Keynote Address: .QL for Source Code Analysis

Oege de Moor, Mathieu Verbaere, Elnar Hajiyev,
Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, Julian Tibble
Semmle Limited
Magdalen Centre, Oxford Science Park
Robert Robinson Avenue, Oxford OX4 4GA, UK
{oege,mathieu,elnar,pavel,torbjorn,neil,damien,julian}@semmle.com

Abstract

Many tasks in source code analysis can be viewed as evaluating queries over a relational representation of the code. Here we present an object-oriented query language, named .QL*, and demonstrate its use for general navigation, bug finding and enforcing coding conventions. We then focus on the particular problem of specifying metrics as queries.*

1. Introduction

Source code analysis involves answering questions about source, and indeed many researchers have taken that as the starting point for phrasing analyses as *queries* over a relational representation of the code. In this paper, we pick up that theme, and we present .QL, a general-purpose query language that is particularly suited to expressing analysis tasks.

The design of .QL is the result of combining ideas from different areas of computer science:

SQL First, .QL is a query language, and we have made it similar to SQL, in an attempt to lower the barrier for developers to learn it. The similarity is however purely syntactic, as the semantic basis is quite different.

Datalog Second, .QL is based on Datalog, a very simple form of logic programming that has an elegant least-fixpoint semantics. Datalog queries can be recursive, and that is important for queries over the inheritance hierarchy or the call graph. Datalog originated in database theory [11], and it has been proposed as a basis for program analysis several times, in particular by Tom Reps [27] and more recently by John Whaley *et al.* [18, 34].

Eindhoven Quantifier Notation Third, for program analysis it is often necessary to compute some metric property. The way such metric computations would be expressed in SQL is painful, involving complex constructs such as *group-by* that are only mastered by experts. The Eindhoven school of Edsger W. Dijkstra [9, 16] have proposed an elegant notation to express these quantifications, but for a different purpose, namely reasoning by fountain pen on paper. .QL is the first query language to adopt the Eindhoven Quantifier Notation as syntax in executable scripts. Later on in this paper, we shall see this notation at work in defining various metrics computations.

Classes are Predicates Fourth, object-orientation is crucial for writing reusable queries that can be shipped in libraries. The question, then, is what semantic model to adopt for an object-oriented query language. .QL takes a disarmingly simple and consistent view, namely that classes are predicates, and inheritance is implication. This has the advantage of simplicity, but the rigorous application of these principles leads to consequences that may seem at first surprising, in particular the need for nondeterministic expressions. Admitting nondeterministic expressions does, however, enable far more concise queries, and thus a better language design.

.QL has an industrial-strength implementation, SemmleCode, which includes an editor with autocompletion and online error checking, as well as a large number of optimisations. Those optimisations are essential, as a naive implementation makes most queries intractable. This paper, however, focuses solely on the language design.

The remainder of the paper is structured as follows. First, we give a general introduction to the .QL language, surveying its most salient features. Next, we zoom in on the particular application of computing software metrics, and we demonstrate how even quite complex metrics definitions can be concisely expressed. Finally, we discuss related work and conclude.

2. Introducing .QL by Example

Our introduction to .QL will involve queries over the codebase of *abc*, an extensible AspectJ compiler [4]. The code queried includes that of abc itself, plus the Polyglot framework for extensible Java compilers [26], and the Soot framework for bytecode analysis [32]. In all, the source involves 572,689 lines of source code (including comments), and 3974 reference types. By default, all the program elements in libraries that are directly referred to in source are loaded as well, adding another 318 types. Below we refer to the complete working set of source plus these library types as *abcplus*.

2.1. Select statements

Suppose we wish to locate all classes that declare a method named *equals*, but which do not declare a method named *hashCode*. Often that is a bug: when defining a new notion of equality, the hash function must be accordingly modified. The following query constructs a new relation, where each tuple consists of the package containing an erroneous class *c*, and that erroneous class *c* itself:

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

Note that this appears quite similar to an SQL select statement. However, the order of the components is different: in .QL, variables are declared before they are used. Apart from increased readability, it also enables better editor support, in particular for autocompletion.

Most users of .QL will start by writing *select* statements such as these. The .QL implementation provides a variety of ways of viewing results, including as a tree (for each package, show the classes found) and as a table (with two columns). Queries with an appropriate return type can also be viewed as charts or graphs.

The above query returns 28 results on *abcplus*. In fact, there are 2 classes in *abc* itself that violate the above design rule. Checking such design rules that are not enforceable by the Java type system is one of the prime applications of .QL.

2.2. Predicates

Queries can be named and parameterised by wrapping them in predicates. To illustrate, we construct a graph of all the types in between two given types in *abcplus*. First, we define a test to check that there exists a path in the hierarchy from *down*, through *between* to *up*:

```
predicate between(RefType down, RefType between,
                  RefType up) {
  down.hasSupertype*(between) and
  between.hasSupertype*(up)
}
```

Here $t_1.hasSupertype(t_2)$ is a test whether t_1 has immediate supertype t_2. The use of the star in the expression $t_1.hasSupertype*(t_2)$ indicates indirection: there exists a chain (possibly of length zero) of supertypes from t_1 to t_2.

This predicate can now be used in another query to find all paths from *abc.aspectj.ast.AdviceDecl_c* (representing advice declarations) to *polyglot.ast.Node*:

```
from RefType c1, RefType c2,
     RefType node, RefType adviceDecl
where
  node.hasQualifiedName("polyglot.ast","Node") and
  adviceDecl.hasQualifiedName
          ("abc.aspectj.ast","AdviceDecl_c") and
  between(adviceDecl,c1,node) and
  between(adviceDecl,c2,node) and
  c2.hasSupertype(c1)
select c1,c2
```

In words, we first find the elements on paths from *AdviceDecl_c* to *Node*, and then we arrange them as edges.

Results of queries such as these can be viewed as graphs, and that view of the result is depicted in Figure 1. Note how in the Polyglot framework, every class has a corresponding interface; this is one of the means by which Polyglot ensures extensibility of compilers.

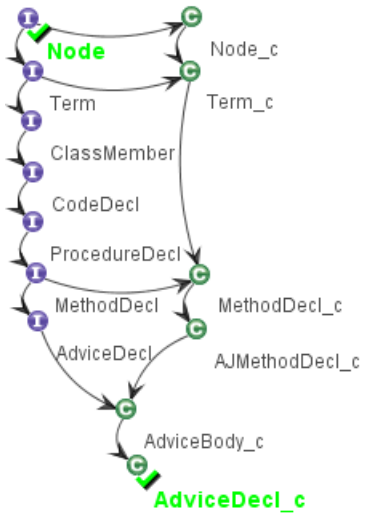

Figure 1. Paths from *Node* to *AdviceDecl_c* in the type hierarchy

2.3. Aggregates

Often we wish to compute some measure of a set of results: the sum, the size, or the average. In SQL, writing such queries can be rather painful, due to the need to use features like *group-by*, or complex nested SQL statements.

There is a very elegant notation for reasoning about such aggregates in the work of Edsger W. Dijkstra and his coworkers [9, 16], known as the *Eindhoven Quantifier Notation*. Their purpose was to use this notation in reasoning about programs, and they showed many convincing examples of how it enables concise proofs. Here we deploy the same idea directly as a query language construct.

To illustrate, here is a query for counting the number of lines in the *ast* package of *abc*:

```
from Package p
where p.hasName("abc.aspectj.ast")
select sum(CompilationUnit cu |
                cu.getPackage()=p |
                cu.getNumberOfLines())
```

The general form of an aggregate in .QL is

```
aggfunc( localvars | condition | term )
```

Here *aggfunc* denotes any aggregate function, *localvars* are declarations of local variables, *condition* restricts the values that the local variables might take, and *term* is the value we are aggregating. Apart from *sum*, other aggregate functions in .QL are *count* (for counting the number of results), *avg* (for computing the average), *min* (for computing the minimum) and *max* (for computing the maximum). Note the pleasing analogy between *from-where-select* and the syntax for aggregates.

2.4. Classes

As we have seen, predicates can be named, and that gives some degree of reusability. However, a more powerful abstraction mechanism is needed, and therefore .QL is object-oriented: you can define your own classes, use overriding, and so on.

We illustrate these features with a coding convention that is specific to Polyglot. Whenever a new AST node class is defined, and it is not a terminal (so it can have children), then it must override the *visitChildren* method. To express that in .QL, we first define a class for AST types:

```
class ASTType extends RefType {
  ASTType() {
    exists (RefType n |
      n.hasQualifiedName("polyglot.ast","Node") and
      this.hasSupertype+(n))
  }
  Field getAChild() {
    result=this.getAField() and
    result.getType() instanceof ASTType
  }
}
```

In words, an *ASTType* is a special kind of *RefType*. The constructor states the defining property of *ASTType*, namely that *polyglot.ast.Node* is a supertype of *this*. Indeed, in general a class in .QL should be thought of as a logical property; that *characteristic predicate* is the conjunction of the property in the constructor and the characteristic predicates of the superclasses. Next, we define a *method* named *getAChild*. Its *result* is a field of *this*, which has a type that is an *ASTType*. More generally, the body of a method is a relation between its parameters, and two special variables named *this* and *result*; void methods which do not return a result are called *predicates*, and do not mention *result*.

All the methods that we used in previous examples were defined in similar class definitions. They are not primitives of the .QL language, and indeed there is nothing in .QL that is specific to the application of querying source code.

We can use the above definition of *ASTType* to find violations of the Polyglot rule that all AST nodes that have a child must implement *visitChildren*:

```
from ASTType t
where not(t.declaresMethod(" visitChildren ")
select t.getPackage(), t, t.getAChild()
```

Interestingly, there are actually quite a few violations in *abcplus*, all outside Polyglot — the designers of Polyglot do as they say, but Polyglot's users make mistakes in using its API. The big advantage of phrasing the coding convention as a query in .QL is that it is easy for Polyglot users to check that they adhere to the rule.

2.4.1. Nondeterminism in Expressions

Methods in .QL should be thought of as nondeterministic mappings. In the above example it is perfectly possible for an ASTType to have multiple children, and then *getAChild()* nondeterministically returns each of them, in an unspecified order.

Concretely, this means that there is no need for the special variable *result* to be assigned a unique value. The following predicate is perfectly well-defined:

```
int twoOrThree() { result=2 or result=3 }
```

In what follows, assume the above predicate is defined in a class named *Int*. When such a nondeterministic method is used in a condition, as below, the result acts like a local variable (in an *exists* clause): *some* result of *i.twoOrThree()* is equal to 3, and so the query succeeds.

```
from Int i
where i.twoOrThree() = 3
select "succeed"
```

By contrast, under a negation the scope of that implicit existential quantifier is *inside* the *not*, so the following query fails:

```
from Int i
where not(i.twoOrThree() = 3)
select "succeed"
```

To get the effect of testing for the existence of a result that is not equal to 3, write:

```
from Int i
where exists(int x | x=i.twoOrThree() and not(x=3))
select "succeed"
```

At first such nondeterminism may appear odd, but in a query language it is a natural consequence of having both expressions (including virtual method calls) as well as disjunction. Furthermore, the use of such nondeterministic mappings avoids the proliferation of named variables that plagues traditional logic programming.

2.4.2. Method Dispatch

Methods can be overridden. To illustrate, let us consider the way query results are displayed in SemmleCode. Every kind of program element defines a method named *getIconPath* that returns the path where the relevant icon can be found. So if we wish to display classes that are *ASTType*s differently from ordinary instances of *Class*, all we need to do is override the *getIconPath* method:

```
class ASTClass extends ASTType,Class {
  string getIconPath() {
    result = "icons/treequery.png"
  }
}
```

Note the use of multiple inheritance: we define a new class that extends not only *ASTType*, but also *Class*. Figure 2 shows the result of querying for the *RefType*s that depend on a particular class in *abc* (see Section 3.1 for a precise definition of this notion); note that some of the results are *ASTType*s, and correspondingly use a different icon.

In general, consider a method call *e.foo*(), where the static type of *e* is *C*. In the last *select* statement of subsection 2.4, the static type of the receiver *t* is *ASTType*.

We wish to determine what method(s) are actually invoked by such a call. The first step is to find the *root definitions* of *foo*() (a *root definition* is a definition of *foo*() in some supertype of *C* that does not itself override another

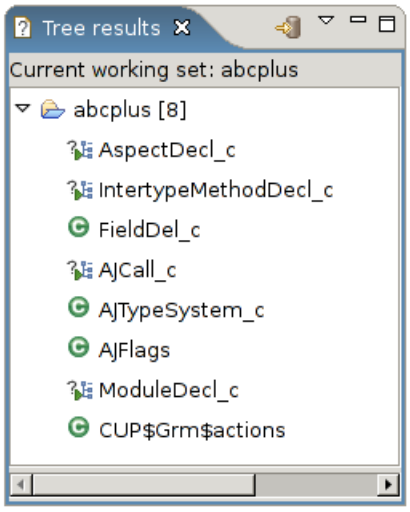

Figure 2. Query results with some *ASTType*s

definition of *foo*()). In the above example, the single root definition of *getIconPath* is in *Element*. Any definition of *foo*() that overrides a root definition (*i.e.* is defined in a subclass of a class with a root definition) is a candidate for execution at the call site *e.foo*().

In the example, that implies all definitions of *getIconPath* anywhere in the library are taken into account. Among these candidates, a method definition in a class *D* is applied if and only if the value of *e* satisfies the defining predicate of *D*, and there is no subtype of *D* for which *e* also satisfies the characteristic predicate. In our example, all classes that are *ASTType*s will be displayed with the tree icon, but other classes are displayed with the normal class icon as before (as we saw in Figure 2).

The process of call resolution thus consists of two steps, one static and one dynamic. In the static step, we find the root definitions and hence all the candidate definitions in subtypes of those roots. For each candidate, we dynamically test whether it is most specific. It may happen that there are two most specific methods applicable: the choice between them is then made nondeterministically (*i.e.* each possibility is tried in turn).

2.5. Generic Queries

The big advantage of object-orientation in .QL turns out to be the ability to create generic queries, distributed as libraries, that can be instantiated quickly and easily to particular code bases.

As an example, consider the *Factory* pattern, which postulates that all elements of a particular kind should be constructed in a special factory class — this allows for greater flexibility. Indeed, it occurs in the Polyglot framework, where all *ASTType*s (*cf.* Section 2.4) must be created

through *ASTNodeFactory* instances, and ignoring this API contract will break the built-in extensibility mechanisms. Like the earlier rule for *visitChildren*, this is a coding convention that is particular to Polyglot.

Let us first define a generic version of the concern: a class to represent some factory, with methods to detect violations of the pattern.

```
class Factory {
  Factory() { this="factory" }
  string toString() {result="factory"}
  predicate factory(RefType t) {any()}
  predicate product(RefType t) {any()}
  ConstructorCall getAViolation() {
        this.product(result.getType()) and
        not(this.factory(result.getCaller().
                         getDeclaringType())) and
        not(result instanceof SuperConstructorCall
            or result instanceof ThisConstructorCall)
  }
}
```

In words, a *Factory* is just a singleton value (arbitrarily chosen to be the string "Factory"), which provides some additional built-in logic. The two predicates *factory*() and *product*() use the special predicate *any*(), which marks them as abstract; subclasses can override them to provide concrete definitions. The idea is that *factory*(t) holds when t is a factory type, and *product*(t) holds if t is a type that should only be produced in the factory.

The meat of the definition is in the *getAViolation* method, which picks out constructor calls that manufacture a product, that occur outside a factory, and that are not *super*- or *this*-constructor calls.

All this work only has to be done once, for the library queries. To actually use it, we simply need to instantiate the generic .QL class to our particular application, *ASTNodeFactory* in Polyglot:

```
class ASTNodeFactory extends Factory {
  predicate factory(RefType t) {
    t.getASupertype+().hasQualifiedName(
          "polyglot.ast", "NodeFactory")
  }

  predicate product(RefType t) {
    t instanceof ASTType
  }
}
```

Note that we use our definition of *ASTType* to pick out things that should be factory products; a factory is just a subtype of the appropriate class in Polyglot.

To find violations, we can now write the following query:

```
from ASTNodeFactory f
select f.getAViolation(), "This constructor call " +
       "should be a factory method call!"
```

Since here we select pairs of program elements and strings, SemmleCode allows us to visualise the results as warnings in the Eclipse IDE: the relevant code locations are marked with the message provided, and can be inspected. It turns out that there are seven violations of the Factory pattern in *abc*, all of which are potential extensibility bugs. Again, Polyglot's designers conform to their own API contracts. Distributing the above .QL snippet with Polyglot would allow end-users to check their own conformance.

3. Metrics

We now illustrate the use of .QL in a particular application area, namely computing software metrics. Often metrics are defined only in an informal style, with many details left unspecified. By contrast, the expression of a metric in .QL can serve as a fully formal specification, which is furthermore executable. The object-oriented features of .QL come in handy in coding up variations that are often found in the literature, and also in tailoring metrics to a particular project or framework.

3.1. Dependency between types

Many metrics use a notion of dependency between types; in turn that notion is used to define dependency between packages, and so on. One type s is *dependent* on another type t if s somehow refers to t. Such a reference can take many different forms. More precisely: s depends on t if

- s is a direct subtype of (implements or extends) t
- s declares a field of type t
- s declares a method or constructor that
  - has t as its return type
  - has a parameter of type t
  - throws an exception of type t
  - calls a method or constructor declared in t
  - accesses a field declared in t

So far, so good: the above list of requirements is easily translated to .QL. However, there are some subtleties related to the use of generic types in Java. The precise details go beyond the scope of this paper; the interested reader is referred to [29], where the full definition of the predicate *usesType*(t_1, t_2) is given. This holds when $t_1 = t_2$ or t_2 is

somehow used to define t_1, *e.g.* as a type parameter to a generic type or the element type of an array type.

With the definition of *usesType* in hand, we can define the *depends* relation itself as follows:

```
predicate depends(RefType s, RefType t) {
    not isParameterized(s) and
    not isRaw(s) and
    (
      usesType(s.getASupertype(), t) or
      usesType(s.getAField().getType(), t) or
        ... )
```

There are in fact five more disjuncts for different ways in which type *t* may occur in type *s*.

3.2. Coupling metrics between types

To measure all the incoming dependencies of a type, we can define a new method on reference types:

```
int getAfferentCoupling() {
    result = count(RefType t | depends(t, this))
}
```

This metric gives some indication of the number of responsibilities of a type, and therefore of the amount of effort that might be involved in changing it. Figure 3 shows a graph of most strongly coupled classes in our working set *abcplus*. Unsurprisingly, the top scorer for this metric is the *Position* class, which records locations in the source and is used very extensively. Other heavily used classes are also on that list.

3.3. Coupling between packages

It is easy to lift the notion of dependency between types to dependency between packages: a package *p* depends on a package *q* precisely when *p* contains some type *s* that depends on a type *t* in *q*. In this case we define a new class named *MetricPackage* especially for metrics on packages. One of its methods returns all the packages (other than itself) that this package depends on:

```
MetricPackage getADependency() {
    exists(RefType t |
     depends(this.getARefType(),t) and
     result = t.getPackage()) and
     result != this
}
```

It is often said that cycles in the package dependence graph are an indication of bad design [17, 24], so let us write a query to identify such cycles. First, we shall restrict ourselves to cycles through source files, leaving libraries out of the equation, since we cannot account for their design. Method *getASrcDep* simply restricts *getADependency* to types that occur in the source:

```
MetricPackage getASrcDep() {
    result = this.getADependency() and
    result.fromSource() and
    this.fromSource()
}
```

Now, to identify members of a cycle, we define the following nondeterministic method. It returns any package that occurs in the same strongly connected component of the dependency graph as *this*:

```
MetricPackage getACycleMember() {
    result.getASrcDep*() = this and
    this.getASrcDep*() = result and
    result.fromSource()
}
```

Here the $*$ operator denotes zero or more applications of the same operation: it is the reflexive transitive closure.

Having defined the notion of dependency cycles, our next job is to decide how such cycles might be reported. Clearly it would be handy to know their size, and that is done by the simple definition

```
int getCycleSize() {
    result = count(this.getACycleMember())
}
```

To report the cycle itself, we need to be able to pick a representative member. In the absence of any other criterion, we just take the minimum in dictionary order:

```
predicate isRepresentative() {
  this.getName() =
    min(MetricPackage p |
          p = this.getACycleMember() |
          p.getName())
}
```

Now we are ready to put it all together. The following query reports all cycles in the source:

```
from MetricPackage p
where p.getCycleSize() > 1 and p.isRepresentative()
select "SCC of size " +
       p.getCycleSize().toString() +
       " for " + p.getName() as s,
       p.getACycleMember()
order by s desc
```

In *abcplus*, there is a large number of such cyclic package dependencies. On close inspection, many of them are quite innocuous, being quite small and confined to one particular

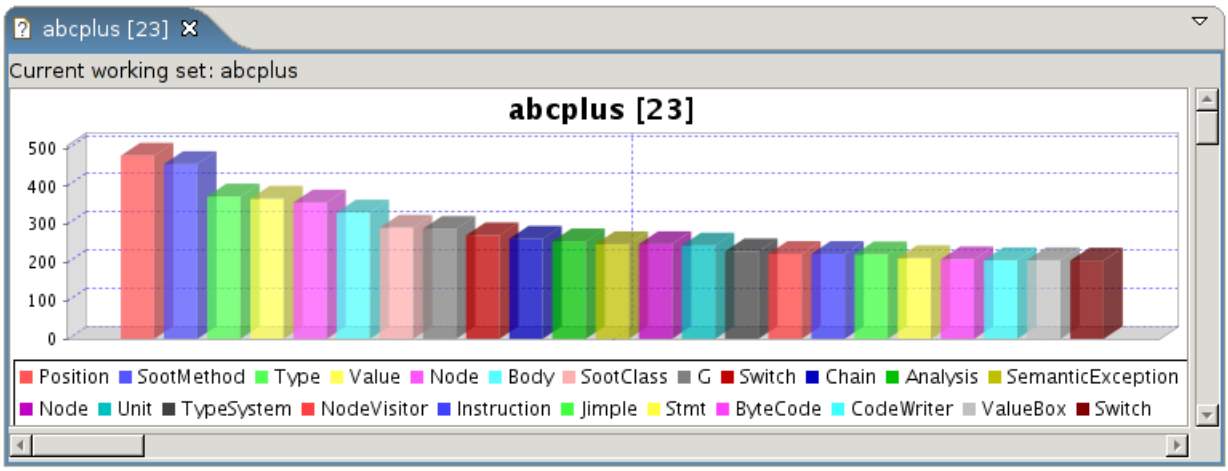

Figure 3. Classes with highest afferent coupling in *abcplus*

project where the different packages are indeed closely related. However, there is a very large cycle in Soot (of size 84), resulting from its use of a root package named *soot* that many other packages cyclically depend on. The query thus correctly draws our attention to that design flaw, and it would be better to refactor Soot to avoid all these cycles that meet at the root package.

3.4. Lack of Cohesion in Methods

We now move on to a different type of metric, namely to measure *lack of cohesion*. The idea is that in a well-designed class, most methods access the same data. While the basic intuition is simple, the precise way to measure this property has been the subject of intense debate. Below we present two different proposals, one due to Shyam Chidamber and Chris Kemerer, and another due to Brian Henderson-Sellers. Interesting evaluations of some of these metrics can be found in [5, 8, 30]. Our aim, however, is not to decide what metric is best: all we wish to do is to demonstrate the power of .QL in experimenting with various definitions.

3.4.1. Chidamber and Kemerer

The Chidamber and Kemerer metric inspects pairs of methods [7]. If there are many pairs that access the same data, then the class is cohesive. On the other hand, if there are many pairs that do not access any common data, then the class is not cohesive. Cohesion is measured as follows:

- n_1 = number of pairs of distinct methods in a reference type that do not have at least one commonly accessed field
- n_2 = number of pairs of distinct methods in a reference type that do have at least one commonly accessed field
- $LCOM = ((n_1 - n_2)/2 \text{ max } 0)$. We divide by 2 because each ordered pair (m_1, m_2) is counted twice in n_1 and n_2.

High values of LCOM indicate a lack of cohesion. Specifically, an LCOM of greater than 500 indicates a potential problem.

We introduce a special subclass of *RefType*, named *MetricRefType*, for recording the relevant definitions in .QL. First, here is a predicate for testing that two methods are distinct members of the same type:

```
predicate distinctMembers(Method m1,
                          Method m2) {
  m1.getDeclaringType() = this and
  m2.getDeclaringType() = this and
  m1 != m2
}
```

Next, our task is to determine whether two methods access a common field:

```
predicate shareField(Method m1, Method m2) {
  exists(Field f |
    m1.accesses(f) and
    m1.getDeclaringType() = this and
    m2.accesses(f) and
    m2.getDeclaringType() = this)
}
```

Finally, we can compute the lack of cohesion metric itself. Its definition closely follows the description in English above. In our view, this is one of the big advantages of .QL:

instead of having to make do with vague descriptions in natural language, a fully formal description can be given in .QL — and that description is executable too. The definition of Chidamber and Kemerer lack of cohesion is:

```
int getLackOfCohesionCK() {
  exists(int n1, int n2, int n |
   n1 = count(Method m1, Method m2 |
              this.distinctMembers(m1,m2) and
              not(this.shareField(m1,m2)))
   and
   n2 = count(Method m1, Method m2 |
              this.distinctMembers(m1,m2) and
              this.shareField(m1,m2))
   and
   n = (n1 - n2)/2
   and
   ((n < 0 and result = 0) or (n >= 0 and result = n))
}
```

To use the above definition, we can use a simple select statement to find all types that have a lack of cohesion greater than 500. In *abcplus*, 75 source types (out of 3974) lack cohesion in this sense. Many of these are adapters for analyses in Soot; it is not surprising that these are lacking cohesion, since the data they access tends to be passed as parameters. An interesting match is *GlobalAspectInfo*, the data structure that stores all aspect-specific information about an AspectJ program in *abc*. Indeed, it has many separate pieces of data, accessed by only a few methods each, so one could argue this rather large class ought to be split up because it represents many different abstractions.

Now one might object to the above definition, because it treats static methods and instance methods on the same footing. Suppose that we are given the above definition (via the class *MetricRefType*); how can we adapt it to exclude static methods? The solution is to simply write a new class definition ourselves:

```
class MyMetricRefType extends MetricRefType {
  predicate distinctMembers(Method m1,
                            Method m2) {
    super.distinctMembers(m1,m2) and
    not(m1.hasModifier("static")) and
    not(m2.hasModifier("static"))
  }
}

from MyMetricRefType t, int loc
where loc = t.getLackOfCohesionCK() and loc > 500
select t, loc order by loc desc
```

This does not change the results for *abcplus*, however.

3.4.2. Henderson-Sellers

The intuition underlying the Henderson-Sellers method of calculating Lack of Cohesion of Methods is that in a cohesive class C, many methods access the same fields of C [13]. Formally, let

$$\begin{aligned} M &= \text{set of methods in class} \\ F &= \text{set of fields in class} \\ r(f) &= \text{number of methods that access field } f \\ ar &= \text{mean of } r(f) \text{ over } f \text{ in } F \end{aligned}$$

We then define LCOM of the class under consideration to be

$$LCOM = (ar - |M|)/(1 - |M|)$$

We follow Lance Walton [33] in restricting M to methods that read some field in the same class, and F to fields that are read by some method in the same class. An LCOM value greater than 0.95 indicates a class that may deserve some further scrutiny.

To code this metric in .QL, the first step is to define the notion of a local field access:

```
predicate accessesLocalField(Method m, Field f) {
  m.accesses(f) and
  m.getDeclaringType() = this and
  f.getDeclaringType() = this
}
```

As we are using Lance Walton's convention, we then define the notion of a method that makes an access, and a field that's being accessed:

```
Method getAccessingMethod() {
  exists(Field f | this.accessesLocalField(result,f))
}
Field getAccessedField() {
  exists(Method m | this.accessesLocalField(m,result))
}
```

It is then straightforward to translate the above semi-formal definition to a .QL method:

```
float getLackOfCohesionHS() {
  exists(int m, float ar |
   m = count(this.getAccessingMethod()) and
   ar = avg(Field f |
            f = this.getAccessedField() |
            count(Method x |
                  this.accessesLocalField(x,f))) and
   m != 1 and
   result = ((ar-m)/(1-m)))
}
```

In *abcplus*, there are 88 classes that lack cohesion in this sense. An interesting question is whether those include many of the classes that were found wanting in cohesion by the Chidamber and Kemerer method. Indeed, there are 15 such classes; and one of them is our old friend *GlobalAspectInfo*, which we already acknowledged might need splitting into multiple smaller classes. Several other matches are also clearly not cohesive, for instance the *Options* class, which is just a collection point for all the options one can pass to *Soot*, without any obvious relation between the fields.

Again there are many variations one might wish to explore of this Henderson-Sellers metric. For instance, it clearly penalises the use of accessor methods, and that is arguably undesirable. It is easy to compensate for that, however, by overriding the *accessesLocalField* predicate:

```
class MyMetricRefType2 extends MetricRefType {
 predicate accessesLocalField (Method m,
                               Field f) {
   super.accessesLocalField (m,f)
   or
   exists (Method n |
         m.getACall(). getCallee () = n and
         n.getDeclaringType() = this and
         super.accessesLocalField (n,f))
  }
}
```

This captures any call to a method in the same class that accesses a local field, not just accessor methods, leading to higher cohesion results. It can be argued, however, that a class that calls many of its own methods *is* more cohesive.

3.5. Specialisation Index

The specialisation index metric measures the extent to which subclasses override (replace) the behaviour of their ancestor classes. If they override many methods, it is an indication that the original abstraction in the superclasses may have been inappropriate. On the whole, subclasses should add behaviour to their superclasses, but not alter that behaviour dramatically.

This metric was proposed by Mark Lorenz and Jeff Kidd [21]. The idea is to weight each overridden method by the depth in the inheritance hierarchy at which it occurs, and then take the average over all methods in the type.

Formally, we compute the number of overridden methods in a class, multiply by the depth in the inheritance hierarchy, and then divide by the total number of callables. It is common (for instance in Frank Sauer's Metrics 1.3.6 [28]) to exclude certain commonly overridden methods from the calculation of the number of overridden methods, for instance *equals*, *toString* and *hashCode*.

A specialisation index of greater than 5 is generally considered suspect and might warrant further investigation.

To code this metric in .QL, we first need to define the inheritance depth of a type, that is the length of the longest path from *Object* to this type in the inheritance hierarchy. In the .QL code, we first compute the length of *some* path to the root of the inheritance hierarchy, and then we take the maximum over all those lengths to obtain the depth:

```
int getADepth() {
   (this.hasQualifiedName("java.lang","Object")
    and result=0)
 or
    (result = ((MetricRefType)this.
                  getASupertype()).
                  getADepth() + 1)
}

int getInheritanceDepth () {
    result = max(this.getADepth())
}
```

Note how the result of a *getASupertype* is cast to *MetricRefType*, so that the *getADepth* method can be called on it. To Java programmers, this may appear strange at first, but remember that .QL classes are characterised by logical properties: casting to a type is just checking that the value satisfies the property of being a *MetricRefType*. In this example, since any *RefType* is also a *MetricRefType*, that check will always succeed.

Next, to compute the specialisation index, we need to specify the number of methods that are being overridden, taking into account any exceptions. First, we define the exceptions as a method (which, in turn, can be overridden by users of the metric to adapt it to their own codebase):

```
predicate ignoreOverride(Method c) {
   c.hasName("equals") or
   c.hasName("hashCode") or
   c.hasName("toString") or
   c.hasName("finalize") or
   c.hasName("clone")
}
```

Next, we find the number of methods that are proper overrides, in that the overridden method is not abstract, and we count their number:

```
Method getOverrides () {
   this.getAMethod() = result and
   exists(Callable c |
            result.overrides(c) and
            not(c.hasModifier("abstract"))) and
   not(this.ignoreOverride(result))
}
```

```
int getNumberOverridden() {
    result = count(this.getOverrides())
}
```

Now we can pin down the definition of the specialisation index itself:

```
float getSpecialisationIndex() {
  this.getNumberOfCallables() != 0
  and
  result = (this.getNumberOverridden()
            *
            this.getInheritanceDepth())
           /
            this.getNumberOfCallables()
}
```

When run on *abcplus*, one of the classes with very high specialisation index is *AdviceDecl_c*, which represents advice declarations in AspectJ. Indeed, that started out as a subclass of *MethodDecl_c*, but then many of its methods were overridden as the design of the *abc* compiler evolved. It is now a candidate for refactoring, taking away that dependency on method declarations, because it doesn't share a lot of behaviour with its superclass.

It is interesting to experiment with variations, especially in terms of the methods that are counted as overrides. Arguably when method *m* overrides *n*, but calls *n* via a super call, it is not altering the behaviour of its superclass, just adding to it. From that point of view, such super-calling methods *m* should be ignored when computing the number of overridden methods. Indeed, in his Metrics 1.3.6 Eclipse plugin [28], Frank Sauer offers precisely such an option. In .QL, we just override the existing definition of the *ignoreOverride* predicate:

```
class MyMetricRefType extends MetricRefType {
  predicate ignoreOverride(Method m) {
    super.ignoreOverride(m) or
    exists(Method n |
      n = m.getACall() and m.overrides(n))
  }
}
```

4. Related work

The idea to use a query language for source code analysis is almost as old as the subject itself. Below we briefly highlight what we regard as the main milestones in the development of the idea. We then compare the design of .QL more generally to other object-oriented query languages in the database literature.

4.1. Code Queries

Mark Linton was the first to propose that a program be stored in a database [19]. That system was named Omega; it did not allow recursive queries. Furthermore, Linton already noted disappointing performance in his initial experiments. Indeed, it appears many in the research community believed that the approach would never be scalable [6], so at best we could store only structural information about the program. As the efficiency of SemmleCode shows, that belief is now wrong, and the reason is two-fold. First, database optimisers have had many advances since the early eighties, and SemmleCode leverages those advances by compiling .QL to SQL. Furthermore, SemmleCode itself applies many optimisations in that compilation process. Yet it is evident we have only scratched the surface in that respect, and a huge number of further optimisations are possible.

In a separate development, many researchers have investigated the use of logic query languages for querying code, starting with the XL C++ Browser [15]. A modern variant of that idea is JQuery [14, 25], which is also nicely integrated with Eclipse. The use of Prolog has some serious drawbacks, however: in particular, termination of queries is hard to predict.

Based on that observation, we previously proposed the use of Datalog as a code query language [12]. Datalog is a very restricted logic programming language, essentially Prolog without data structures. In that earlier paper, we demonstrated excellent performance, but arguably Datalog is too Spartan a notation for writing queries, and it certainly does not lend itself to the creation of libraries of queries, which we regard as crucial.

A more comprehensive account of related work on code queries can be found in [12]. An excellent, in-depth account of many of the important issues in the field can be found in the PhD dissertation of Michael Eichberg [10].

4.2. Object-oriented Query Languages

Naturally we are not alone in observing that Datalog ought to be augmented with facilities for the creation of query libraries before it can be applied on an industrial scale. The difficulties were, however, well summarised in Jeff Ullman's landmark paper [31], where he stated:

> *It is not possible for a query language to be seriously logical and seriously object-oriented at the same time.*

History appears to have confirmed this statement. While a number of theoretical papers were written on the subject, the combination of object-orientation and Datalog never made it to the mainstream. As we claim .QL to be a truly object-oriented language with a proper logical foundation,

we need to examine Ullman's arguments, and understand how they are side-stepped in the design of .QL.

The first argument put forward by Ullman concerns object identity: in his view of object-orientation, it is an essential feature that each object is given a unique identity upon creation. In particular, if a tuple is generated in multiple ways, it gets a new identity for each creation. Ullman then convincingly demonstrates this idea does not combine well with the least-fixpoint semantics of Datalog. The object identity problem does not arise in .QL, because there is no object identity: classes are just logical properties.

Ullman's second argument concerns the idea that each object has precisely one type. Clearly in a database setting this does not make sense, as it is likely that we need many overlapping types. Again, such overlaps occur naturally in .QL, because classes are predicates, and multiple predicates can be true of the same object. Furthermore, when writing little queries, it's essential that new types can be easily introduced on the fly, and in .QL you can indeed write little classes as part of a query.

Ullman's final argument (actually phrased as an open problem) is to point out that overlapping types can potentially lead to huge performance problems. This has been solved in the .QL implementation via a number of proprietary optimisations. A full exposition of these optimisations is beyond the scope of the present paper.

It appears that much of the literature after Ullman's paper has concentrated on overcoming his first objection: object identity plays a dominant role in the research on object-oriented deductive databases after 1991. We regard that as a mistake, as the convenience of writing simple declarative queries is the *raison d'être* for a query language; it is not object-identity that is essential, but the ability to build libraries of reusable queries. Indeed, .QL is partly the result of attempting to follow the path mapped out by Ullman, but focussing on the open problem he identified rather than attempting to retrofit object-identity on Datalog.

One notable exception is a paper by Serge Abiteboul *et al.* [2], which proposes (as an afterthought to the main body of the paper) a notion of *virtual class* that is quite close to the normal classes of .QL. However, the precise definition of virtual method dispatch is quite different there: instead of considering all candidates below the root definitions, it takes all candidates below the 'closest match'. Also multiple inheritance is not permitted; in .QL, multiple inheritance just means conjunction of characteristic predicates. Furthermore, the authors remark, again, that novel optimisations are needed to make the approach feasible, but none are offered.

Abiteboul went on to build on these ideas in his design of IQL(2) [1]. However, there the inheritance hierarchy is tied to a notion of record subtyping; again this is far more complex than the choice taken in .QL, where inheritance is simply logical implication. The semantics of overriding are left implicit in [1]. An interesting feature of IQL(2), which is not present in .QL, is the use of parameterised classes, for instance to distinguish between locations: the class *Friends*(*LA*) defines a *phone* attribute, whereas *Friends*(*Paris*) has a *téléphone* attribute. In .QL, there would have to be a single class *Friends* with a subclass for each location.

Other works that also build on Abiteboul *et al.*'s 1991 paper, however, ignore the notion of virtual classes, instead again focusing on object identity, and assigning a single type to each object, for instance [3]. In such complex semantics multiple inheritance is difficult to account for, while in .QL it is very simple indeed.

In more recent proposals to arrive at a synergy between objects and deductive databases, like Mengchi Liu's ROL language [20], the simplicity of Datalog is sacrificed by bringing in complex terms, and even set-valued results, thus foregoing the advantages of the very simple fixpoint semantics of Datalog (which in turn enable many optimisations).

5. Conclusions

We have presented .QL, an object-oriented query language, and demonstrated its suitability for source code analysis, in particular focusing on metrics. .QL is unique in its simple object model, where classes are just logical properties and inheritance is implication. Furthermore, its notation for aggregate computations, which we borrowed from the program derivation community, provides a simple way of expressing operations that would be awkward in SQL.

There are many other potential application areas to be considered. For now, SemmleCode does not store control flow information, and therefore it is not yet possible to express analyses like those proposed by Reps [27] and Whaley [18, 34]. It would be interesting to see whether such typical program analysis applications also benefit by the object-oriented nature of .QL. We conjecture that there too, it will be beneficial to define generic analyses that are subsequently specialised by subclassing.

Another application area concerns the discovery of crosscutting concerns, as advocated by Marin, Van Deursen and Moonen in [22]. Indeed, the same authors have recently proposed a query-based tool for such tasks named SoQueT [23], and it would be interesting to examine whether .QL can be used for the same purpose.

Finally, we would like to stress once again that nothing in the design of .QL is specific to the application of source code analysis. Given an annotated database schema, one can build a library for common tasks just like the one sketched here for source code analysis. The details of the database schema used for querying Java in Eclipse can be found on our website [29].

References

[1] S. Abiteboul and C. S. dos Santos. IQL(2): A model with ubiquitous objects. In P. Atzeni and V. Tannen, editors, *Database Programming Languages (DBPL-5)*, Electronic Workshops in Computing. Springer, 1995.

[2] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proceedings of SIGMOD 1993*, pages 32–41, 1993.

[3] F. N. Afrati. On inheritance in object oriented datalog. In *International Workshop on Issues and Applications of Database Technology (IADT)*, pages 280–289, 1998.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible AspectJ compiler. In *Proceedings of AOSD*, pages 87–98. ACM Press, 2005.

[5] V. Basili, L. Brand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–760, 1996.

[6] Y. Chen, M. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[8] D. P. Darcy, S. A. Slaughter, C. F. Kemerer, and J. E. Tomayko. The structural complexity of software: an experimental test. *IEEE Transactions on Software Engineering*, 31(11):982–995, 2005.

[9] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1990.

[10] M. Eichberg. *Open Integrated Development and Analysis Environments*. PhD thesis, Technische Universität Darmstadt, 2007. `http://elib.tu-darmstadt.de/diss/000808/`.

[11] H. Gallaire and J. Minker. *Logic and Databases*. Plenum Press, New York, 1978.

[12] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: scalable source code queries with Datalog. In D. Thomas, editor, *Proceedings of ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006.

[13] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[14] D. Janzen and K. de Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003.

[15] S. Javey, K. Mitsui, H. Nakamura, T. Ohira, K. Yasuda, K. Kuse, T. Kamimura, and R. 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.

[16] A. Kaldewaij. *The Derivation of Algorithms*. Prentice Hall, 1990.

[17] J. Lakos. *Large-Scale C++ Software Design*. Addison Wesley, 1996.

[18] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of PODS*, pages 1–12. ACM Press, 2005.

[19] M. A. Linton. Implementing relational views of programs. In P. B. Henderson, editor, *Software Development Environments (SDE)*, pages 132–140, 1984.

[20] M. Liu, G. Dobbie, and T. W. Ling. A logical foundation for deductive object-oriented databases. *ACM Transactions on Database Systems*, 27(1):117–151, 2002.

[21] M. Lorenz and J. Kidd. *Object-oriented Software Metrics*. Prentice Hall, 1994.

[22] M. Marin, A. Van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, page To appear, 2007.

[23] M. Marin, A. Van Deursen, and L. Moonen. SoQueT: Query-based documentation of crosscutting concerns. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 758–761, 2007.

[24] R. C. Martin. *Agile Software Development, Principles, Patterns and Practices*. Prentice Hall, 2002.

[25] E. McCormick and K. D. Volder. JQuery: finding your way through tangled code. In *Companion to OOPSLA*, pages 9–10. ACM Press, 2004.

[26] N. Nystrom, M. R. Clarkson, and A. 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.

[27] T. W. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, volume 296 of *International Series in Engineering and Computer Science*, pages 163–196. Kluwer, 1995.

[28] F. Sauer. Eclipse metrics 1.3.6. `http://metrics.sourceforge.net`, 2006.

[29] Semmle Ltd. Company website with free downloads, documentation, and discussion forums. `http://semmle.com`, 2007.

[30] D. D. Spinellis. *Code Quality: the Open Source Perspective*. Addison-Wesley, 2007.

[31] J. 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.

[32] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[33] L. Walton. Eclipse metrics plugin — State of Flow. `http://eclipse-metrics.sourceforge.net/`, 2006.

[34] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog and binary decision diagrams for program analysis. In K. Yi, editor, *Proceedings of APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.