables, defined by an existential quantifier. As a result, the Datalog translation of the above query, omitting types, is:

```
predicate p(res1, res2, ..., resk) {
  exists (T1 x1, T2 x2, ..., Tn xn |
      formula2
      and res1 = e1
      and res2 = e2
      and ...
      and resk = ek
    )
}
```

where *formula*2 is obtained from *formula* by translating away all non-Datalog features of .QL, and in particular method calls, as described below

Translating Classes Classes are translated into individual Datalog predicates, representing constructors, methods and class predicates. In most cases the translation is straightforward, the key aspect being the translation of method calls.

A .QL method is merely a particular kind of Datalog predicate involving two special variables — **this** and **result**. The **this** variable holds the value that is a member of the class, while the **result** variable holds the result of the method. As an example, consider the following method to compute a string representation of the fully qualified name of a type:

```
class RefType {
  ...

  string getQualifiedName() {
    result = this.getPackage() + "." + this.getName()
  }

  ...
}
```

This is translated into the following Datalog predicate

```
predicate RefType_getQualifiedName(RefType this, string result) {
  exists(string package, string type |
        RefType_getName(this, type)
    and RefType_getPackage(this, package)
    and result = type + "." + package
  )
}
```

This extends to methods taking an arbitrary number of parameters, in which case the two parameters **this** and **result** are simply added to the list of parameters. Apart from the translation of method calls, which we will describe shortly, there are few differences between the body of the method and the body of the generated predicate. Class predicates are similar, but as predicates do not return a value, the **result** variable is not used. For instance, the method

```
class RefType {
  ...

  predicate declaresField(string name) {
    this.getAField().getName() = name
  }

  ...
}
```

is translated to the following Datalog predicate:

```
predicate RefType_declaresField(RefType this, string name) {
```

```
  exists(Field field |
        RefType_getAField(this, field)
    and Field_getName(field, name)
  )
}
```

Both examples highlight one of the crucial advantages of .QL methods over Datalog predicates, in addition to extensibility. In Datalog, it is necessary to name each intermediate result, as is the case with the field in the above example. In contrast, methods returning (many) values allow queries to be written in a much more concise and readable manner.

Finally, constructors are simply translated to Datalog predicates denoting the character of each class. For instance, consider the definition of anonymous Java classes:

```
class AnonymousClass extends NestedClass {
  AnonymousClass() { this.isAnonymous() }
}
```

The constructor for this class is translated into a predicate defining precisely those elements that are nested classes. These are the Java elements that are nested classes, additionally satisfying the *isAnonymous* predicate:

```
predicate AnonymousClass(NestedClass this) {
  NestedClass_isAnonymous(this)
}
```

In the above, the type of **this** enforces the fact that an anonymous class must be nested. When a class inherits from multiple classes, the translation is a little more complicated. Consider the class *Interface*, with no constructor:

```
class Interface extends RefType, @interface {
  ...
}
```

This class extends both *RefType* and the column type *@interface*, and thus an element is an *Interface* exactly when it is both a *RefType* and an *@interface*. This is encoded in the generated constructor for *Interface*:

```
Interface(RefType this) { @interface(this) }
```

Despite the fact that *Interface* does not define a constructor, it restricts the range of values that it encompasses by inheritance, and thus this characteristic predicate must be generated.

Translating Method Calls In the above, we have described the translation of methods into Datalog predicates with extra arguments **this** and **result**, and informally shown some method calls translated into calls to the generated predicates. In our examples, the translation was straightforward, as the type of the receiver was known, and so it was immediately apparent which predicate should

be called. However, as .QL uses virtual dispatch, the method that is actually used depends on the value it is invoked on, and this translation scheme cannot work in general.

To illustrate the translation of method dispatch in .QL, let us recall the class hierarchy defined in Section 3, simplified for this example:

```
class All {
  All() { this=1 or this=2 or this=3 or this=4 }
  string foo() { result = "A" }
}

class OneOrTwo extends All {
  OneOrTwo() { this=1 or this=2 }
  string foo() { result = "B" }
}

class TwoOrThree extends All {
  TwoOrThree() { this=2 or this=3 }
  string foo() { result="C" }
}
```

As we have seen previously, each of the implementations of *foo* is translated into a Datalog predicate:

```
predicate All_foo(All this, string result) { result = "A" }
predicate OneOrTwo_foo(OneOrTwo this, string result) { result = "B" }
predicate TwoOrThree_foo(TwoOrThree this, string result) { result = "C" }
```

However, when a call to the *foo* method is encountered, the appropriate methods must be chosen, depending on the value of the receiver of the call. .QL method dispatch selects the most specific methods, of which there may be several due to overlapping classes, and returns results from all most specific methods. Only the most specific methods are considered, so that a method is not included if it is ooverriddenby a matching method.

This virtual dispatch mechanism is implemented by defining a *dispatch predicate* for each method, testing the receiver against the relevant types and choosing appropriate methods. Testing the type of the receiver is achieved by invoking the characteristic predicate for each possible class, leading to the following dispatch method for *foo*:

```
predicate Dispatch_foo(All this, string result) {
      OneOrTwo_foo(this, result)
  or TwoOrThree_foo(this, result)
  or (not(OneOrTwo(this)) and not(TwoOrThree(this))
      and All_foo(this, result))
}
```

Let us examine this dispatch predicate a little more closely. The parameter **this** is given type *All*, as this is the most general possible type in this case. The body

of the predicate consists of three possibly overlapping cases. In the first case, the *foo* method from *OneOrTwo* is called. Note that this only applies when **this** has type *OneOrTwo*, due to the type of the **this** parameter in *OneOrTwo*. As *OneOrTwo* does not have any subclasses, its *foo* method cannot be ooverridden and whenever it is applicable it is necessarily the most specific. The second case is symmetrical, considering the class *TwoOrThree*. These cases are overlapping, if **this** = 2, and so the method can return several results. Finally, the third case is the "default" case. If **this** did not match either of the specific classes *OneOrTwo* or *TwoOrThree*, the default implementation in *All* is chosen.

Suppose now that we extend the example to the full class hierarchy shown in Figure 5, as follows:

```
class OnlyTwo extends OneOrTwo, TwoOrThree {
  foo() { result = "D" }
}
class AnotherTwo extends All {
  AnotherTwo() { this = 2 }
  foo() { result = "E" }
}
```

In this new hierarchy, we added two classes with exactly the same characteristic predicate. This changes method dispatch whenever **this** = 2, as the newly introduced methods are more specific than previous methods for this case. To extend the previous example with these new classes, we simply lift out the new implementations of *foo*:

```
predicate OnlyTwo_foo(OnlyTwo this, string result) { result = "D" }
predicate AnotherTwo_foo(AnotherTwo this, string result) { result = "E" }
```

and change the dispatch predicate accordingly:

```
predicate Dispatch_foo(All this, string result) {
      OnlyTwo_foo(this, result)
  or AnotherTwo_foo(this, result)
  or (not(OnlyTwo(this))
      and OneOrTwo_foo(this, result))
  or (not(OnlyTwo(this))
      and TwoOrThree_foo(this, result))
  or (not(OneOrTwo(this))
      and not(TwoOrThree(this))
      and not (AnotherTwo(this))
      and All_foo(this, result))
}
```

The only changes, apart from the introduction of cases for the two new classes, is that the existing cases for *OneOrTwo*, *TwoOrThree* and *All* must be amended to check whether the method is indeed the most specific one.

4.3 Implementing Datalog Queries

Database Implementation The use of Datalog as an intermediate language for .QL has two benefits. The first is the simplicity of Datalog, making it straightforward to define the semantics of .QL by translation to Datalog. In addition, Datalog was designed as a query language over relational databases, and can be implemented efficiently over familiar relational query languages, in particular SQL.

A .QL program ranges over a database schema defining the relations that queries can inspect. In the translated Datalog program these just form the extensional predicates, while intensional predicates define new relations that are computed by querying this data. Such Datalog queries can be translated directly into SQL statements, and the aim of this section is to introduce this translation.

For each defined predicate, say

```
predicate p(A x, B y) {
  exists (C z | q(x, z) and r(z, y))
}
```

a new table (also called `p`) is created. The table `p` has columns `x` and `y`, corresponding to the query fields. The types of these columns can be deduced from the .QL column types, but are not identical: .QL allows for rich user-defined column types such as *@class*, while databases typically only provided simple scalar types such as integers or characters. Primitive types can be represented directly in the database, naturally, but for user-defined types some representation (typically based on unique identifiers) must be chosen.

This table is then populated with the result of the query, as computed by an SQL **SELECT** statement. The first step of this translation is to make the variable types explicit. Recall that variable types restrict the range of values that a variable can take, which must be represented in the SQL query. We therefore make these types explicit in the Datalog query, resulting in the following (untyped) query:

```
predicate p(x, y) {
      A(x) and B(y)
  and exists(z | C(z) and q(x,z) and r(z,y))
}
```

This relation is essentially a join of the `q` and `r` relations, together with the type restrictions on variables. This may be computed by the following SQL statement, assuming that tables `q(a,b)` and `r(c,d)` have already been computed, as have all type tables `A(x)`, `B(x)` and `C(x)`:

```
SELECT DISTINCT q.a, r.d
FROM q
     INNER JOIN r
       ON r.c = q.b
     INNER JOIN C
       ON C.x = q.b
```

```
      INNER JOIN A
        ON q.a = A.x
      INNER JOIN B
        ON r.d = B.x
```

The first line of this query selects the `x` and `y` variables from tables `q` and `r`. The **DISTINCT** modifier is used to guarantee that the result is a set and does not contain duplicates, as SQL queries otherwise produce bags of results. The relation constructed in the **FROM** clause is simply the join of all predicates conjoined together in the predicate *p*, joining on any variables that appear in several predicates.

This implementation strategy allows arbitrary Datalog predicates to be implemented as SQL queries. A conjunction may, as we have seen above, simply be translated as an SQL join. More general formulas can be implemented by converting the body of each predicate to disjunctive normal form, in which the formula is expressed as a disjunction of conjunctions. As an example, consider the following predicate (in disjunctive normal form), ignoring types for concision:

predicate p(x, y) {
    exists (z | q(x,z) **and** r(z,y))
  or (q(x, y) **and not**(t(y)))
}

This may be translated into the following SQL query, in which the disjunction is simply turned into a union, where in addition to previous tables `t(e)` has been computed:

```
SELECT DISTINCT q.a, r.d
FROM q
      INNER JOIN r
        ON r.c = q.b
UNION

SELECT DISTINCT q.a, q.b
FROM q
WHERE NOT EXISTS
  (SELECT t.e
   WHERE t.e = q.b)
```

These examples illustrate the principles behind the translation of Datalog queries, and thus .QL programs, to SQL. The only Datalog feature that we have not considered are the use of expressions and aggregates, which are beyond the scope of these notes (note, however, that both are present in SQL, and so do not give rise insurmountable obstacles). This translation is crucial for the efficient implementation of .QL on very large data sets, thanks to the efficiency of database query optimisers. However, it is clear that .QL is far better suited to writing queries over complex data sets, such as the representations of programs, than SQL.

Recursion The translation from Datalog to SQL requires the program to be evaluated bottom-up, so that a relation is computed only when all the relations it depends on have themselves been evaluated. However, this is only possible for nonrecursive programs. Any recursive predicate will depend on itself, and thus the evaluation strategy is a little more involved. To conclude our description of the implementation of .QL we therefore outline the translation of recursive predicates. For simplicity, we exclude mutual recursion and consider only a single recursive predicate.

The most straightforward translation of recursive queries is to use recursive SQL queries as a direct translation. The SQL:1999 standard specifies *common table expressions*, with which queries that refer to their own result set may be written. However, support for common table expressions among widespread database management systems is patchy, and available implementations suffer from performance problems. As recursive queries are common when analysing programs, this application of .QL requires good performance in the implementation of recursion. As a result, we use our own implementation, based on well-known algorithms for evaluating recursive equations.

A recursive query, say (omitting types):

predicate p(x, y) { q(x, y) **or exists** (z | q(x, z) **and** p(z, y)) }

gives rise to a recursive equation of the form $p = F(p)$, where F is a function from relations to relations. In the above case the function is simply:

$$F(R) = q \cup q; R$$

That is, this function simply computes the value of the body of the predicate, replacing the recursive occurrence of p with the parameter R. The semantics of Datalog then prescribe that the value of p should be the *least* solution of the equation $p = F(p)$. To compute this, we may appeal to the *Knaster-Tarski fixpoint theorem*, which asserts that such a least solution exists, as long as F is monotonic (guaranteed in the absence of negated recursive calls), and that the solution can be obtained by iterating the F function, starting with the empty relation:

$$p = \lim_{n \to \infty} F^n(\emptyset)$$

This suggests an algorithm for computing the fixpoint:

```
1  old = ∅
2  p = F(old)
3  while (p ≠ old)
4     old = p
5     p = F(old)
```

The assignment $p = F(old)$ can be computed as a nonrecursive SQL query, this clearly provides an implementation strategy. However, it is not optimal. The successive iterations of this algorithm give the following values for p:

$$p = \emptyset$$

$$
\begin{aligned}
p &= F(\emptyset) = q \cup q; \emptyset = q \\
p &= F(q) = q \cup q; q = q \cup q^2 \\
p &= F(q \cup q^2) = q \cup q; (q \cup q^2) = q \cup q^2 \cup q^3 \\
&\ldots
\end{aligned}
$$

In general, after n iterations the value of p is $q \cup q^2 \cup \cdots \cup q^n$. The difference between the results for iterations n and $n+1$ is therefore just q^{n+1}. However, the relations q to q^n are recomputed anyway, making this algorithm expensive.

The inefficiency of the naive algorithm for evaluating recursion leads to the so-called "semi-naive" algorithm presented below [6]. The idea is to observe that at each step, we need only apply the function F to values that were newly created at the previous step. In our example, the new tuples at step n are those of q^n. In step $n+1$ we thus only need to add the relation $F(q^n)$, and keep all other tuples in the accumulated relation.

The semi-naive evaluation strategy is almost always applicable, but does impose a restriction on the predicates it is used for. More precisely, the function F corresponding to this predicate must be *distributive*, in the sense that

$$F(A \cup B) = F(A) \cup F(B)$$

This is always guaranteed for (safe) predicates with *linear* recursion, that is predicates in which there is only one recursive call per disjunct in the disjunctive normal form representation. Such predicates form the overwhelming majority of recursive predicates, apart from artificial examples, and so this is not a great restriction. In any other cases the naive strategy may be used.

The semi-naive keeps a *frontier* of tuples that were added in the last step:

```
1  p = ∅
2  frontier = F(p)
3  while (frontier ≠ ∅)
4    p = p ∪ frontier
5    newFrontier = F(frontier)
6    frontier = newFrontier \ p
```

At each step, the current frontier is added to the accumulated relation, while the new frontier is computed by applying F to the frontier from the previous iteration. This is guaranteed to contain all new tuples, but may contain some tuples already in the accumulated in the relation p. The last statement of the loop therefore removes any such tuples. The algorithm stops when no more tuples can be added. A proof of correctness of this algorithm may be found in [28].

To illustrate semi-naive evaluation, the following shows its iterations for our example predicate:

Iteration	p	*newFrontier*	*frontier*
0	$\emptyset$	q	q
1	q	$q \cup q^2$	q^2
2	$q \cup q^2$	$q \cup q^3$	q^3
3	$q \cup q^2 \cup q^3$	$q \cup q^4$	q^4
4	$q \cup q^2 \cup q^3 \cup q^4$	$q \cup q^5$	q^5
...	...	...	...

This example illustrates the efficiency gain offered by semi-naive evaluation. While the accumulated relation p naturally grows at each iteration, the frontier remains relatively constant as it contains only new tuples. The efficiency gain arises because the possibly expensive function F is only applied to the frontier, while the accumulated p is only used in inexpensive union and difference operations. Semi-naive evaluation is therefore crucial to the efficient implementation of recursion in .QL.

5 Related Work

.QL builds on a wealth of previous work by others, and it is impossible to survey all of that here. We merely point out the highlights, and give sources for further reading.

5.1 Code Queries

The idea to use code queries for analysing source code has emerged from at least three different communities: software maintenance, program analysis and aspect-oriented programming. We discuss each of those below.

Software maintenance. As early as 1984, Linton proposed the use of a relational database to store programs [39]. His system was called Omega, and implemented on top of INGRES, a general database system with a query language named QUEL. Crucially, QUEL did not allow recursive queries, and as we have seen in these notes, recursion is indispensable when exploring the hierarchical structures that naturally occur in software systems. Furthermore, Linton already observed extremely poor performance. In retrospect, that is very likely to have been caused by the poor state of database optimisers in the 1980s. Furthermore, in the implementation of .QL, we have found it essential to apply a large number of special optimisations (which are proprietary to Semmle and the subject of patent applications) in the translation from .QL to SQL.

Linton's work had quite a large impact on the software maintenance community as witnessed by follow-up papers like that on CIA (the C Information Abstraction system) [13]. Today there are numerous companies that market

products based on these ideas, usually under the banner of "application mining" or "application portfolio management". For instance, Paris-based CAST has a product named the 'Application Intelligence Platform' that stores a software system in a relational database [11]. Other companies offering similar products include ASG [3], BluePhoenix [9], EZLegacy [25], Metallect [43], Microfocus [44], Relativity [49] and TSRI [51]. A more light-weight system, which does however feature its own SQL-like query language (again, however, without recursion), is NDepend [55].

The big difference between SemmleCode and all these other industrial systems is the emphasis on agility: with .QL, *all* quality checks are concise queries that can be adapted at will, by anyone involved in the development process. Some of the other systems mentioned above have however one big advantage over the free Java-only version of SemmleCode: they offer parsers for many different languages, making it possible to store programs in relational form in the database. Indeed, large software systems are often heterogeneous, and so the same code query technology must work for many different object languages. We shall return to this point below.

Meanwhile, the drive for more expressive query languages, better suited to the application domain of searching code, gathered pace. Starting with the XL C++ Browser [32], many researchers have advocated the use of the logic programming language Prolog. In our view, there are several problems with the use of Prolog. First, it is notoriously difficult to predict whether Prolog queries terminate. Second, today's in-memory implementations of Prolog are simply not up to the job of querying the vast amounts of data in software systems. When querying the complete code for the *bonita* workflow system, the number of tuples gathered by SemmleCode is 4,349,156. In a very recent paper, Costa has demonstrated that none of the leading Prolog implementations is capable of dealing with datasets of that size. That confirms our own experiments with the XSB system, reported in [29]. A few months ago, however, Kniesel *et al.* reported some promising preliminary experiments with special optimisations in a Prolog-based system for querying large software systems [35].

A modern system that uses logic programming for code querying is JQuery, a source-code querying plugin for Eclipse [30, 42]. It uses a general-purpose language very similar to Prolog, but crucially, its use of tabling guarantees much better termination properties. It is necessary to annotate predicate definitions with mode annotations to achieve reasonable efficiency. We have resolutely excluded any such annotation features from .QL, leaving all the optimisation work to our compiler and the database optimiser. Despite the use of annotations, JQuery's performance does not scale to substantial Java projects.

Instead of using a general logic programming language like Prolog, it might be more convenient to use a language that is more specific to the domain. For instance Consens *et al.* proposed GraphLog [16], a language for querying graph structures, and showed that it has advantages over Prolog in the exploration of software systems. Further examples of domain-specific languages for code search are the relational query algebra of Paul and Prakash [48], Jarzabek's PQL [31]

and Crew's ASTLog [17]. A very recent proposal in this tradition is JTL (the Java Tools Language) of Cohen *et al.* [15]. Not only is this query language specific to code querying, it is specific to querying Java code. That has the advantage that some queries can be quite concise, with concrete Java syntax embedded in queries.

By contrast, there is nothing in .QL that is specific to the domain of code querying, because its designers preferred to have a simple, orthogonal language design. This is important if one wishes to use .QL for querying large, heterogeneous systems with artifacts in many different object languages. Furthermore, the creation of dedicated class libraries goes a long way towards tailoring .QL towards a particular domain. We might, however, consider the possibility of allowing the embedding of shorthand syntax in scripts themselves. There is a long tradition of allowing such user-defined syntactic extensions in a query language, for instance [10].

Program Analysis. Somewhat independently, the program analysis community has also explored the use of logic programming, for dataflow analyses rather than the structural analyses of the software maintenance community. The first paper to make that connection is one by Reps [50], where he showed how the use of the so-called 'magic sets' transformation [7] helps in deriving demand-driven program analyses from specifications in a restricted subset of Prolog, called *Datalog* (the variant of Datalog employed here incorporates certain extensions, *e.g.* expressions and aggregates).

Dawson *et al.* [19] demonstrate how many analyses can be expressed conveniently in Prolog — assuming it is executed with tabling (like JQuery mentioned above). Much more recently Michael Eichberg *et al.* demonstrated how such analyses can be incrementalised directly, using existing techniques for incrementalisation of logic programs [23]. While this certainly improves response times in an interactive environment for small datasets, it does not overcome the general scalability problem with Prolog implementations outlined above.

Whaley *et al* [37, 59] also advocate the use of Datalog to express program analyses. However, their proposed implementation model is completely different, namely Binary Decision Diagrams (BDDs). This exploits the fact that in many analyses, there are a large number similar sets (for instance of allocation sites), and BDDs can exploit such similarity by sharing in their representation. Lhoták *et al* [38] have independently made the same observation; their system is based on relational algebra rather than Datalog.

We have not yet experimented with expressing these types of program analysis in .QL, because the Eclipse plugin does not yet store information about control flow.

Aspect-oriented programming. Of course all these independent developments have not gone unnoticed, and many of the ideas are brought together in research on aspect-oriented programming. Very briefly, an 'aspect' instruments certain points in program execution. To identify those points, one can use any of the search techniques reviewed above.

One of the pioneers who made the connection between code queries and aspects was de Volder [20]. More recently, others have convincingly demonstrated that indeed the use of logic programming is very attractive for identifying the instrumentation points [27, 47]. A mature system building on these ideas is LogicAJ [52]. In [5], the patterns used in AspectJ (an aspect-oriented extension of Java) are given a semantics by translation into Datalog queries [5].

The connection is of course also exploited in the other direction, suggesting new query mechanisms based on applications in aspect-oriented programming. For example, in [24], Eichberg *et al.* propose that XQuery is an appropriate notation for expressing many of the queries that arise in aspects. It appears difficult, however, to achieve acceptable performance on large systems, even with considerable effort [22].

Earlier this year, Morgan *et al.* proposed a static aspect language for checking design rules [45], which is partly inspired by AspectJ, marrying it with some of the advantages of code querying systems. In many ways, it is similar to JTL, which we mentioned above. Like JTL, it is tailored to the particular application, allowing concrete object syntax to be included in the queries. As said earlier, because large software systems are often heterogeneous, written in many different languages, we believe the query language itself should not be specific to the object language. Many users of .QL at first believe it to be domain-specific as well, because of the library of queries that tailor it to a particular application such as querying Java in Eclipse.

5.2 Object-oriented Query Languages

.QL is a general query language, and we have seen how one can build a class library on top of any relational database schema, by annotating its fields with column types. There exists a long tradition of research on object-oriented query languages, so it behooves us to place .QL in that context.

In the 1980s, there was a surge of interest in so-called *deductive databases*, which used logic programming as the query language. The most prominent of these query languages was Datalog, which we mentioned above. In essence, Datalog is Prolog, but without any data structures [26]; it thus lacks any object-oriented features.

Since the late 80s saw a boom in object-oriented programming, it was only natural that many attempts were made to integrate the idea of deductive databases and objects. Unfortunately a smooth combination turned out to be hard to achieve, and in a landmark paper, Ullman [57] even went so far as to state that a perfect combination is impossible.

Abiteboul *et al.* [1] proposed a notion of 'virtual classes' that is somewhat reminiscent of our normal classes [1]. However, the notion of dispatch is very different, using a 'closest match' rather than the 'root definitions' employed in .QL. Their definition of dispatch leads to brittle queries, where the static type of the receiver can significantly change the result. In our experience, such a design makes the effective use of libraries nearly impossible.

Most later related work went into modelling the notion of object-identity in the framework of a query language, *e.g.* [2, 40]. In .QL that question is sidestepped because there is no object identity: a class is just a logical property. From that then follows the definition of inheritance as conjunction, and the disarmingly simple definition of virtual dispatch. Previous works have had much difficulty in defining an appropriate notion of multiple inheritance: here it is just conjunction.

6 Conclusion

We have presented .QL, a general object-oriented query language, through the particular application of software quality assessment. While this is the only concrete application we discussed, it was shown how, through appropriate annotation of the fields in a normal relational database schema with column types, one can build a library of queries on top of any relational database.

The unique features of .QL include its class mechanism (where inheritance is just logical 'and'), its notion of virtual method dispatch, nondeterministic expressions, and its adoption of Dijkstra's quantifier notation for aggregates. Each of these features contributes to the fun of playing with queries in .QL.

We hope to have enthused the reader into further exploring the use of .QL. A rich and interesting application area is the encoding of rules that are specific to an application domain. We have already done so for J2EE rules [53], but that only scratches the surface. Another application, which we hinted at in one of the exercises, is the use of .QL to identify opportunities for refactoring.

References

1. Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. Methods and rules. In Peter Buneman and Sushil Jaodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 32–41. ACM Press, 1993.
2. Foto N. Afrati. On inheritance in object oriented datalog. In *International Workshop on Issues and Applications of Database Technology (IADT)*, pages 280–289, 1998.
3. ASG. ASG-becubic™ for understanding and managing the enterprise's application portfolio. Product description on company website at http://asg.com/products/product_details.asp?code=BSZ, 2007.
4. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer, 2006.
5. Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of static pointcuts in AspectJ. In Matthias Felleisen, editor, *Principles of Programming Languages (POPL)*, pages 11–23. ACM Press, 2007.

6. Isaac Balbin and Kotagiri Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3):259–262, 1987.
7. François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986*, pages 1–16. ACM, 1986.
8. Victor Basili, Lionel Brand, and Walcelio Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–760, 1996.
9. BluePhoenix. IT discovery. Product description available from company website at: http://www.bphx.com/Discovery.cfm, 2004.
10. Luca Cardelli, Florian Matthes, and Martín Abadi. Extensible grammars for language specialization. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Database Programming Languages*, pages 11–31. Springer, 1993.
11. Cast. Application intelligence platform. Product description on company website at: http://www.castsoftware.com, 2007.
12. Checkstyle. Eclipse-cs: Eclipse checkstyle plug-in. Documentation and download at http://eclipse-cs.sourceforge.net/, 2007.
13. Yih Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.
14. Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
15. Tal Cohen, Joseph Gil, and Itay Maman. JTL - the Java Tools Language. In *21st Annual Conference on Object-oriented Programming, systems languages and applications (OOPSLA '06)*, pages 89–108. ACM Press, 2006.
16. Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 138–156. ACM Press, 1992.
17. Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997.
18. David P. Darcy, Sandra A. Slaughter, Chris F. Kemerer, and James E. Tomayko. The structural complexity of software: an experimental test. *IEEE Transactions on Software Engineering*, 31(11):982–995, 2005.
19. Stephen Dawson, C. R. Ramakrishnan, and David Scott Warren. Practical program analysis using general purpose logic programming systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 117–126. ACM Press, 1996.
20. Kris de Volder. Aspect-oriented logic meta-programming. In Pierre Cointe, editor, *REFLECTION*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 1999.
21. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1990.
22. Michael Eichberg. *Open Integrated Development and Analysis Environments*. PhD thesis, Technische Universität Darmstadt, 2007. http://elib.tu-darmstadt.de/diss/000808/.
23. Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. Automatic incrementalization of prolog based static analyses. In Michael

Hanus, editor, *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007*, volume 4354 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2007.
24. Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In *Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *Springer Lecture Notes in Computer Science*, pages 366–381, 2004.
25. EZLegacy. EZ Source™. Product description on company website at http://www.ezlegacy.com, 2007.
26. Hervé Gallaire and Jack Minker. *Logic and Databases*. Plenum Press, New York, 1978.
27. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-Oriented Software Development*, pages 60–69. ACM Press, 2003.
28. Elnar Hajiyev. CodeQuest: Source Code Querying with Datalog. MSc Thesis, Oxford University Computing Laboratory, September 2005. Available at http://progtools.comlab.ox.ac.uk/projects/codequest/.
29. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: scalable source code queries with Datalog. In Dave Thomas, editor, *Proceedings of ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006.
30. Doug Janzen and Kris de Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003.
31. Stan Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215, 1998.
32. Shahram Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Tsuyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, and Richard Helm. Architecture of the XL C++ browser. In *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 369–379. IBM Press, 1992.
33. JFreeChart. Website with documentation and downloads. http://www.jfree.org/jfreechart/, 2007.
34. Anne Kaldewaij. *The Derivation of Algorithms*. Prentice Hall, 1990.
35. Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *LATE R07 – Linking Aspect Technology and Evolution*. ACM, March 12 2007. http://www.cs.uni-bonn.de/~gk/papers/knieselHannemannRho-late07.pdf.
36. John Lakos. *Large-Scale C++ Software Design*. Addison Wesley, 1996.
37. Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of PODS*, pages 1–12. ACM Press, 2005.
38. Ondrej Lhoták and Laurie Hendren. Jedd: A BDD-based relational extension of Java. In *Programming Language Design and Implementation (PLDI)*, pages 158–169, 2004.
39. Mark A. Linton. Implementing relational views of programs. In Peter B. Henderson, editor, *Software Development Environments (SDE)*, pages 132–140, 1984.
40. Mengchi Liu, Gillian Dobbie, and Tok Wang Ling. A logical foundation for deductive object-oriented databases. *ACM Transactions on Database Systems*, 27(1):117–151, 2002.
41. Robert C. Martin. *Agile Software Development, Principles, Patterns and Practices*. Prentice Hall, 2002.

42. Edward McCormick and Kris De Volder. JQuery: finding your way through tangled code. In *Companion to OOPSLA*, pages 9–10. ACM Press, 2004.
43. Metallect. IQ server. Product description on company website at http://www.metallect.com/what-we-offer/technology/, 2007.
44. MicroFocus. Application portfolio management. Product description on company website at http://www.microfocus.com/Solutions/APM/, 2007.
45. Clint Morgan, Kris De Volder, and Eric Wohstadter. A static aspect language for checking design rules. In Oege De Moor, editor, *Aspect-Oriented Software Development (AOSD '07)*, pages 63–72, 2007.
46. Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.
47. Klaus Ostermann, Mira Mezini, and Christoph Bockish. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240, 2005.
48. Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996.
49. Relativity. Application analyzer^{TM}. Product description on company website at http://www.relativity.com/pages/applicationanalyzer.asp, 2007.
50. Thomas W. Reps. Demand interprocedural program analysis using logic databases. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, volume 296 of *International Series in Engineering and Computer Science*, pages 163–196. Kluwer, 1995.
51. The Software Revolution. Janus technology^{TM}. Product description on company website at http://www.softwarerevolution.com/, 2007.
52. Tobias Rho, Günter Kniesel, Malte Appeltauer, and Andreas Linder. LogicAJ. http://roots.iai.uni-bonn.de/research/logicaj/people, 2006.
53. Semmle Ltd. Company website with free downloads, documentation, and discussion forums. http://semmle.com, 2007.
54. Semmle Ltd. Installation instructions for this tutorial. http://semmle.com/gttse-07, 2007.
55. Patrick Smacchia. NDepend. Product description on company website at http://www.ndepend.com, 2007.
56. Diomides D. Spinellis. *Code Quality: the Open Source Perspective*. Addison-Wesley, 2007.
57. Jeffrey D. Ullman. A comparison between deductive and object-oriented database systems. In *2nd International Conference on Deductive and Object-Oriented Databases*, Springer Lecture Notes in Computer Science, pages 263–277, 1991.
58. Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.
59. John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog and binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

Appendix: Answers to the Exercises

Exercise 1. Run the query to find suspicious declarations of `compareTo` in JFreeChart. You can do that in a number of ways, but here the nicest way to look at the results is as a table, so use the run button marked with a table at the top right-hand side of the Quick Query window. You will get two results, and you can navigate to the relevant locations in the source by double-clicking. Are both of them real bugs? ♡

Answer: The table is not shown here. One of the matches, the class named `PieLabelRecord`, is indeed an example where `compareTo` and `equals` are inconsistent. The `compareTo` method will return 0 whenever the `baseY` values are equal, but `equals` is inherited from `Object` and so compares object identity. The other match `Outlier` is not a bug: in fact consistency between `equals` and `compareTo` is clearly assured because `compareTo` calls `equals`.

Exercise 2. Write a query to find all methods named `main` in packages whose names end with the string `demo`. You may find it handy to use the predicate *string.matches*("%demo") (as is common in query languages, % is a wildcard matching any string). ♡

Answer: We want to select a method, so that is what goes in the **from** clause. Next, we want those methods to have name "main" and occur in a package with a name that matches the given pattern. Note the repeated use of dispatch on the result of methods. If you tried to write the same query in Prolog, you would have to give a name to each of those intermediate results, considerably cluttering the query.

```
from Method m
where m.hasName("main") and
      m.getDeclaringType().getPackage().getName().matches("%demo")
select m.getDeclaringType().getPackage(),
       m.getDeclaringType(),
       m
```

Exercise 3. The above queries show how to find types that define a method named "equals", and how to find types that do not have such a method. Write a query picking out types that define at least *one* method which is not called "equals". ♡

Answer: This query is more verbose, but straightforward. We use **exists** to find a method and test that its name is not "equals":

```
from Class c
where exists (Method m | m = c.getACallable()
                         and not (m.hasName("equals")))
select m
```

Note that *getACallable* returns several results, so this succeeds if at least one of the methods is not called "equals".

Exercise 4. Continuing Exercise 2.1. You will have found that one class represents a real bug, whereas the other does not. Refine our earlier query to avoid such false positives. ♡

Answer: We exclude declarations of `compareTo` that make a call to `equals`:

```
from Class c, Method compare
where compare.getDeclaringType()=c and
      compare.hasName("compareTo") and
      not(c.declaresMethod("equals")) and
      not(compare.getACall().hasName("equals"))
select c.getPackage(),c,compare
```

An interesting point concerns the fact that the method *getACall* is nondeterministic. Negating the nondeterministic call means that *none* of the methods called by *compare* has name "equals".

Exercise 5. Write a query to find all types in JFreeChart that have a field of type *JFreeChart*. Many of these are test cases; can they be excluded somehow? ♡

Answer: Inspecting the results of the obvious query (the one below without the extra conjunct in the **where** clause), it is easy to see that all of the test cases are in fact subtypes of `TestCase`, so that is the condition we use to exclude them:

```
from RefType t
where t.getAField().getType().hasName("JFreeChart")
      and
      not t.getASupertype().hasName("TestCase")
select t
```

Exercise 6. There exists a method named *getASuperType* that returns *some* supertype of its receiver, and sometimes this is a convenient alternative to using *hasSubtype*. Uses of methods such as *getASuperType* that return an argument can be chained too. Using $x.getASuperType*()$, write a query for finding all subtypes of `org.jfree.chart.plot.Plot`. Try to use no more than one variable. ♡

Answer: Again, note how the use of nondeterministic methods leads to very concise queries:

```
from RefType s
where s.getASupertype*().hasName("Plot")
select s
```

Exercise 7. When a query returns two program elements plus a string you can view the results as an edge-labelled graph by clicking on the graph button (shown below). To try out that feature, use chaining to write a query to depict the hierarchy above the type `TaskSeriesCollection` in package `org.jfree.data.gantt`. You may wish to exclude `Object` from the results, as it clutters the picture. Right-clicking on the graph view will give you a number of options for displaying it. ♡

Answer: First, find the `TaskSeriesCollection` type, and name it *tsc*. Now we want to find pairs s and t that are supertypes of *tsc*, such that furthermore t is a direct supertype of s. Finally, we don't want to consider `Object`, so that is our final conjunct. If we now select the pair (s, t) that becomes an edge in the depicted graph:

```
from RefType tsc, RefType s, RefType t
where tsc.hasQualifiedName("org.jfree.data.gantt","TaskSeriesCollection")
      and
      s.hasSubtype*(tsc)
      and
      t.hasSubtype(s)
      and
      not(t.hasName("Object"))
select s,t
```

Exercise 8. Display the results of the above query as pie chart, where each slice of the pie represents a package and the size of the slice the average number of methods in that package. To do so, use the *run* button marked with a chart, and select 'pie chart' from the drop-down menu. ♡

Answer: No comment; just an exercise to play with!

Exercise 9. Not convinced that metrics are any good? Run the above query; it will be convenient to display the results as a bar chart, with the bars in descending order. To achieve that sorting, add "**as** s **order by** s **desc**" at the end. Now carefully inspect the packages with high instability. Sorting the other way round (using **asc** instead of **desc**) allows you to inspect the stable packages. ♡

Answer: The most unstable packages are precisely the *experimental* ones in JFreeChart. The most stable package of all is `java.lang`. Amazing that such a simple metric can make such accurate predictions!

Exercise 10. The following questions are intended to help reinforce some of the subtle points about aggregates; you could run experiments with SemmleCode to check them, but really they're just for thinking.

1. What is $\mathbf{sum}(\mathbf{int} i | i = 0 \ \mathbf{or}\ i = 0 | 2)$?
2. Under what conditions on p and q is this a true equation?

$$\mathbf{sum}(\mathbf{int}\ i \mid p(i)\ \mathbf{or}\ q(i)) = \mathbf{sum}(\mathbf{int}\ i \mid p(i)) + \mathbf{sum}(\mathbf{int}\ i \mid q(i))$$

♡

Answer:

1. It's just 2. You can use normal logical equivalences to manipulate the range condition in an aggregate.
2. This equation is true only if p and q are disjoint, that is: $\forall i : \neg(p(i) \wedge q(i))$.

Exercise 11. Queries can be useful for identifying refactoring opportunities. For example, suppose we are interested in finding pairs of classes that could benefit by extracting a common interface or by creating a new common superclass.

1. As a first step, we will need to identify *root definitions:* methods that are not overriding some other method in the superclass. Define a new .QL class named *RootDefMethod* for such methods. It only needs to have a constructor, and no methods or predicates.
2. Complete the body of the following classless predicate:

   predicate similar(RefType t, RefType s, Method m, Method n) { ... }

   It should check that m is a method of t, n is a method of s, and m and n have the same signature.
3. Now we are ready to write the real query: find all pairs (t, s) that are in the same package, and have more than one root definition in common. All of these are potential candidates for refactoring. If you have written the query correctly, you will find two types in JFreeChart that have 99 root definitions in common.
4. Write a query to list those 99 commonalities.

♡

Answer:

1. The class for root definitions is:

```
class RootDefMethod extends Method {
    RootDefMethod() { not exists(Method m | overrides(this, m)) }
}
```

2. The definition of the predicate can be completed as follows:

```
predicate similar(RefType t, RefType s, Method m, Method n) {
  m.getDeclaringType() = t and n.getDeclaringType() = s
  and m.getSignature() = n.getSignature()
}
```

3. Finally, the required query is shown below. To try out the answer, just type the class definition, the predicate and the query all together in the Quick Query window. (Warning: this query takes a while to execute.)

```
from RefType t, RefType s, int c
where t.getPackage() = s.getPackage()
      and
      t.getQualifiedName() < s.getQualifiedName()
      and
      c = count(RootDefMethod m, RootDefMethod n | similar(t,s,m,n))
      and
      c > 1
select c, t.getPackage(), t,s order by c desc
```

4. This is a simple re-use of the predicate *similar* defined above:

```
from RefType t, RefType s, RootDefMethod m, RootDefMethod n
where t.hasName("CategoryPlot") and s.hasName("XYPlot")
      and
      t.getPackage() = s.getPackage()
      and
      similar(t,s,m,n)
select m,n
```

Exercise 12. We now explore the use of factories in JFreeChart.

1. Write a query to find types in JFreeChart whose name contains the string "Factory."
2. Write a class to model the Java type `JFreeChart` and its subtypes.
3. Count the number of constructor calls to such types.
4. Modify the above query to find violations in the use of a `ChartFactory` to construct instances of `JFreeChart`.

5. There are 53 such violations; it is easiest to view them as a table. The interesting ones are those that are not in tests or demos. Inspect these in detail — they reveal a weakness in the above example, namely that we may also wish to make an exception for *this* constructor calls. Modify the code to include that exception. Are all the remaining examples tests or demos?

♡

Answer:

1. Here is a query to find factories in JFreeChart:

   ```
   from RefType t
   where t.getName().matches("%Factory%")
   select t
   ```

   We shall use the first result, `ChartFactory`, in the remainder of this exercise.
2. The class just has a constructor and no methods or predicates. The constructor says that **this** has a supertype named `JFreeChart`. If desired, that could be refined by using a qualified name rather than a simple name.

   ```
   class JFreeChart extends RefType {
     JFreeChart() { this.getASupertype*().hasName("JFreeChart") }
   }
   ```

3. We want calls where the callee is a constructor of a *JFreeChart* type:

   ```
   select count(Call c | c.getCallee() instanceof Constructor and
       c.getCallee().getDeclaringType() instanceof JFreeChart)
   ```

   A shorter alternative (which does however require you to know the class hierarchy quite well) is

   ```
   select count(ConstructorCall c | c.getCallee().getDeclaringType()
                                    instanceof
                                    JFreeChart)
   ```

   The answer is 88.
4. The definitions are very similar to the ones in the *ASTFactory* example:

   ```
   class ChartFactory extends RefType {
     ChartFactory() { this.getASupertype*().hasName("ChartFactory") }
     ConstructorCall getAViolation() {
         result.getType() instanceof JFreeChart and
         not(result.getCaller().getDeclaringType()
                                  instanceof ChartFactory) and
         not(result instanceof SuperConstructorCall)
     }
   }

   from ChartFactory f, Call c
   ```

```
where c = f.getAViolation()
select c.getCaller().getDeclaringType().getPackage(),
       c.getCaller().getDeclaringType(),
       c.getCaller(),
       c
```

5. Change the *getAViolation* definition to:

```
ConstructorCall getAViolation() {
    result.getType() instanceof JFreeChart and
    not(result.getCaller().getDeclaringType()
                          instanceof ChartFactory) and
    not(result instanceof SuperConstructorCall or
        result instanceof ThisConstructorCall)
}
```

No, there are still two matches in the package `org.jfree.chart.plot`. One of them says "An initial quick and dirty"; both matches seem to be real mistakes. The other 49 are all in packages that do not use the factory at all, so that is probably intended.

Exercise 13. The above definition of *getLevel* is in the default library; write queries to display barcharts. Do the high points indeed represent components that you would consider high-level? For types, write a query that calculates how many classes that have maximum level do not define a method named "main". ♡

Answer: The level metric is surprisingly effective in finding components that are high-level in the intuitive sense.

```
from MetricPackage p, float c
where p.fromSource() and c = p.getLevel()
select p, c order by c desc
```

The following query calculates what proportion of the highest-level types do not define a method named "main":

```
predicate maxLevel(MetricRefType t) {
   t.fromSource() and
   t.getLevel() = max(MetricRefType t | | t.getLevel())
}

from float i, float j
where
  i =count(MetricRefType t | maxLevel(t) and
                        not(t.getACallable().hasName("main")))
  and
```

```
  j = count(MetricRefType t | maxLevel(t))
select i/j
```

About 24% of high-level matches do not define a "main" method.

Exercise 14. Suppose the class *OnlyTwo* does not override *foo*. Does that make sense? What does the .QL implementation do in such cases? ♡

Answer: There is then a choice of two different implementations that could be overridden. At first it might seem that it makes sense to take their disjunction, but clearly that is wrong as subclassing means conjunction. The implementation forbids such cases and insists that *foo* be overridden to ensure a unique definition is referenced.

Exercise 15. Construct an example to demonstrate how dispatch depends on the static type of the receiver. ♡

Answer: We need two root definitions that have the same signature. For instance, in the class hierarchy below, there are root definitions of *foo* both in class *B* and in class *C*:

```
class A {
  A() { this=1 }
  string toString() { result="A"}
}

class B extends A {
  string foo() { result="B" }
}

class C extends A {
  string foo() { result="C"}
}

from C c select c.foo()
```

The answer of the query is just "C". If *foo* was also declared in class *A*, then that would be the single root definition, and "B" would also be an answer.

Exercise 16. Extend the above class definition with *getDeclaringType*. ♡

Answer: The definition of *getDeclaringType* is just a minor variation on the definition of *getName* we saw earlier:

```
class MyMethod extends @method {
  string getName() { methods(this,result,_,_,_,_)}
  string toString() { result = this.getName() }
  RefType getDeclaringType() { methods(this,_,_,_,result,_,_)}
}
```