QL: Object-oriented Queries on Relational Data

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, Max Schäfer[1]

1 **Semmle Ltd** `publications@semmle.com`

Abstract

This paper describes QL, a language for querying complex, potentially recursive data structures. QL compiles to Datalog and runs on a standard relational database, yet it provides familiar-looking object-oriented features such as classes and methods, reinterpreted in logical terms: classes are logical properties describing sets of values, subclassing is implication, and virtual calls are dispatched dynamically by considering the most specific classes containing the receiver. Furthermore, types in QL are prescriptive and actively influence program evaluation rather than just describing it. In combination, these features enable the development of concise queries based on reusable libraries, which are written in a purely declarative style, yet can be efficiently executed even on very large data sets. In particular, we have used QL to implement static analyses for various programming languages, which scale to millions of lines of code.

1 Introduction

QL is a declarative, object-oriented logic programming language for querying complex, potentially recursive data structures encoded in a relational data model. It is a general-purpose query language, but its strong support for recursion and aggregates makes it particularly well suited for implementing static analyses, code queries and software metrics. Although this paper is not about static analysis *per se*, it is in this area that QL, being the technical basis of Semmle's engineering analytics platform, has seen most use so far, so we will use it as our main source of motivating examples.

A static analysis implemented in QL is simply a query run on a special database: the database contains a representation of the program to analyse (encoding, say, its abstract syntax tree or control flow graph), from which the query computes a set of result tuples. A bug finding analysis, for instance, could return pairs of source locations and error messages. Since the database describes the program as it was at one particular point in time, we refer to it as a *snapshot database*. A snapshot database is created by a language-specific *extractor*. We have built extractors for various different languages based on existing compiler frontends.

As our first example of a QL query, let us consider an analysis for finding useless expressions in JavaScript programs, i.e., pure (that is, side effect-free) expressions appearing in a void context where their value is immediately discarded. Typically, this indicates a typo, for instance mistyping an assignment "`x = 42;`" as an equality check "`x == 42;`".

To identify such expressions we need to implement a purity analysis and a check to determine whether an expression appears in a void context. Fortunately, the former is already implemented in our standard QL library for JavaScript, so we can concentrate on the latter.

A simple query for finding useless expressions is shown in Listing 1. At a very high level, it breaks down into three sections:

- An `import` statement pulls in the existing QL library `javascript`, which, as its name suggests, provides general support for working with JavaScript snapshot databases.
- A predicate `inVoidContext` is defined to identify expressions in void context.
- The main `from-where-select` clause defines the analysis itself:
  - the `from` part declares a variable `e` ranging over all expressions in the analysed program;


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Listing 1 QL query for finding useless expressions in JavaScript

```
import javascript

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."
```

- the `where` part imposes three conditions on `e`: it must be pure, appear in a void context, and not be a `void` expression, which explicitly discards the value of its operand;
- the `select` part specifies the results to return for values of the `from` variables that pass the `where` conditions; in this case, `e` itself is returned with an explanatory message.

Taking a closer look at the definition of `inVoidContext`, it is declared as a unary predicate with a single parameter `e` of type `Expr`. `Expr` and its subtypes model JavaScript expression ASTs: for instance, `BinaryExpr` is a subclass of `Expr` representing all binary expressions, which in turn has a subclass `AddExpr` representing additions; another subclass of `Expr` is `SeqExpr`, representing sequence (or "comma") expressions with two or more operands.[1]

The body of the predicate is a first-order formula with two disjuncts. Its first disjunct says that `e` is in void context if it is the toplevel expression in an expression statement (as in our example above). The second disjunct handles the case where `e` is an operand of a sequence expression: `e` is in void context if it is not the last operand, or if the entire sequence is in void context, as determined by a recursive call to `inVoidContext`.

Judging from this example, QL looks like a domain-specific language for querying JavaScript ASTs, but this not the case: the classes used in this example and the navigation operations available on them are defined entirely in QL, not built into the language. In fact, there is nothing about QL that is specific to dealing with ASTs, or even for writing static analyses, but its object-oriented features allow the development of reusable domain-specific libraries (such as the `javascript` library and its cousins for other languages), providing a rich and convenient API for query writers.

Perhaps more surprisingly, there are not even any objects in the traditional sense of structured records with fields and methods. QL programs only work with atomic values; structured data is encoded as relational tables. For example, it is natural at first to think of the formula `e = seq.getOperand(i)` as an operation on an object `seq`, perhaps involving reading the i-th element of one of its fields holding an array of references to other objects, and then storing the result in variable `e`. In fact, though, all three variables `e`, `seq` and `i` range over atomic values: the latter is an integer, and the former two are *entity values*, that is, opaque identifiers representing entities modelled in the database (in this case, expressions).

In JavaScript snapshot databases, the expression AST structure of the program is encoded in a relation `exprs` containing 4-tuples (c, k, p, i), for entity values c and p and integers k and i, stating that expression c is the the i-th child of p in the AST, and has kind k. Class `Expr`

[1] We use the popular ESTree encoding for JavaScript ASTs format (`https://github.com/estree/estree`), which coalesces nested sequence expressions into a single n-ary expression.

and its subclasses define an object-oriented view of this relation; for example, `getOperand` is defined such that `e = seq.getOperand(i)` is compiled to $\exists k.$`exprs`$(\texttt{e}, k, \texttt{seq}, \texttt{i})$: no field reads, no assignments, just logic.

In particular, `e` is not an output computed from inputs `seq` and `i`: all three variables are on an equal footing, and there is not even any requirement that `e` is functionally determined by `seq` and `i` (though in this particular case it is). This becomes obvious in our use of the **count** aggregate to determine the number of operands to `seq`: `op = seq.getOperand(_)` holds for any value of `op` that is an operand of `seq` (where "`_`" is the special don't-care variable familiar from other logic languages), so `seq.getOperand(_)` behaves like a multi-valued expression. We use the aggregate to count how many of those values there are to obtain the number of operands of `seq`.

In spite of their unusual semantic underpinnings, QL classes offer very similar features to their more traditional counterparts. In particular, classes can have member predicates, such as the `isPure` predicate on `Expr`, which is defined in the standard QL library for JavaScript and overridden with different implementations for various subclasses of `Expr`. Calls such as `e.isPure()` are dispatched virtually, looking up the most specific applicable definitions of `isPure` based on the (runtime) value of `e`.

Like in Java and many other languages, all variables in QL have statically declared types, offering the usual benefits of enabling smart IDEs.[2] However, type declarations in QL are not just assertions to be checked by the compiler, but do, in fact, affect program semantics at runtime: the values that a variable can take during execution are restricted to those that conform to the declared type. In particular, the declared types of predicate parameters and quantified variables restrict the set of values they may range over.

As mentioned above, our example query makes use of (a small part of) the standard QL libraries for JavaScript. Just like the query, the libraries are implemented in an entirely declarative style, specifying *what* should be computed rather than *how*. In fact, QL exposes no details at all of the underlying database system on which the queries are run, and it is up to the optimiser to translate the high-level QL code into an efficiently executable query plan.

In this paper, we present the core features of QL:

- We explain the semantics of classes, member predicates and virtual dispatch, first informally (Section 2) and then more formally via a translation from a subset of the language, Core QL, to plain Datalog (Section 3).
- We discuss practical usage of QL in Section 4, and report on a case study in using QL to implement static checks for Java in Section 5.
- We put QL into context in Section 6, exploring in detail how far it matches the principles of object orientation laid down in the literature, and briefly survey related work.

2 Overview of QL

The fundamental semantic model of QL is that of Datalog: programs define a set of *intensional predicates*, one of which is a distinguished *query predicate*. They are evaluated on top of an *extensional database* (EDB), which defines a set of *extensional predicates*. While intensional predicates are defined by formulas of first-order logic (possibly involving recursion between predicates), extensional predicates are defined as explicit sets of tuples stored in the database.

[2] In fact, this is the main motivation for choosing the `from-where-select` query syntax instead of SQL's `select-from-where`: variables are declared upfront, so code completion is available in the `select` part.

Unlike Prolog, Datalog does not allow the use of complex terms, so intensional predicates can only refer to values already contained in the database and cannot build up new data structures, such as lists. Like many Datalog dialects, QL somewhat relaxes this restriction by providing support for arithmetic and string operations.

The semantics of a program is the least fixpoint of its intensional predicates, that is, intensional predicates are assigned the smallest sets of tuples that satisfy their recursive definitions. Since such a fixpoint need not exist in general, QL imposes the restriction that (mutual) recursion is only allowed under an even number of negations, which is a variant of the *stratified negation* restriction used in many Datalog systems [32]. Once a fixpoint solution has been found, the set of tuples assigned to the query predicate is returned as the overall result of the program.

The grounding of QL's semantics in Datalog is not just an expository device: as explained in Section 4, our implementation compiles QL to plain Datalog, and we shall provide a precise semantics for a core calculus of QL in the next section by formalising the essential parts of that translation.

2.1 Classes

A type in QL represents a set of values, which we will call the *extent* of the type. Classes are types whose extent is defined by a unary intensional predicate called the *characteristic predicate* (or *character* for short) of the class.

There are also two kinds of *base types*, that is, types which are not themselves defined in QL: *primitive types* such as `int` or `string` are built into the language; *entity types* are defined by unary extensional predicates, whose names by convention start with an "@" character. Primitive types always have the same extent, regardless of the content of the EDB, while the extent of entity types and classes may depend on the EDB. For example, snapshot databases representing JavaScript programs defines entity types `@expr` and `@seqexpr` whose extent is, respectively, the set of all expressions and the set of sequence expressions in the represented program.

Subtyping can be thought of as set inclusion of extents: if A is a subtype of B, then the extent of A is a (not necessarily proper) subset of the extent of B.[3] For entity types the subtyping relation is given by the database schema: for instance, the schema for JavaScript snapshot databases declares `@seqexpr` to be a subtype of `@expr`, and it is up to the database system to ensure that this constraint is met at runtime. For classes, direct supertypes are specified as part of their declaration using a Java-like `extends` clause.

While entity types can only be subtypes of other entity types, classes can also extend base types. For instance, we can define a class `Digit` with the extent $\{0, 1, 2, \ldots, 9\}$:

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

The `extends` clause makes `Digit` a subtype of the built-in `int` type, and the character (which syntactically looks like a constructor in Java) further restricts the extent of `Digit`. The `x in [a..b]` notation is a convenience for defining ranges (note that an explicit cast is necessary when using variables with a class type in numeric operations).

Characteristic predicates can contain arbitrary QL code. For instance, we can define the class of even digits and the class of prime digits by subclassing `Digit` and performing arithmetic checks on `this`.

[3] The reverse direction cannot, in general, hold: since characters are arbitrary predicates, inclusion of extents is undecidable, while our subtyping relation needs to be kept decidable.

```
class Even extends Digit { Even() { (int)this % 2 = 0 } }

class PrimeDigit extends Digit {
  PrimeDigit() { count(Digit divisor | (int)this % (int)divisor = 0) = 2 } }
```

Observe that the extents of the two classes overlap, yet neither is a subset of the other. This is a natural consequence of defining types by arbitrary characteristic predicates, but it means that not every value has a unique tightest type.

Like Java, QL has an `instanceof` operator, which in QL is really just syntactic sugar for calling the character of a class. For instance, the class of odd digits can be defined like this:

```
class Odd extends Digit { Odd() { not this instanceof Even } }
```

Being intensional predicates, characters can be recursive. For instance, we could define:

```
class Even extends Digit { Even() { this = 0 or (int)this-1 instanceof Odd } }
class Odd  extends Digit { Odd() { (int)this-1 instanceof Even } }
```

However, recursion has to be stratified, so the following is not acceptable, since there is no unique least fixpoint solution to the predicate definitions:

```
  Even() { not this instanceof Odd } // illegal recursion through not
  Odd() { not this instanceof Even } // illegal recursion through not
```

A class may extend multiple supertypes, which simply means that it is a subtype of the their intersection. The (potentially trivial) intersection of all supertypes of a class is called the *domain* of the class. For instance, the class of even prime digits ($\{2\}$) is defined as

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

In fact, since `EvenPrime` imposes no additional constraints on `this` in its character, its extent is exactly equal to its domain. In general, the extent of a class consists of all those values in its domain that satisfy the body of the character; hence, the implicit `this` variable in the character ranges over the domain of the class.

It should be emphasised that the constructor-like syntax for characters is purely superficial: QL has no `new` expression. Like plain Datalog, QL programs can never construct new values or objects, they can only work with primitive values and the values present in the EDB.

2.2 Prescriptive typing

Every variable in QL has a declared type. In most statically typed imperative and functional languages, such declarations are purely compile-time artefacts that describe the set of values the variable is allowed to take on at runtime; they are checked for consistency by the compiler but play no role at runtime. In contrast to this *descriptive* typing discipline, QL follows a *prescriptive* model, where the syntactic type declaration corresponds to a semantic containment check at runtime.

For instance, consider the predicate `isSmall` that holds for all `Digit`s smaller than five:

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

We can use it in a query like the following (which will return the numbers $0, \ldots, 4$):

```
from int i where isSmall(i) select i
```

Note that `i` is declared to be of type `int`, but is passed as an argument to `isSmall`, whose parameter is declared to be a `Digit`. Under a descriptive typing discipline, this would

be a compile-time type error, but not so in QL: declaring `d` to be a `Digit` simply means that in order for a value to satisfy the predicate `isSmall`, it has to *both* be a `Digit` *and* satisfy the logical conditions imposed by the body of the predicate (namely, being smaller than five).

Another way of looking at it is that type declarations entail an implicit `instanceof` test (which is, in fact, made explicit when translating to plain Datalog), and our definition of `isSmall` is equivalent to

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

The call `isSmall(i)` thus has a perfectly well-defined semantics, regardless of the declared type of `i`, and regardless of what set of values `i` ranges over at runtime. If none of these values happen to be in `Digit`, then `isSmall(i)` will evaluate to an empty set of tuples, as in the following query:

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

In practice, a query or formula that never returns any values usually indicates a mistake. The problem of finding such empty formulas can be reduced to the problem of inferring types for the generated Datalog [33]; (Datalog) formulas for which we infer the empty type are mapped back to the (QL) formulas they arise from, and reported. In general, emptiness of Datalog formulas is undecidable (even without arithmetic or string operations), so we can never find all empty formulas, but in practice this approach has proved to be quite effective.

Besides type declarations, `instanceof` tests and casts also restrict the possible values of variables and expressions: `x instanceof A` restricts `x` to only take on values from `A`, and similarly `(A)x` is an expression picking out those values of `x` that are in `A`.

2.3 Member predicates

The predicate `isSmall` really describes a property of `Digit`s, so it thus 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 } }
```

Like characteristic predicates, member predicates have an implicit parameter `this`, and they are invoked using a method call-like syntax:

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

Of course, member predicates can have other parameters besides `this`. For instance, we could add a predicate `divides` to check whether one digit is a divisor of another:

```
class Digit extends int {
  ...
  predicate divides(Digit that) { (int)that % (int)this = 0 } }
```

There is one important difference between characters and member predicates: in the former, `this` ranges over the domain of the class (that is, the intersection of the extents of its supertypes), while in the latter `this` ranges over the extent of the class itself. This is because the character is what defines the extent of the class in the first place, so by restricting `this` to range over the extent of the class in the character, we would introduce a direct recursive call from the character to itself, which under least fixpoint semantics would mean that the character (and hence the extent of the class) is always empty.

2.4 Multi-valued expressions

Taking the analogy between member predicates and methods further, QL allows treating predicates as multi-valued "functions" with a dedicated `result` parameter. For instance, the member predicate `divides` could instead be written as a multi-valued function returning any of the divisors of a digit:

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

Note that member predicates using the function syntax have an implicitly declared `result` variable whose type is the declared result type. The results of the predicate are precisely those values that the `result` variable is bound to. Syntactically, calls to predicates in function syntax are treated like function calls; in particular, they can be chained as in `d.getADivisor().getADivisor()`, which evaluates to all divisors of divisors of `d`.

Semantically, however, such predicates are still relations: there is no requirement that `result` has precisely one value for each value of `this`, or that `result` is "computed from" `this` in some operational sense. In fact, it is quite possible to use `getADivisor()` in reverse to compute all values of `this` yielding a given `result` value, as shown in the following query:

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

When translating to Datalog, predicates using the function syntax are desugared into normal predicates by making the `result` parameter explicit and introducing temporary variables as necessary. For instance, `d.getADivisor()=2` is translated into code of this form:

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

Thus, multi-valued expressions are a purely syntactic, if practically very useful, feature.

2.5 Overriding and virtual dispatch

Given that we have classes that contain member predicates and that may extend each other, it is natural to ask whether there is a notion of overriding and virtual dispatch, and indeed there is: intuitively, at runtime a call `x.p(...)` is dispatched to the definition of `p` belonging to the tightest class containing `x`, i.e., the most specific applicable definition of `p`.

There are two sources of ambiguity: first, `x` may, in general, have multiple values; this is solved by dispatching the call separately for each individual value. Second, classes may overlap, so even for a single value of `x` there can be multiple most specific definitions of `p`; this is solved by dispatching to each definition separately and unioning the results.

More formally, let us represent member predicates by *relation specifiers* of the form $C.p/n$, where C is the name of the class in which the predicate is declared, p is the name of the predicate itself, and n is the predicate's arity, not including the result parameter.[4] We say that a predicate $C.p/n$ *overrides* a predicate $C'.p/n$ if C is a transitive subtype of C'; in this case, we also say that $C.p/n$ is *more specific* than $C'.p/n$.

A member predicate is a *root definition* (or *rootdef* for short) if it does not override any other predicate. The set of rootdefs of a predicate is the set of all rootdefs that it overrides, or the predicate itself if it is already a rootdef. Note that due to multiple inheritance a predicate can have more than one rootdef, but every predicate has at least one.

[4] QL allows overloading, so there may in fact be multiple member predicates with the same arity as long as they have different parameter types. Like in Java, overloading is resolved entirely statically based on declared types, and hence plays no role in virtual dispatch, and we shall ignore it for simplicity.

The *static target* of a member predicate call `x.p(...)`, where the declared type of `x` is a class C, is the most specific predicate $D.p/n$ such that C is a reflexive, transitive subtype of D and n is the number of arguments in the call. In a valid QL program, every predicate call must have a unique static target. The *dispatch candidates* of `x.p(...)` are all the rootdefs of the static target, as well as any predicates that override at least one of the rootdefs.

At runtime, for every value v of `x`, the *applicable targets* of the call are those dispatch candidates $D.p/n$ for which v is in the extent of D, and the *actual targets* are the most specific applicable targets. The call is dispatched to all actual targets for each value of `x`.

In summary, dispatch for a call `x.p(...)` occurs in two stages, one static and one dynamic. At compile-time we compute the set of dispatch candidates, which contains all rootdefs of `p` above the declared type of `x` (that is, member predicates with the same signature that do not themselves override another definition) and all methods that override them. At runtime, each of these candidates applies only if the value of `x` is contained in the corresponding class, and there is no more specific class that also contains `x`.

For example, assume we add a member predicate `kind` to class `Digit` like this:

```
class Digit extends int { ... string kind() { result = "digit" } }
```

We override `kind` in the subclasses of `Digit` to result in `"even"` for `Even`, `"odd"` for `Odd` and `"even prime"` for `EvenPrime`. Now consider this query:

```
from Even e select e, e.kind()
```

The static target of the call `e.kind()` is `Even.kind`/0, whose (unique) root definition is `Digit.kind`/0. The dispatch candidates are `Digit.kind`/0, `Even.kind`/0, `Odd.kind`/0 and `EvenPrime.kind`/0. Since `e` is declared to be an `Even`, it ranges over the set $\{0, 2, 4, 6, 8\}$. For the runtime values 0, 4, 6 and 8, the applicable targets of `e.kind()` are `Digit.kind`/0 and `Even.kind`/0, and the (unique) actual target is `Even.kind`/0. For the value 2, the applicable targets are `Digit.kind`/0, `Even.kind`/0 and `EvenPrime.kind`/0, and the actual target is `EvenPrime.kind`/0. Hence, the query evaluates to {(0, `"even"`), (2, `"even prime"`), (4, `"even"`), (6, `"even"`), (8, `"even"`)}.

Now consider what happens if we add a new class

```
class Two extends Digit { Two() { this = 2 } string kind() { result = "two" } }
```

`Two.kind`/0 has `Digit.kind`/0 as its root definition, so it is now also a dispatch candidate for `e.kind()`, and it is an applicable target for $e = 2$. We now have *two* applicable targets in this case, neither of which is more specific than the other. Hence they will *both* be called, so the query additionally returns the tuple `(2, "two")`.

If, on the other hand, we define `Two` to extend `int` instead of `Digit`, its extent does not change, but `Digit.kind`/0 is no longer a root definition of `Two.kind`/0, which hence is no longer a dispatch candidate for `e.kind()`.

Discussion

Support for multiple actual targets is perhaps the most unusual feature of virtual dispatch in QL and can be confusing to novice QL programmers. Its main motivations are:

Naturality Virtual dispatch selects the most specific implementation; with overlapping classes there may be more than one, so in a logic language it is natural to take their disjunction.

Simplicity Outlawing multiple actual targets is difficult, since it is undecidable whether two classes overlap (by undecidability of emptiness in Datalog with negation). Requiring

characteristic predicates to be decidable would rule out many practically interesting cases, while checking ambiguity at runtime and aborting with an error seems undesirable.

Usefulness Some common idioms use this feature. For example, a flow analysis could be implemented as a member predicate on data flow nodes; different overriding definitions handle different kinds of flow, and hence naturally overlap. For instance, one definition could model intra-procedural def-use chains, while another models inter-procedural argument passing; both definitions overlap for parameters that are reassigned.

In particular, the last example above rules out a simpler approach where each predicate is considered its own rootdef, which would make dispatch depend very strongly on static types. As another extreme, one could consider all predicates of the right name and arity as dispatch candidates, ignoring static types altogether. This seems undesirable in practice, since it can cause dispatch to completely unrelated predicates that just happen to have the same name. Instead, QL views a rootdef and all its overriding methods as alternative implementations of the same operation. For a given call, all operations implemented by the static target are determined at compile time, and at runtime the most specific implementations are selected.

There is no intrinsic connection between multiple call targets and multiple inheritance: the former arises without the latter, for example if a class defines a predicate that is overridden by two overlapping subclasses. Ambiguous inheritance is, in fact, illegal in QL (as we shall discuss below), and hence does not give rise to multiple call targets.

2.6 Abstract classes

QL classes as we have described them so far lend themselves quite well to 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. In particular, the extent of a class is always defined by filtering its domain through the body of its character.

A classic example where this approach is useful is when modelling ASTs: the node types of an AST form a natural inheritance hierarchy, where, for example, there is a class `Expr` representing all expression nodes, with many different subclasses for different categories of expressions; there might, for instance, be a class `ArithmeticExpr` representing arithmetic expressions, which in turn could have classes `AddExpr` and `SubExpr`.

In other cases, however, we might prefer to instead think of a class as being the union of its subclasses. Here, the superclass exists purely as an interface that provides certain member predicates, with subclasses filling in concrete implementations.

QL supports a notion of abstract classes that allow us to do exactly this: like a concrete class, an abstract class has one or more superclasses and a characteristic predicate. However, the extent of an abstract class is not the set of values that satisfies its character, but rather the union of the extents of all its subclasses. In particular, an abstract class without subclasses has an empty extent. We will present a practical example of an abstract class in Section 4.

2.7 Miscellanea

QL has various other language features that are important in practice but are either not semantically fundamental, or have direct counterparts in other Datalog dialects.

As we mentioned at the beginning of this section, QL predicates may be recursive, and our program analysis queries make heavy use of this feature. A particularly common kind of recursion is transitive closure, for which QL offers a syntactic shorthand: for a binary predicate `p`, `p+` denotes its transitive closure, and `p*` its reflexive transitive closure. Obviously, this is purely syntactic sugar that is easily translated into plain recursion.

$prog$	::=	$\overline{cd}\ \overline{pd}$	program
cd	::=	$\textbf{abstract}^?\ \textbf{class}\ C\ \textbf{extends}\ \overline{T}\ \{C()\ \{f\}\ \overline{pd}\}$	class definition
pd	::=	$\textbf{predicate}\ p(\overline{T\ x})\ \{f\}$	predicate definition
f, g	::=	$p(\overline{x}) \mid x.p(\overline{y}) \mid C.\textbf{super}.p(\overline{x}) \mid \textbf{not}\ f$	formula
	|	$f\ \textbf{and}\ g \mid f\ \textbf{or}\ g \mid \textbf{exists}(T\ x \mid f)$	
S, T	::=	$C \mid @b \mid C.\texttt{domain}$	type reference

Figure 1 Syntax of Core QL; $\bar{\cdot}$ denotes (possibly empty) sequences, $\cdot^?$ optional elements

In addition to virtual calls, QL also provides statically dispatched `super` calls of the form `C.super.p(...)`. Their static target is looked up in `C` (which must be a superclass of the enclosing class), and serves as the single actual target.

The `import` statement makes definitions from one module available in another. For instance, the QL standard library for JavaScript is split into 40 modules, which are all imported into a single module `javascript.qll`. Since imports are transitive, queries can simply import that module to gain access to the entire library (cf. Listing 1). As in Java, implementation hiding is facilitated by access modifiers: member predicates may be marked `private`, meaning that they cannot be called from outside the enclosing class.

QL supports aggregates to perform arithmetic operations such as sum or average on (multi-)sets of values. While very useful in practice, aggregates are really a feature of the Datalog dialect into which QL is compiled, and they do not interact with the language's object-oriented features, hence we will not further discuss them.

As a syntactic convenience, casts may be written in a postfix form as `x.(A)`, which is semantically equivalent to `(A)x`, but saves parentheses in chained calls.

Finally, member predicates of abstract classes may themselves be abstract, meaning that they do not have a body, and the QL compiler checks that each subclass provides an overriding definition of the predicate. Observe that our definition of virtual dispatch guarantees that an abstract member predicate is never the actual target of a call: since the extent of the abstract class is the union of the extents of its subclasses and since each of those subclasses overrides the abstract predicate, there must always be at least one more specific applicable target. Thus, abstract predicates are not semantically fundamental, and in particular have no deep semantic connection with abstract classes.

3 Semantics of Core QL

To formally describe the semantics of QL, we concentrate on a subset dubbed *Core QL* that captures the object-oriented features of QL, while omitting other features that are either purely syntactic or are semantically orthogonal. The semantics of Core QL will be described by a translation to plain Datalog.

3.1 Core QL

Figure 1 presents the syntax of Core QL. Like full QL, Core QL programs consist of toplevel predicates and (concrete and abstract) classes with a characteristic predicate and member predicates. We do not model QL's `from-where-select` query syntax, but simply consider queries as special toplevel predicates.

Predicates can declare parameters, and their bodies are first-order formulas with predicate calls as atomic formulas. As in full QL, there are calls to toplevel predicates and to member predicates, and the latter may be either virtual calls or **super** calls. Unlike full QL, **super** calls always have to be explicitly annotated with the class they refer to.

Type references appearing in **extends** clauses, parameter declarations or existential quantifiers are either class names C or base type names $@b$. We assume the latter to be defined by an underlying database schema. Core QL also has a syntax for *domain types* of the form C.`domain` for a class name C; these cannot appear at the source level but play a crucial role in the semantics of classes.

Among the QL features omitted from Core QL are overloading, the function syntax for predicates, expressions, the **forall** quantifier, casts and **instanceof**: these can all be desugared into Core QL features. Other QL features such as primitive types and aggregates have no counterpart in Core QL, but their semantics is largely orthogonal to the object-oriented features of the language, which are the focus of our presentation.

3.2 Datalog

The target language for our translation is an untyped variant of Datalog. A Datalog program consists of a series of intensional predicate definitions of the form $p(\overline{x}) \leftarrow \varphi.$, where p is a predicate name, $\overline{x}$ is a possibly empty sequence of variable names, and φ is a formula of first-order logic with the usual logical connectives. The free variables of φ must be exactly $\overline{x}$. The atoms of φ are calls of the form $r(\overline{y})$, where r is either the name of an intensional predicate defined in the same program, or the name of an extensional predicate.

We say that an intensional predicate p calls a predicate q, written $p \rightarrow q$, if the body of p contains a call to q. As usual, $\rightarrow^*$ denotes the reflexive transitive closure of this relation. $p \stackrel{-}{\rightarrow} q$ means that one of the calls to q in p occurs under an odd number of negations.

We require all Datalog programs to be *stratified*, that is, recursive call chains of the form $p \rightarrow^* q \stackrel{-}{\rightarrow} r \rightarrow^* p$ are not allowed. Any stratified Datalog program has a least fixpoint semantics, that is, given an interpretation of the extensional predicates, each intensional predicate has a unique minimal interpretation that satisfies the predicate's definition.

3.3 Valid Core QL

In order to be meaningfully translatable to Datalog, a Core QL program has to fulfil a set of syntactic requirements and pass some static semantic checks. There is one additional check that is easiest to perform on the generated Datalog and will be discussed later.

The syntactic requirements are entirely standard and mostly naming related:

▶ **Definition 1** (Syntactic validity)**.** In order for a Core QL program to be *syntactically valid*, the following conditions have to be satisfied:

- No two classes and no two toplevel predicates with the same arity may have the same name; no two member predicates of the same class with the same arity, and no two parameters of the same predicate may have the same name.
- Every **extends** clause must list at least one type.
- Every characteristic predicate must have the same name as its enclosing class.
- No predicate parameter may have the name **this**.
- For every variable name appearing in a formula, there must either be an enclosing **exists** declaring a variable of that name, or the enclosing predicate must have a parameter of that name, or the variable name is **this** and it appears in a member predicate or character. In particular, every variable name can be associated with a declared type.

- Similarly, for every class name appearing in a type reference there must be a class of the same name, and for every predicate name appearing in a call to a toplevel predicate, there must be a toplevel predicate of that name with the appropriate arity.
- **super** calls may only appear in member predicates.

To formulate the static semantic checks, we first introduce some terminology.

▶ **Definition 2** (Relation specifiers)**.** A *relation specifier* $C.p/n$ consists of a class name C and a pair p/n, where p is a predicate name and n a natural number.

Unless otherwise specified, we require relation specifiers to be *valid*, that is, C must be the name of a class defined in the program, and C must declare a member predicate p with n parameters. We abbreviate $C.p/n$ as $C.p$ where n is not important or obvious from context.

▶ **Definition 3** (Subtyping)**.** The *subtyping* relation $S <: T$ is the smallest relation such that for every class C we have $C <: C.\texttt{domain}$, and if C extends T, then $C.\texttt{domain} <: T$.

As usual, $S <:^+ T$ denotes the transitive closure of this relation.

▶ **Definition 4** (Overriding)**.** $C.p/n$ overrides $D.p/n$, written $C.p/n \prec D.p/n$, if $C <:^+ D$. We write $C.p/n \preceq D.p/n$ to mean that either $C = D$ or $C.p/n \prec D.p/n$. If $D.p/n$ overrides no other member relation, it is a *rootdef*. We write $\rho(C.p/n)$ for the set of all rootdefs $D.p/n$ such that $C.p/n \preceq D.p/n$.

▶ **Definition 5** (Member predicate lookup)**.** We define a lookup function $\lambda(S, p, n)$ that looks up a member predicate in a type given a name and its arity and returns a set of candidates:

$$\lambda(S,p,n) = \begin{cases} \{C.p/n\} & \text{if } S = C \text{ and } C.p/n \text{ is valid} \\ \bigcup_{S<:T} \lambda(T,p,n) & \text{otherwise} \end{cases}$$

The static semantic checks guarantee that a program can be translated to Datalog:

▶ **Definition 6** (Translatability)**.** A syntactically valid Core QL program is *translatable* if the following conditions are met:

- It is not the case that $T <:^+ T$ for some type T; that is, the subtyping relation is acyclic.
- For every (not necessarily valid) relation specifier $C.p/n$, we have $|\lambda(C,p,n)| \leq 1$; in other words, classes must override ambiguously inherited predicates.
- For every member predicate call $x.p(\overline{y})$ where x has type T we have $\lambda(T,p,|\overline{y}|) \neq \emptyset$, i.e., all calls can be resolved to a static target.
- Similarly, for every call $D.\textbf{super}.p(\overline{x})$ in a member predicate of a class C, we must have $C <:^+ D$ and $\lambda(D,p,|\overline{x}|) \neq \emptyset$.

3.4 Translation to Datalog

The translation from (translatable) Core QL to Datalog is presented in Figure 2 as a family of structurally recursive translation functions:

- $\mathcal{T}_c$ translates Core QL class definitions into sequences of Datalog predicates, using an auxiliary function $\mathcal{K}$ to generate the extent predicate as explained below;
- $\mathcal{T}_m$ translates Core QL member predicates into Datalog predicates; it takes the declaring class of the member predicate as an additional argument;
- $\mathcal{T}_p$ translates toplevel Core QL predicates into Datalog predicates;
- $\mathcal{T}_b$ translates Core QL predicate and character bodies into Datalog formulas; it takes a type environment as an additional argument;

Translation of a class definition $cd \equiv \textbf{abstract}^? \textbf{ class } C \textbf{ extends } \overline{T}\ \{C()\ \{f\}\ \overline{pd}\}$:

$$\mathcal{T}_c(cd) := \begin{array}{lll} C.\texttt{domain}(\textbf{this}) & \leftarrow & \bigwedge_{C<:B} B.B(\textbf{this}) \wedge \bigwedge_{C<:@b} @b(\textbf{this}). \\ C.C(\textbf{this}) & \leftarrow & \mathcal{T}_b(f, \langle \textbf{this} := C.\texttt{domain} \rangle). \\ C(\textbf{this}) & \leftarrow & \mathcal{K}(cd). \\ \overline{\mathcal{T}_m(pd_i, C)} & & \end{array}$$

$$\mathcal{K}(cd) := \bigvee_{D<:C} D(\textbf{this}) \quad \text{if } cd \text{ is abstract}$$

$$\mathcal{K}(cd) := C.C(\textbf{this}) \quad \text{if } cd \text{ is concrete}$$

Translation of a toplevel predicate definition $pd \equiv \textbf{predicate}\ p(\overline{T\ x})\ \{f\}$:

$$\mathcal{T}_p(pd) := p(\overline{x}) \leftarrow \mathcal{T}_b(f, \langle \overline{x_i := T_i} \rangle).$$

Translation of a member predicate definition $pd \equiv \textbf{predicate}\ p(\overline{T\ x})\ \{f\}$:

$$\mathcal{T}_m(pd, C) := \begin{array}{lll} C.p(\textbf{this}, \overline{x}) & \leftarrow & \mathcal{T}_b(f, \langle \textbf{this} := C, \overline{x_i := T_i} \rangle). \\ C.p^{\text{disp}}(\textbf{this}, \overline{x}) & \leftarrow & (\bigwedge_{D.p \prec C.p} \neg D(\textbf{this})) \wedge C.p(\textbf{this}, \overline{x}). \end{array}$$

Translation of a predicate or character body f:

$$\mathcal{T}_b(f, \Gamma) := (\bigwedge_{(x,S)\in\Gamma} S(x)) \wedge \mathcal{T}_f(f, \Gamma)$$

Translation of a predicate call:

$$\begin{array}{lll} \mathcal{T}_f(p(\overline{x}), \Gamma) & := & p(\overline{x}) \\ \mathcal{T}_f(x.p(\overline{y}), \Gamma) & := & \bigvee_{R.p\in\rho(D.p)} (\bigvee_{B.p \preceq^* R.p} B.p^{\text{disp}}(x, \overline{y})) \quad \text{where } D.p := \lambda(\Gamma(x), p, |\overline{y}|) \\ \mathcal{T}_f(C.\textbf{super}.p(\overline{x}), \Gamma) & := & D.p(\textbf{this}, \overline{x}) \quad \text{where } D.p := \lambda(C, p, |\overline{x}|) \end{array}$$

Translation of other formulas:

$$\begin{array}{lll} \mathcal{T}_f(\textbf{not } f, \Gamma) & := & \neg\mathcal{T}_f(f, \Gamma) \\ \mathcal{T}_f(f \textbf{ and } g, \Gamma) & := & \mathcal{T}_f(f, \Gamma) \wedge \mathcal{T}_f(g, \Gamma) \\ \mathcal{T}_f(f \textbf{ or } g, \Gamma) & := & \mathcal{T}_f(f, \Gamma) \vee \mathcal{T}_f(g, \Gamma) \\ \mathcal{T}_f(\textbf{exists}(C\ x \mid f), \Gamma) & := & \exists x.\, (C(x) \wedge \mathcal{T}_f(f, \Gamma[x := C])) \end{array}$$

Figure 2 Translation from Core QL to Datalog (for readability, we write $C <: T$ to mean $C.\texttt{domain} <: T$)

- $\mathcal{T}_f$ translates Core QL formulas into Datalog formulas; it takes a type environment as an additional argument.

The type environments Γ used by $\mathcal{T}_b$ and $\mathcal{T}_f$ are partial functions from variable names to type references. We write $\langle x := T \rangle$ to denote the type environment that maps x to T, and contains no other mappings. As usual, $\Gamma[x := T]$ is a type environment that is identical to Γ except that it maps x to T. $\bigvee_{i \in I} \varphi_i$ and $\bigwedge_{i \in I} \varphi_i$ denote disjunctions and conjunctions of families of formulas indexed by a set I. For empty index sets we define $\bigvee_{i \in \emptyset} \varphi_i := \bot$ and $\bigwedge_{i \in \emptyset} \varphi_i := \top$, i.e., empty disjunctions are false and empty conjunctions are true.

We now discuss the individual translation functions in greater detail.

Classes

For every Core QL class C, we generate a definition for its domain predicate $C.\texttt{domain}$, its characteristic predicate $C.C$, and its extent predicate C. Additionally, each member predicate pd_i is translated recursively using $\mathcal{T}_m$.

The domain predicate is the intersection of the characteristic predicates of all supertypes of C. The characteristic predicate is generated from its Core QL definition, additionally enforcing prescriptive typing, which is not a feature of plain Datalog. The extent predicate, finally, is the Datalog predicate that actually defines the extent of the class. For concrete classes, this is the same as the characteristic predicate. For abstract classes, however, their extent is instead defined as the union of the extents of their subclasses.

The distinction between these three predicates is subtle, but crucial. $C.\texttt{domain}$ is mainly needed to circumscribe the type of **this** inside the characteristic predicate of C. To see why it cannot have type C, consider what the definitions of the characteristic predicate and the extent predicate would look like if it did:

$$\begin{array}{lcl} C.C(\textbf{this}) & \leftarrow & C(\textbf{this}) \wedge \ldots. \\ C(\textbf{this}) & \leftarrow & C.C(\textbf{this}). \end{array}$$

Note the recursion between $C.C$ and C, which is resolved by computing least fixpoints. Clearly, both rules are satisfied if $C.C$ and C are empty, and this is also the least fixpoint. In other words, typing **this** as C in the character would render every concrete class empty. Using type $C.\texttt{domain}$ instead breaks the recursion, and both predicates are now interpreted as the subset of the extent of $C.\texttt{domain}$ that satisfies the body of the character, as expected.

The distinction between the characteristic predicate $C.C$ and the extent predicate C is only relevant for abstract classes (and the two are indeed equal for concrete classes): the extent of an abstract class is not the extent of its characteristic predicate, but rather the union of the extents of its subclasses, and this is precisely how C is defined.

Predicates

Toplevel Core QL predicates are translated directly into Datalog predicates of the same name, using $\mathcal{T}_b$ to translate the body and enforce prescriptive typing for all parameters.

Member predicates $C.p$ are translated into two Datalog predicates: an implementation predicate of the same name, and a dispatch predicate $C.p^{\text{disp}}$. The latter is used during dispatch translation as explained below.

Formulas

Most Core QL formulas are straightforward to translate into their Datalog counterparts, except that for quantifiers we again enforce prescriptive typing. The two most interesting

cases are **super** calls and virtual calls. For the former, we simply use the λ function to look up the member predicate to invoke, explicitly passing in **this** as the first argument.

For virtual calls, we need to implement dispatch. Recall that for a call with static target $D.p$, the dispatch candidates are all member predicates that override a rootdef for $D.p$. Hence the call is translated into two nested disjunctions: the outer disjunction is over all rootdefs $R.p$ of the static target $D.p$, while the inner is over all methods overriding the rootdef. For each dispatch candidate $B.p$ identified in this way, we emit a call to $B.p^{disp}$, which in turn invokes $B.p$, but only if it is a most specific implementation of p for the given parameter.

The syntactic and static semantic checks ensure that the result of the translation is a valid Datalog program, except that they do not yet ensure stratification. While it would be possible to devise a QL-level check for this, it is conceptually simpler to check stratification of the generated Datalog, and map any violations of this condition back to the QL code it originated from. This is also how we implement the stratification check in our QL compiler.

3.5 Example

As a concrete example of the translation, we show the Datalog definitions generated for classes `Digit` and `Even` from Section 2, including definitions for the method `kind` and a query predicate that computes all even digits `e` and their kinds `k`.[5]

$$\texttt{Digit.domain}(\texttt{this}) \leftarrow \texttt{int}(\texttt{this}).$$
$$\texttt{Digit.Digit}(\texttt{this}) \leftarrow \texttt{Digit.domain}(\texttt{this}) \wedge \texttt{range}(\texttt{this}, 0, 9).$$
$$\texttt{Digit}(\texttt{this}) \leftarrow \texttt{Digit.Digit}(\texttt{this}).$$
$$\texttt{Digit.kind}(\texttt{this}, \texttt{result}) \leftarrow \texttt{Digit}(\texttt{this}) \wedge \texttt{string}(\texttt{result}) \wedge \texttt{result} = \texttt{"digit"}.$$
$$\texttt{Digit.kind}^{\text{disp}}(\texttt{this}, \texttt{result}) \leftarrow \neg\texttt{Even}(\texttt{this}) \wedge \texttt{Digit.kind}(\texttt{this}, \texttt{result}).$$
$$\texttt{Even.domain}(\texttt{this}) \leftarrow \texttt{Digit.Digit}(\texttt{this}).$$
$$\texttt{Even.Even}(\texttt{this}) \leftarrow \texttt{Even.domain}(\texttt{this}) \wedge \texttt{mod}(\texttt{this}, 2, 0).$$
$$\texttt{Even}(\texttt{this}) \leftarrow \texttt{Even.Even}(\texttt{this}).$$
$$\texttt{Even.kind}(\texttt{this}, \texttt{result}) \leftarrow \texttt{Even}(\texttt{this}) \wedge \texttt{string}(\texttt{result}) \wedge \texttt{result} = \texttt{"even"}.$$
$$\texttt{Even.kind}^{\text{disp}}(\texttt{this}, \texttt{result}) \leftarrow \texttt{Even.kind}(\texttt{this}, \texttt{result}).$$
$$\texttt{query}(\texttt{e}, \texttt{k}) \leftarrow \texttt{Even}(\texttt{e}) \wedge \texttt{string}(\texttt{k}) \wedge (\texttt{Digit.kind}^{\text{disp}}(\texttt{e}, \texttt{k}) \vee \texttt{Even.kind}^{\text{disp}}(\texttt{e}, \texttt{k})).$$

4 QL in Practice

The previous two sections have presented the semantics of QL in some detail, using toy examples for simplicity. We now show how these concepts apply in a more realistic setting.

4.1 Databases and schemata

Recall that QL programs (or rather, the Datalog programs into which they are translated) are run on a relational database. In practice, we use our own custom database system, but in principle QL programs could just as well be run on an off-the-shelf system.[6]

As in any relational database, data is represented in terms of tuples (rows), grouped into relations (tables) such that all tuples in a relation have the same arity (number of columns).

[5] Core QL does not include primitive types or arithmetic operations, so for the purposes of this example we have treated `int` and `string` like entity types, and assumed EDB relations `range(a,b,c)` and `mod(x,y,z)` corresponding to QL's range operator `a in [b..c]` and the modulo operator `x%y=z`, respectively.

[6] In fact, early versions of our compiler translated QL to SQL, using a third-party database system as our backend. Performance was disappointing, since typical QL compiles to very complex SQL with deeply nested joins and lots of recursion. Most SQL systems are not designed with such queries in mind, and hence handle them badly. On the other hand, our queries do not make use of complex extra-logical features such as database updates or transactions, which made it relatively easy to implement our own engine and saves us some performance overhead.

A table may have a distinguished *primary key* column, meaning that no two rows in the table have the same value in this column; in other words, each value in the primary key column uniquely identifies a row. A table t_1 may also have *foreign key* columns referring to the primary key column of a table t_2 ($t_1 = t_2$ is allowed), meaning that any value occurring in the foreign key column must also be present in the corresponding primary key column; in other words, each row of t_1 references a unique row of t_2.

Keys can be used to model hierarchical data structures using flat tables. Suppose, for instance, that we want to build a snapshot database representing a JavaScript program. Among other data, we may want to represent the abstract syntax tree of source files, including, in particular, all expressions. Simplifying somewhat, we could introduce a table `exprs` with four columns: a primary key column `id` with a unique ID for each expression; a column `kind` indicating what kind of expression we are dealing with, encoded as an integer; a foreign key column `parent` referencing the ID of the parent expression in the AST; and an integer column `idx` recording the ordering among children of the same parent.[7] Since `id` is a primary key, every expression is guaranteed to have a unique ID, and since `parent` is a foreign key, it is a well-defined reference to another expression in the same table.

For example, assume we want to represent a comparison expression `x === 1`; its two children are the variable reference `x` and the integer literal `1`. Assume further that we assign them the IDs 0, 1 and 2, and encode "equality expression" as kind 2, "variable reference" as kind 1, and "integer literal" as kind 0. The variable reference `x` thus has `id` 1, `kind` 1, `parent` 0, and `idx` 0, corresponding to the tuple `exprs(1, 1, 0, 0)`, while "`1`" gives rise to `exprs(2, 0, 0, 1)`. In practice, we would additionally store the name of the referenced variable and the value of the integer literal in separate tables, which we elide for simplicity.

At the storage level, all four columns of the `exprs` table look the same: they are just integers. QL, on the other hand, espouses a strongly typed view where keys are treated as opaque values and annotated with an entity type. Primary key columns define an entity type whose extent is the set of values occurring in that column. For instance, the `id` column of `exprs` could be annotated with the entity type `@expr`, meaning that it defines the extent of entity type `@expr`. Foreign key columns are also annotated with entity types, and the database system ensures that they only contain values drawn from the extent of their type.

This information about tables, the types of their columns, and the entity types they define is described by a *schema*, which thus defines the interface between a QL program and the database on which it is run.

4.2 Data abstraction

Given a snapshot database with a schema, we could write our analyses in plain Datalog, directly accessing the information stored in the tables. However, this can become quite cumbersome since we need to remember which column contains which piece of information, and is not robust against schema changes.

QL classes hide the specifics of how data is stored in tables behind a higher-level interface, thereby acting like abstract datatypes. For instance, we could implement a QL class `Expr` to provide an abstract view of the `exprs` table discussed above:

```
class Expr extends @expr {
  Expr getParent() { exprs(this, _, result, _) }
```

[7] Alternatively, each expression could keep references to their child expressions, but as different kinds of expressions have different numbers of children, this would require tuples with different arities, which would have to be stored in different tables.

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

Since `Expr` should contain *all* expressions represented in the database, it has a trivial character, and hence the same extent as `@expr`. The member predicate `getParent` provides access to the `parent` column of the `exprs` table, while `getChildExpr` enables navigation in the other direction. Note that we do not need to check that the index `i` is in range: if there is no i-th child, the predicate will simply fail to hold. QL also requires each class to define (or inherit) a `toString` member predicate, for which we provide a dummy implementation.

This interface allows us to navigate the program AST as a graph without being exposed to the details of its relational representation. For instance, `e.getParent+() = f` expresses the property that expression `e` is nested within expression `f` (using QL's "+" syntax for transitive closure).

If all client analyses use `Expr` instead of directly accessing the EDB, we can easily change our data representation later on. For instance, it may not be desirable to record the parent expression directly in the `exprs` table, since toplevel expressions do not have a parent expression. Instead, the parent-child relation could be stored in a separate three-column table `expr_nesting(child,parent,idx)`. The first two columns are foreign keys, so they must refer to properly defined `@expr` values, but there is no requirement that every `@expr` value appears in the first column (or, for that matter, the second column), so expressions without parents can now be modelled.

If we want to switch to this representation, we can simply update the definitions of `getParent` and `getChildExpr` without affecting any client analyses:

```
class Expr extends @expr {
  Expr getParent() { expr_nesting(this, result, _) }
  Expr getChildExpr(int i) { expr_nesting(result, this, i) } ...
```

4.3 Inheritance

Class `Expr` abstracts away from the details of the relational encoding of the AST and is useful for implementing generic syntax tree traversal, but if we want a richer semantic interface we have to implement subclasses of `Expr`. For instance, we could implement a class `EqExpr` to exclusively represent equality checks (and no other expressions):

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

The characteristic predicate filters out those expressions that do not have kind 2 (which, in our example encoding, represents equality). We provide getter predicates for the two operands of the equality, further abstracting away from the details of our AST representation, and we override the `toString` predicate to provide a more specialised string representation. Similar classes can be implemented for variable references, literals, and other expressions.

QL classes thus allow us to impose an abstract data type representation on relational data. Since classes can freely overlap, we can even implement multiple representations for the same data. For instance, we could overlay a control flow graph structure on top of the AST by defining a class `CFGNode` that also extends `@expr`, but presents it under a different interface, offering, say, a method `getASuccessor()` to compute call graph successors.

4.4 Overriding

As a practical example of overriding, consider implementing the member predicate `Expr.isPure` used in Listing 1. Its default implementation in class `Expr` is `none()`, which is a built-in predicate that always fails. In other words, we conservatively assume that all expressions are impure, and override it in subclasses:

```
class Expr extends @expr { predicate isPure() { none() } ...
```

In class `Literal`, we override `isPure` as `any()`, a built-in predicate that always succeeds:

```
class Literal extends Expr { predicate isPure() { any() } ...
```

As another example, equality checks are pure if all of their children are:

```
class EqExpr extends Expr {
  predicate isPure() { forall (Expr c | c = this.getChildExpr(_) | c.isPure()) } ...
```

4.5 Interface vs Implementation

Abstract classes support decoupling interface and implementation even further: while `Expr` implements an interface in terms of one particular set of EDB relations, abstract classes specify only an interface, which may be implemented in multiple different ways.

As an example, assume we want to implement an analysis for JavaScript to find comparisons between expressions with incompatible (dynamic) types, which will always evaluate to `false` at runtime. Assume further that we have implemented a binary predicate `incompatTypes(e, f)` that infers possible types of `e` and `f` and checks whether they are compatible. Using class `EqExpr` defined above, we could implement our analysis as follows:

```
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."
```

Other JavaScript language constructs that compare values in the same way include, e.g., the `switch` statement. If we want to consider them in our query, we could add another disjunct to the `where` part, but this would make it less readable, and we would need to keep extending it for any other equality tests we want to support. Instead, we introduce an abstract class capturing the common interface for all equality tests:

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

Like all classes, `EqualityTest` needs a superclass: we choose `ASTNode`, which is a common superclass of `Expr` and `Stmt` defined in the JavaScript QL libraries. The interface defined by `EqualityTest` consists of member predicates to access the left and right operands of the comparison. We allow a single equality test to have multiple left or right operands; e.g., in a `switch`, every `case` is viewed as a right operand of the comparison.

We can implement this interface on `EqExpr` and `SwitchStmt` by introducing new classes that have the same extent as `EqExpr` and `SwitchStmt`, respectively, but extend `EqualityTest`:

```
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() } }
```

The extent of `EqualityTest` now contains all equality expressions and all switch statements, under a convenient interface for our query:

```
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."
```

To add support for other kinds of equality tests, all we need to do is to define new subclasses of `EqualityTest`; the query need no longer be changed.

4.6 Optimisation

In practice, the translation shown in Figure 2 can produce very inefficient Datalog, particularly when translating virtual calls: the disjunction over all candidates can be quite large, and in many cases the context restricts the receiver variable in such a way that some disjuncts end up always being false, which would lead to a lot of wasted computation if evaluated naively. For example, in the translation shown in Section 3.5, the `query` predicate restricts `e` to `Even`, so the dispatch disjunct $\texttt{Digit.kind}^{\text{disp}}(\texttt{e}, \texttt{k})$ can never apply. Another source of inefficiency are superfluous type guards for variables that are already restricted sufficiently by their uses. For instance, the conjunct `string(result)` in `Even.kind` is implied by `result="even"` and hence unnecessary.

One could devise a more sophisticated compilation scheme that avoids this, but we choose to instead perform these optimisations at the Datalog level, keeping the translation as simple as possible: infeasible dispatch disjuncts are simply a special case of formulas that logically contradict other formulas in their context and hence are equivalent to false, while unnecessary type tests are logically implied by other formulas in their context and hence equivalent to true. Both cases can be detected by a form of type inference [15, 33]. We also apply various standard optimisations such as inlining, join ordering and the magic sets transformation [7]; the latter two rely on (compile-time) estimation of (run-time) relation sizes [34].

Ultimately, the example from Section 3.5 is simplified by our optimiser to

```
Even(this) ← range(this, 0, 9) ∧ mod(this, 2, 0).
query(e, k) ← Even(e) ∧ k = "even".
```

5 Case Study

To demonstrate the benefits of QL in implementing static checks, we reimplemented Error Prone (`http://errorprone.info`) in QL. Error Prone is a bug finding tool for Java that integrates with the compiler and checks for common mistakes, reports them and suggests possible fixes. As of version 2.0.4, there are 101 checks. We reimplemented the checks (but not the fix suggestions), ensuring that they pass all the unit tests of the original. This required about one man-month of effort by an experienced QL programmer.

Error Prone is originally implemented in Java, comprising about 10,500 lines of code, not including supporting libraries such as the `javac` Compiler Tree API.[8] Our reimplementation, by contrast, is slightly less than 2,000 lines of code, not including the QL standard library for Java. However, our implementation only covers the checks themselves, not the suggested fixes. Manual inspection suggests that the latter account for about 1,100 lines of code in the Java implementation, leaving 9,400 lines of analysis code.

[8] That is, counting only files in the `src/main/java/com/google/errorprone/bugpatterns` directory.

Listing 2 QL code for detecting nested null checks in Java

```
// a "==" test where one operand is a null literal
class NullCheck extends EQExpr { NullCheck() { getAnOperand() instanceof NullLiteral } }

// "inner" is nested inside the then−branch of "outer", and both check nullness of "v"
predicate nestedNullCheck(IfStmt outer, IfStmt inner, Variable v) {
  inner.getParent+() = outer.getThen() and
  outer.getCondition().(NullCheck).getAnOperand() = v.getAnAccess() and
  inner.getCondition().(NullCheck).getAnOperand() = v.getAnAccess() }
```

Java is famously verbose, which explains part of the size difference, although QL is overall syntactically quite similar to Java. If we exclude Java `package` declarations and `@Override` annotations (which have no QL counterparts) and `import` statements (the number of which is largely determined by the organisation of the supporting libraries), we can subtract a further 2,800 lines from Error Prone and 100 lines from our implementation. In other words, the Java implementation is 3.5x the size of the QL implementation.

This is mostly because AST traversal and filtering, which require lots of boilerplate code in Java, can be expressed very concisely using recursion and prescriptive typing in QL. For example, Listing 2 shows part of a query for finding incorrect uses of double-checked locking: `NullCheck` picks out comparisons to `null`, and `nestedNullCheck` identifies nested `if` statements that check the same variable for nullness. Recursion is used to check the nesting condition. The casts to `NullCheck` would fail in a descriptive interpretation, since `s.getCondition()` is not a `NullCheck` for most `if` statements `s`. In QL, they act as filters, restricting `outer` and `inner` to those `if` statements that do, in fact, check nullness.

To assess scalability, we ran both the original Error Prone analyses and our reimplementations on a recent snapshot of Apache Hadoop,[9] which is about 1.5 MLoC. Four of the Error Prone checks failed with null pointer exceptions, the remaining 97 finished in 46 seconds. Our reimplementation is about 4x slower, at 201 seconds for the same 97 analyses.[10]

The performance difference is not so much due to the use of Java instead of QL, but to a fundamentally different execution model: while Error Prone checks are performed on a per-file basis, our queries run on a database representing the entire program, and hence are implicitly global. This reflects different usage scenarios: Error Prone is normally run during compilation or as an IDE service, hence instantaneous feedback is important. Our analysis usually runs offline, often even on a dedicated machine, hence a runtime of several minutes on a large code base (and memory requirement of a few gigabytes) is entirely acceptable. Ultimately, these differences are orthogonal to the choice of language, and there is no fundamental obstacle to running local QL analyses on small databases representing one file each.

In summary, this case study shows that QL allows us to quickly implement static analysis checks as concise and scalable queries, though the whole-program approach of our implementation imposes a performance overhead.

As of March 2016, Semmle's static analysis platform offers about 2500 individual analyses for eight languages (C/C++/Objective-C, C#, Cobol, Java, JavaScript, PL/SQL, Python, Scala), all implemented in QL. While some analyses are shallow local checks similar to Error Prone, many crucially depend on whole-program analysis. This is especially true for the

[9] Commit SHA 855d529 from `https://git-wip-us.apache.org/repos/asf/hadoop.git`.
[10] As measured on an Intel Core i7-4900MQ laptop, with 1GB of heap allocated to the analysis. Timings do not include compilation time for Error Prone and database construction time for QL.

dynamically typed languages, where global invariants that a statically typed language would express in the type system instead have to be derived by flow analysis. For instance, our Python analysis suite includes checks for common mistakes such as using undefined attributes or hashing unhashable objects, which sit on top of a whole-program points-to analysis also implemented in QL. As another example, our Java and C++ suites include security analyses based on inter-procedural taint tracking that identifies vulnerabilities to common attacks such as cross-site scripting or SQL injection. All these analyses are in daily use by our customers on multi-million line code bases. A thorough discussion of technical details is out of scope for this paper, but the advantages of using Datalog for static analysis are well-known [35], to which QL adds the benefits of object-oriented modularity and reusability.

Besides implementing static checks, we also use QL for structural analysis to determine dependencies between different parts of a code base that are then visualised in an architectural design tool. Finally, we use QL as a meta-query language to write queries over static analysis results, comparing and correlating results across multiple revisions of the same code base in order to track introductions and fixes of defects over the history of a code base [6].

6 Discussion and Related Work

In this section, we will discuss QL's object-oriented features in the light of popular definitions of object orientation in the literature, and then proceed to survey related work.

6.1 Discussion

The deliberately minimalist examples of Section 2 perhaps make the gap between QL classes and methods and their more traditional counterparts appear wider than it is. Recall that in practice QL programs rely on an extensional database, and here the parallels become quite striking: tuples in a table are records with columns as fields; primary keys uniquely identify tuples, and hence play the role of addresses; foreign keys uniquely reference other tuples, thus acting like references to other records. Hence, a database can be viewed as a strongly typed heap containing a collection of objects with fields that may contain primitive values or references to other objects, where objects with the same layout are collected into tables.

A QL class like `Expr` whose extent is (a subset of) the primary key column of a table thus describes a set of records; its member predicates can access record elements, using the primary key for lookup. Subclasses inherit member definitions and can override them, allowing different implementations of the same operation for different objects, the appropriate implementation being chosen at runtime based on the dynamic type of the object.

Of course, QL classes are not restricted to this particular setting, since classes can be sets of arbitrary values, not just primary keys, and EDB tables can be arbitrary relations, for which there is no parallel in other object-oriented languages.

QL fits the folklore definition of object orientation as "data abstraction plus inheritance": Section 4 shows how QL classes achieve the former by abstracting from the concrete layout of database tables; inheritance in QL is entirely conventional, and, as usual, can be used both for implementing an interface and for code reuse.

A more refined characterisation is given by Cook [11] (building on the classic definition by Wegner [39]), whereby object orientation is support for the dynamic creation and use of objects. An object, in turn, is a first-class, dynamically dispatched *behaviour*, where a behaviour is a collection of named operations, and dynamic dispatch means that different objects can implement the same operations in different ways.

QL satisfies the second half of this definition: classes associate named operations with arbitrary values, and subtyping and virtual dispatch allow different implementations of the same operations. The first half of Cook's definition is *not* satisfied, as QL programs cannot create new values, which is arguably a natural limitation for a query language. QL also does not support mutable state, and has no notion of object identity as opposed to value identity, but Cook, in fact, argues that these features are not essential to object orientation.

Ullman [37] claims that query languages cannot be "seriously logical and seriously object-oriented at the same time". His argument is framed in the context of a radical reinterpretation of the relational model, where tuples and relations are understood as objects with relational operators as methods. In Ullman's analysis, this would mean that each tuple computed during evaluation is distinct from any previously computed tuple, conflicting with the set semantics of Datalog and its least fixpoint semantics. Furthermore, it is unclear what types to assign to relational operators such union or join in such a setting. Our approach differs significantly from this model: QL classes range over values, not tuples, sidestepping his first point. Relational operators are primitive language constructs, not methods, and hence not typed. In Ullman's view, this might disbar QL from being "seriously" object-oriented, but we have argued that it nevertheless provides the usual benefits of object orientation.

Proposals for integrating object identity and logic programming range from explicitly exposing object identifiers [42] to sophisticated extensions of the underlying logic [24]. Our experience with QL suggests that object identity is not a prerequisite for object orientation.

In summary, QL supports data abstraction and inheritance with dynamic dispatch, widely considered as hallmarks of object orientation. Less conventionally, QL supports overlapping classes and relational member predicates, which are quite natural for a logic query language, while object creation and mutation do not fit this paradigm well and hence are not supported.

6.2 Related Work

Encoding hierarchical data in relational form is conceptually quite straightforward, but querying the encoded data in a traditional relational language is cumbersome. This has been termed the *object-relational impedance mismatch*, and led to calls for replacing the relational model with models directly supporting structured data [5]. We will not discuss the literature on this topic, since QL is based on a completely conventional relational data model. Our approach agrees in spirit, if not in detail, with the so-called Third Manifesto [13], which argues that object orientation should be built on, rather than supplant, the relational model.

Object-oriented extensions of Datalog have been studied in the literature before. Abiteboul et al. [2] propose a language where individual rules for predicates may be associated with classes. They consider three variants of overriding, one of which, termed *static inheritance*, is somewhat similar to QL's approach, although they define overriding at the level of individual rules, not entire predicates. Since their language has no static types they have no concept of rootdefs, instead considering all methods with the right name and arity as dispatch candidates; as we have argued, this makes dispatch highly non-local and brittle in the presence of overlapping classes. They also do not consider multiple inheritance. While for the most part they assume that classes are defined directly by the EDB, they also briefly consider "virtual" classes (first proposed in [1]), which, like concrete QL classes, are defined by a characteristic predicate (though it is unclear whether they can be recursive). Their proposal never seems to have been implemented, making it hard to gauge its practicality.

Extensions of Prolog with subtyping and inheritance have also been proposed [3, 36]. These approaches focus on types for structured terms and on performing unification modulo subtyping between term constructors, which are not available in Datalog.

The idea of storing source code in a database and exploring and analysing it using relational queries goes back at least to Linton [27], who used the INGRES database system. To overcome the limited expressiveness of standard database query languages, several authors have suggested the use of Prolog [23, 14], which, however, tends to suffer from scalability problems on large code bases. Aiming for a middle ground, Paul et al. [31] propose writing queries in a relational algebra with transitive closure, while Consens et al. [10] employ a subset of Datalog just powerful enough to express properties of paths in graphs.

Many new languages specifically designed for code exploration have been proposed; we discuss just a few examples: ASTLOG [12] focuses on AST traversal, permitting a very efficient implementation. PQL [22] allows more general queries over graphs, while JQuery [21] is based on a full-featured, Prolog-like logic language. JTL [9] only supports querying Java source code, allowing for a very concise and specialised syntax; its expressive power is that of first order logic with transitive closure. Martin et al. [28] propose a DSL for matching sequences of events on objects both statically and dynamically, based on the bddbddb Datalog system [40], which has also been used directly for program analysis.

Our own work in the area started with CodeQuest [20], which compiled Datalog queries to SQL. An early version of QL, which likewise compiled to SQL and only supported analysing Java, was described in [17, 16], where it was presented informally through examples. The present paper provides a more rigorous description of QL including new features such as abstract classes and recursive characters, presents an experimental case study, and (perhaps most importantly) gives a formal semantics. Apart from its potential use in exploring the metatheory of QL, the semantics has already proved useful in clarifying subtle semantic issues such as the type of `this` in characters, which was left implicit in previous work.

The benefits of Datalog for specifying and implementing highly scalable flow analyses [8, 35] have recently been demonstrated on the LogicBlox platform [4], which extends Datalog with support for database updates and creating fresh values. It would be interesting to investigate whether these concepts could be fruitfully combined with QL's object-oriented features.

Graph databases have also been suggested as a basis for source code exploration and analysis. Ebert et al. [18] use the custom graph query language GreQL, while Urma et al. [38] show that the Cypher language of Neo4j (`http://neo4j.com`) allows simple queries to be expressed concisely and evaluated efficiently even over large code bases. Neither language appears to provide abstraction mechanisms like user-defined predicates, making them less well-suited for implementing more complex analyses. Compiling QL to GreQL or Cypher is not possible in general, since they do not seem to provide native support for recursion (besides transitive closure), which is heavily used by typical QL analyses. The four example queries shown by Urma can be written in QL just as concisely (5, 4, 8 and 6 lines, respectively), and execute efficiently (0.4s, 5s, 0.1s, 4s) on a recent snapshot of OpenJDK (2.5MLoC), somewhat faster than the runtimes they report on a similar-sized code base.

Another interesting alternative are metaprogramming languages like Rascal [25], which provide seamless integration of program analyses with code generation and transformation tools. However, their suitability for deep analyses has not been shown yet.

Virtual dispatch in QL is a form of predicate dispatch [19], with class characters as guards. While exhaustiveness is ensured (that is, each call has at least one dispatch candidate), we make no attempt to prevent ambiguity: calls with more than one actual target are fully supported. Also, our translation from QL to Datalog is non-modular and requires reasoning about the entire program, unlike more recent work on predicate dispatch [30].

Prescriptive type systems have been studied in the context of Prolog [41, 26]. As pointed out by Meyer [29], prescriptive type annotations are essentially runtime type checks; hence,

in spite of its static type declarations QL is, in some sense, dynamically typed.

7 Conclusion

We have presented QL, an object-oriented dialect of Datalog with classes, subtyping and dynamic dispatch, which we have described both informally and through a translation to plain Datalog: classes are unary predicates representing sets of values; subtyping is set inclusion; and dynamic dispatch resolves calls in the smallest class containing the receiver value. As a typical application of QL, we have shown its use in implementing static checks, and presented a case study highlighting its advantages in this domain over Java. Finally, we have discussed QL's merits as an object-oriented language: while it is missing object creation and mutable state, QL does offer the twin features of abstraction and dynamic dispatch, usually considered to be at the heart of object-oriented programming, without relying on objects in the traditional sense. Apart from QL's practical usefulness, its model of object orientation is an interesting contribution in itself, which, we hope, will spur further discussion of and investigation into the nature of object-oriented programming.

References

1 Serge Abiteboul and Anthony J. Bonner. Objects and views. In *SIGMOD*, 1991.
2 Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. Methods and rules. In *SIGMOD*, 1993.
3 Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *JLP*, 3(3), 1986.
4 Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, 2015.
5 Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *DOOD*, 1989.
6 Pavel Avgustinov, Arthur I. Baars, Anders S. Henriksen, R. Greg Lavender, Galen Menzel, Oege de Moor, Max Schäfer, and Julian Tibble. Tracking static analysis violations over time to capture developer characteristics. In *ICSE*, 2015.
7 François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, 1986.
8 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.
9 Tal Cohen, Joseph Gil, and Itay Maman. JTL: The Java tools language. In *OOPSLA*, 2006.
10 Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *ICSE*, 1992.
11 William R. Cook. On understanding data abstraction, revisited. In *OOPSLA*, 2009.
12 Roger Crew. ASTLOG: A language for examining abstract syntax trees. In *DSL*, 1997.
13 Hugh Darwen and C. J. Date. The third manifesto. *SIGMOD Records*, 24(1), 1995.
14 Stephen Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems. In *PLDI*, 1996.
15 Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type inference for Datalog and its application to query optimisation. In *PODS*, 2008.
16 Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. In *GTTSE*, 2007.

17 Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. .QL for source code analysis. In *SCAM*, 2007.

18 Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO—generic understanding of programs. *ENTCS*, 72(2), 2002.

19 Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP*, 1998.

20 Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *CodeQuest*: Scalable source code queries with Datalog. In *ECOOP*, 2006.

21 Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *AOSD*, 2003.

22 Stan Jarzabek. Design of flexible static program analyzers with PQL. *TSE*, 24(3), 1998.

23 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*, 1992.

24 Michael Kifer and James Wu. A logic for object-oriented logic programming. In *PODS*, 1989.

25 Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *SCAM*, 2009.

26 T. L. Lakshman and Uday S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *ISLP*, 1991.

27 Mark Linton. Implementing relational views of programs. In *SDE*, 1984.

28 Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: A program query language. In *OOPSLA*, 2005.

29 Gregor Meyer. On types and type consistency in logic programming. Technical Report Informatik Berichte 199, FernUniversität Hagen, 1996.

30 Todd D. Millstein, Christopher Frost, Jason Ryder, and Alessandro Warth. Expressive and modular predicate dispatch for java. *TOPLAS*, 31(2), 2009.

31 Santanu Paul and Atul Prakash. A query algebra for program databases. *TSE*, 22(3), 1996.

32 Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming*. 1988.

33 Max Schäfer and Oege de Moor. Type inference for Datalog with complex type hierarchies. In *POPL*, 2010.

34 Damien Sereni, Pavel Avgustinov, and Oege de Moor. Adding magic to an optimising Datalog compiler. In *SIGMOD*, 2008.

35 Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog Reloaded*, 2010.

36 Gert Smolka and Hassan Aït-Kaci. Inheritance hierarchies: Semantics and unification. *JSC*, 7(3/4), 1989.

37 Jeffrey D. Ullman. A comparison between deductive and object-oriented database systems. In *DOOD*, 1991.

38 Raoul-Gabriel Urma and Alan Mycroft. Source-code queries with graph databases—with application to programming language usage and evolution. *SCP*, 97, 2015.

39 Peter Wegner. Dimensions of object-based language design. In *OOPSLA*, 1987.

40 John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *APLAS*, 2005.

41 Jiyang Xu and David S. Warren. Semantics of types in logic programming. Technical Report DPS-102, ECRC, Munich, 1990.

42 Carlo Zaniolo. Object identity and inheritance in deductive databases—an evolutionary approach. In *DOOD*, 1989.