Lund University
Computer Science Department

QL: Object-oriented Queries on Relational Data

SDE Reading Group

Idriss Riouak

March 1, 2022

What is QL ?

- QL is:
  - A **logic** language based on first-order logic
  - A **declarative** language without side-effects
  - An **Object-oriented** language
  - A **query** language working on a relational data models.
- General purpose language ... well suited for implementing static analyses.
- Developed by **Semmle** and bought by GitHub in 2019.
- Now is the core of **CodeQL**

Snapshot database

Query are executed on a special database called **snapshot database**.

- The database contains a representation of the program to analyse
- Describes the program as it was at one particular point in time.

The result of a query is as set of tuples.

Example

Goal: find useless expressions, i.e., pure expressions in a void context, in JS.

```
import javascript // Provides general support for working with JS

predicate inVoidContext(Expr e) {
exists (ExprStmt s | e = s.getExpr()) or
exists (SeqExpr seq, int i | e = seq.getOperand(i) and
(i < count(Expr op | op = seq.getOperand(_))-1 or inVoidContext(seq)))
}

from Expr e
where e.isPure() and inVoidContext(e) and not (e instanceof VoidExpr)
select e, "This expression has no effect."
```

QL overview

A QL program is composed by:

- a set of **intensional** predicates, e.g., `inVoidContext(·)`
  - one of which is a distinguished query predicate **from_where_select**.
- Evaluated on top of an **extensional** database which defines a set of extensional predicates

The target language of QL is a dialect of ***Datalog***. This dialect provides support for arithmetic and string operations.

QL overview

The semantics of a program is the **lfp** of its intensional predicates.

Intensional predicates are assigned the smallest sets of tuples that satisfy their recursive definitions.

Types

- A type in QL represents a set of values. This set is called **Extent**
- Two kinds of base type:
  - **Primitive types**: e.g., int or string. Fixed extent
  - **Entity types**: defined by a unary extensional predicate. Context-dependent extent.
    - Extent of **@expr** in JS: set of all expression in the program
    - Extent of **@seqexpr** in JS: set of sequence expressions in the program
- Classes are types whose extent is defined by the **characteristic predicate** of the class.

```
class Digit extends int { Digit() { (int)this in [0..9] } }
```

LTH
FACULTY OF ENGINEERING

Subtyping

- Subtyping can be viewed as **set inclusion** of extents.
  - If **A** is a subtype of **B**, then the extent of **A** is a subset of the extent of **B**.
- For entity types, the subtyping relation is given by the database schema:
  - `@seqexpr` $<:$ `@expr`
  - Entity types can only be subtypes of other entity types: `@NullLiteral` $\not<:$ `string`
- For classes, direct supertypes are specified as part of their declaration (using java-like syntax).

```
class Even extends Digit { Even() { (int)this % 2 = 0 } }
class Odd extends Digit { Odd() { not this instanceof Even } }
class PrimeDigit extends Digit {
PrimeDigit() { count(Digit divisor | (int)this % (int)divisor = 0)
    = 2 } }
```

LTH
FACULTY OF ENGINEERING

Multiple supertypes

- A class can have multiple supertypes
- The intersection of all the extent of the supertypes is called **domain**.
- The domain of a class is not always equal its extent.
- Example

```
class EvenPrime extends Even, PrimeDigit {}
```

- In this case the domain is equals to the class extent.
- `EvenPrime` is a subtype of the intersection between `Even` and `PrimeDigit`.

Prescriptive vs Descriptive typing

- QL follows a **prescriptive** typing discipline: the syntactic type declaration corresponds to a semantic containment check at runtime.

```
predicate isSmall(Digit d) { (int)d < 5 }

from int i where isSmall(i) and i < 0 select i
```

- Under a **descriptive** typing discipline, this would be compile-time error.
- The predicate is syntactic sugar for

```
predicate isSmall(int d) { d instanceof Digit and d < 5 }
```

LTH
FACULTY OF ENGINEERING

Member predicates

The predicate isSmall describes a property of Digits , so it makes sense to add it to class Digit as a *member predicate*.

```
class Digit extends int {
  Digit() { (int)this in [0..9] }
  predicate isSmall() { (int)this < 5 }
  predicate divides(Digit that) { (int)that % (int)this = 0 } }
}

from Digit d where d.isSmall() select d
```

LTH
FACULTY OF ENGINEERING

Multi-valued expressions

- QL allows treating predicates as multi-valued "functions" with a dedicated `result` parameter.

```
Digit getADivisor() { (int)this % (int)result = 0 }
```

That can be used:

```
from Digit d where d.getADivisor() = 2 select d // selects 0, 2,
    4, 6, 8
```

- When translated to Datalog, the predicate si desugared in a normal predicate by making the result parameter explicit. For instance `d.getADivisor()=2` is translated into:

```
exists (Digit tmp | d.getADivisor(tmp) and tmp = 2)
```

LTH
FACULTY OF ENGINEERING

Abtract classes

- Top-down modelling: starting from a general superclass representing a large set of values, we carve out individual subclasses representing more restricted sets of values.
- Bottom-up modelling: think about a class as being the union of its subclasses. QL supports this using the notion of abstract classes.
  - An abstract class can have one or more superclasses
  - And a characteristic predicate
  - But the extent of an abstract class is the union of the extends of all its subclasses.

Storage level

QL program are run on a relational database.

The abstract syntax tree is encoded in tables:

ID	Kind
2	EqExpr
1	VarRef
0	IntegerLiteral

Expr: (x === 1)

Id - PK	Kind - FK	Parent - FK	Idx
0 //(x === 1)	2	…	…
2 // 1	0	0	1
1 // x	1	0	0

LTH FACULTY OF ENGINEERING

Data Abstraction

- QL classes hide the specifics of how data is stored in tables behind a higher-level interface, thereby acting like abstract datatypes.

```
class Expr extends @expr {
  Expr getParent() { exprs(this, _, result, _ ) }
  Expr getChildExpr(int i) { exprs(result, _, this, i) }
  string toString() { result = "expr" }
}
```

- Easier to change data representation if all client analyses use Expr instead of directly accessing the DB.

Inheritance

We can have a richer semantic interface by defining subclasses of Expr:

```
class EqExpr extends Expr {
  EqExpr() { exprs(this, 2, _, _) } // Characteristic predicated
  Expr getLeftOperand() { result = this.getChildExpr(0) }
  Expr getRightOperand() { result = this.getChildExpr(1) }
  string toString() { result = "===" }
}
```

Overriding

As a practical example of overriding, consider implementing **Expr.isPure**:

```
class Expr extends @expr { predicate isPure() { none() } //Built-in
    predicate that always fails

class Literal extends Expr { predicate isPure() { any() } //Built-in
    predicate that always succeeds

class EqExpr extends Expr {
predicate isPure() { forall (Expr c | c = this.getChildExpr(_) |
    c.isPure()) //Propagating the check to all the children
 }
```

LTH FACULTY OF ENGINEERING

Interface vs Implementation

We want to implement an analysis for JS to find comparisons between expressions with incompatible (dynamic) types, which will always evaluate to false at runtime

```
from EqExpr eq, Expr l, Expr r
where l = eq.getLeftOperand() and r = eq.getRightOperand() and
    incompatTypes(l, r)
select eq, "Operands have incompatible types."
```

LTH
FACULTY OF ENGINEERING

Interface vs Implementation

We want to implement an analysis for JS to find comparisons between expressions with incompatible (dynamic) types, which will always evaluate to false at runtime

```
from EqExpr eq, Expr l, Expr r, AnoterhKindOfExpr akoe ...
where l = eq.getLeftOperand() and r = eq.getRightOperand() and
    incompatTypes(l, r) or akoe ...
select eq, "Operands have incompatible types."
```

LTH FACULTY OF ENGINEERING

Interface vs Implementation

- Let's define an abstract class

```
abstract class EqualityTest extends ASTNode {
abstract Expr getALeftOperand();
abstract Expr getARightOperand();
 }
```

- Let's define two new classes:

```
class EqExprEqualityTest extends EqExpr, EqualityTest {
Expr getALeftOperand() { result = this.getLeftOperand() }
Expr getARightOperand() { result = this.getRightOperand() }
}

class SwitchEqualityTest extends SwitchStmt, EqualityTest {
Expr getALeftOperand() { result = this.getExpr() }
Expr getARightOperand() { result = this.getACase().getExpr() }
 }
```

Interface vs Implementation

Now we can rewrite the query in terms of `EqualityTest`

```
from EqualityTest eq, Expr l, Expr r
where l = eq.getALeftOperand() and r = eq.getARightOperand() and
    incompatTypes(l, r)
select eq, "Operands have incompatible types."
```

LTH
FACULTY OF ENGINEERING

A study case

- Reimplemented **ErrorProne** in QL: 101 checks
- One man-month of effort by experienced QL programmer
- ErrorProne LOC: 10500 - 1100 (suggested fixes) - 2800 (import, packages and Override) = 6600
- QL LOC: 2000 - 100 (imports) = 1900
- Java implementation is 3.5x the size of the QL implementation
- QL is 4 time slower than ErrorProne (Warm-up or steady state ?)
- QL runs offline

Conclusions

QL is a lot of things and support many things:

- Data abstraction
- Inheritance with dynamic dispatch
- Overlapping classes
- Relational member predicates
- Object creation and mutation are not supported.
- Parallelism comes for free
- Conciseness
- March 2016: Semmle's static analysis platform offers about 2500 individual analyses for 8 languages.

LTH
FACULTY OF ENGINEERING