Algebraic Data Types for Object-oriented Datalog

MAX SCHÄFER, PAVEL AVGUSTINOV, OEGE DE MOOR, Semmle

Datalog is a popular language for implementing program analyses: not only is it an elegant formalism for concisely specifying least fixpoint algorithms, which are the bread and butter of program analysis, but these declarative specifications can also be executed efficiently. However, plain Datalog can only work with atomic values and offers no first-class support for structured data of any kind. This makes it cumbersome to express algorithms that need even very simple data structures like pairs, and impossible to express those that need trees or lists. Hence, non-trivial analyses tend to rely on extra-logical features that allow creating new values to represent compound data on the fly. We propose a more high-level solution: we extend QL, an object-oriented dialect of Datalog, with a notion of algebraic data types that offer the usual combination of products, disjoint unions and recursion. In addition, the branches of an algebraic data type can be full-fledged QL predicates, which may be recursive not only with other data types but with arbitrary other predicates, enabling very fine-grained control over the structure of the data type. The new types integrate smoothly with QL's existing notions of classes and virtual dispatch, the latter playing the role of a pattern matching construct. We have implemented our proposal by extending the QL evaluator with a low-level operator for creating fresh values at runtime, and translating algebraic data types into applications of this operator. To demonstrate the practical usefulness of our approach, we discuss three case studies tackling problems from the general area of program analysis that were previously difficult or impossible to solve in QL.

CCS Concepts: •**Software and its engineering → Abstract data types;** *Object oriented languages; Constraint and logic languages;*

ACM Reference format:
Max Schäfer, Pavel Avgustinov, Oege de Moor. 2017. Algebraic Data Types for Object-oriented Datalog. 1, 1, Article 1 (April 2017), 24 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

It has been said (Wirth 1976) that "algorithms + data structures = programs". In program analysis, many of the most important algorithms are least fixpoint computations on subset lattices. The logic programming language Datalog is a natural choice for expressing such algorithms: being a first-order logic with recursion, it is rich enough to allow elegant, declarative specifications of fixpoint algorithms, yet simple enough to admit aggressive optimisation and efficient evaluation on relational database systems (Aref et al. 2015; Semmle 2017a).

Consequently, Datalog-based program analysis has a long research pedigree, and has recently seen a revival, with systems such as Doop (Bravenboer and Smaragdakis 2009) and the Semmle platform (Avgustinov et al. 2016) demonstrating its viability for real-world analysis tasks. Usually, an *extractor* (not written in Datalog) is first used to create a database with a representation of the program to be analysed, for example in the form of three address code as used by Doop, or by encoding the entire AST structure as in the case of Semmle. The analyses themselves are then written as Datalog queries that are evaluated over this database and yield relations representing the analysis results. For example, the result of a Datalog-based pointer analysis might be a binary relation between program variables and abstract objects they may point to.

DOI: 10.1145/nnnnnnn.nnnnnnn

While Datalog has proved its mettle in expressing program analysis algorithms, data structures are another matter: plain Datalog simply offers no support at all for expressing and working with structured data. Programs can only use atomic values, typically including primitive values like numbers or strings, as well as any entity values that appear in the underlying database. Entity values can be used to encode references and thereby represent complex data structures (Avgustinov et al. 2016), but this can only be done at database creation time. The program itself operates in a fixed universe of values: any structured value that isn't already available in the database is simply not denotable.[1]

Other logic programming languages, such as Prolog, come with built-in support for structured values, but this tends to complicate their semantics and makes them more difficult to implement efficiently. While high-performance Prolog engines are an active area of research (Hermenegildo et al. 2012; Swift and Warren 2012), we are not aware of any implementations that are as stable and fast as their Datalog counterparts.

We briefly discuss three typical examples of program analyses that need structured data of one kind or another.

First, many analyses work on a control flow graph (CFG) representation of the program. CFG edges can be very easily and naturally computed in Datalog from, say, the program AST, but there is no way of creating new entities representing the CFG nodes. It is tempting to recycle AST nodes to represent CFG nodes, but this is problematic since AST and CFG do not correspond cleanly to each other: some AST nodes are simply syntax without CFG semantics, while conversely CFG entry and exit nodes can only be mapped onto the AST with difficulty. Alternatively, the extractor could create entities for representing CFG nodes at database creation time, but this causes an awkward split of the CFG construction across different analysis phases that is difficult to work with. In particular, it introduces an undesirable dependence of the extractor on the analysis, since changes to the CFG construction might now necessitate changes to the way the database is created.

As a second example, consider converting the program under analysis to static single assignment (SSA) form. In SSA form, each source variable in the original program is split up into multiple SSA variables, each of which have precisely one definition, and variable uses are renamed to refer to the most recently defined SSA variable. Crucially, this requires introducing *phi nodes*, which are pseudo-assignments that merge the values of multiple SSA variables at join points in the CFG. While the placement of phi nodes can be beautifully expressed in Datalog, there is no way of creating new entities to represent them. Phi nodes can be characterised as a pair (n, x) of a CFG node n and a (source) variable x, so a Datalog program could deal with SSA variables by carrying around n and x in separate (Datalog) variables, but this is tedious and error prone. Alternatively, the extractor could be pressed into service to create entities for representing phi nodes, but this basically amounts to doing full SSA conversion in the extractor, losing the benefits of a high-level, declarative specification.

Our final example is context-sensitive points-to analysis. Here, structured values are needed to express abstract values and to express contexts. As an example of the latter, a 2-CFA analysis deals with contexts that are pairs of call sites (c_1, c_2) such that c_1 may call the function containing c_2, which in turn may call the function currently being analysed. Similarly, abstract values in points-to analysis are generally pairs of the form (k, a), where k is a context (itself, as we have seen, a structured value), and a is an allocation site that may be analysed in context k. Again, these compound values can be represented in Datalog by using separate (Datalog) variables to hold the individual components, which we have already argued is error-prone. Allocating entities for all possible contexts and abstract values in the extractor is not a viable choice, since, for example, not all pairs of call sites are valid contexts, and the set of contexts actually needed during the analysis is smaller still.

In summary, all these examples show the need for structured values in program analysis. In some cases these values can be emulated by explicitly tracking their components, and in other cases the extractor can enrich the database sufficiently to introduce entities for representing the structured values ahead of time, but neither solution is generally applicable, and both diminish the attractiveness of implementing the analysis in Datalog.

[1] At a higher level, of course, Datalog programs do create structured values, in that they define relations, which are sets of tuples. But relations are not first-class values, and cannot be operated on by the program itself (for example, relations cannot take other relations as arguments).

In practice, Datalog-based program analysis systems tend not to restrict themselves to pure Datalog: some only express core algorithms in Datalog and fall back on other languages for the rest of the analysis (Lhoták and Hendren 2004; Whaley et al. 2005); others, notably Doop (Bravenboer and Smaragdakis 2009), are implemented end-to-end in Datalog, but rely on language extensions to emulate structured values.

In particular, Doop makes crucial use of *constructors*, an extension to Datalog that allows inventing new values during program execution. A predicate that is declared as a constructor has a special output parameter that is not defined or referenced in the predicate body; instead, at runtime for each tuple $\overline{v}$ of values that the ordinary parameters of the predicate are bound to by its body, a fresh value is created and assigned to the output parameter. This value uniquely identifies the tuple $\overline{v}$ and can, for all intents and purposes, be used as if it were that tuple.

For example, to represent 2-CFA contexts one could define[2] a constructor predicate `TwoCfaContext(c1, c2, k)` where `c1` and `c2` are ordinary parameters defined by the body of `TwoCfaContext` to range over the set of pairs that should be considered as contexts, and `k` is the special output parameter. At runtime, the engine introduces for each tuple (v_1, v_2) that satisfies the body of `TwoCfaContext` a fresh value $k(v_1, v_2)$, and adds the tuple $(v_1, v_2, k(v_1, v_2))$ to the relation `TwoCfaContext`.

While this is sufficient for encoding simple compound values, constructor predicates cannot be recursive (either with themselves or with other predicates), so more complex tree or list structures cannot be expressed.

We propose to instead extend Datalog with algebraic data types, an approach for specifying structured data types that has proved its worth in the functional programming community. In its general form, an algebraic data type is a union of one or more *branch types*, each of which, in turn, is a tuple type. The union is kept disjoint by tagging tuples from a branch type with the name of the branch, so even if two branches are the same when considered as tuple types their values will not be merged. Moreover, the component types of a branch type can (directly or indirectly) reference the enclosing algebraic data type, which allows representing recursive data structures such as lists or trees.

Unlike functional or imperative programming languages, where types are meta-level entities that belong to a different conceptual realm than the programs they describe, typed dialects of Datalog tend to view types simply as a kind of unary relation, which may either be defined in the underlying database or by the program itself. We adopt this view for our algebraic data types and their branch types.

This, however, immediately raises a problem: recursive data types are generally infinite, so trying to interpret them as unary predicate definitions and computing them in full will lead to non-termination. To avoid this, we could handle them specially and introduce some sort of lazy evaluation mechanism to only construct as many of their values as needed, but this would be quite a disruptive extension to the language semantics and would mostly negate the advantage of conceptual simplicity we gain from treating types as unary predicates.

Instead, we observe that while data types may be infinite, a given (terminating) program only ever uses a finite subset of it, so we add another feature to our algebraic data types (besides union, tupling and recursion) which provides fine-grained control over the extent of the data type: each branch type can have a *branch body* that restricts the set of tuples that go into the branch type. By providing branch bodies, algebraic data types can be restricted to only contain those values that the program actually needs, thus ensuring that they are finite and can be evaluated like any other predicate. Another interesting feature of branch bodies is that they can, naturally, be recursive, not only with other branch bodies, but with arbitrary other predicates. As we will see, this allows specifying data types that are much more finely structured than algebraic data types in functional languages.

One important feature of algebraic data types in functional languages that we do *not* support is polymorphism: under Datalog's types-as-predicates approach a polymorphic type would correspond to a higher-order predicate, which is far beyond the expressive power of Datalog.

[2] We use a simplified syntax for expository purposes; see Section 8.5 of the LogicBlox Reference Manual (LogicBlox 2017) for full details.

We have implemented our proposal as an extension of QL (Avgustinov et al. 2016), an object-oriented dialect of Datalog with classes and virtual dispatch that compiles down to plain Datalog without classes. At the language level, algebraic data types are introduced as a new kind of types. While orthogonal to classes, the two can be combined freely and naturally, with virtual dispatch playing the role of a pattern matching construct. To provide runtime support, we have extended the Datalog evaluator underlying QL with a *tuple numbering* operator, which is similar to LogicBlox's constructors, but permits recursion. We show how algebraic data types can be compiled to applications of this tuple numbering operator.

We briefly study the metatheory of tuple numbering, showing that it fits smoothly into Datalog's least-fixpoint semantics and interacts well with common optimisations. It also provides a dramatic boost to expressiveness, making plain Datalog without primitive types, which can only express polynomial algorithms, Turing-complete.

Moving from theoretical considerations to practical experience, we report on three case studies tackling problems from the general area of program analysis: we discuss an implementation of the Cartesian Product Algorithm, a context sensitivity strategy that employs very precise list-structured contexts; a library for building control flow graphs for Java from an AST representation; and a parser for regular expressions that produces ASTs. All three problems are hard or impossible to solve without language support for structured values.

In summary, our contributions are as follows:

- We propose an extension of QL, a dialect of Datalog, with monomorphic algebraic data types.
- We demonstrate how these data types can be implemented by translating them into applications of a low-level tuple numbering operator.
- We show that tuple numbering is Turing complete, yet semantically well-behaved.
- We present three case studies demonstrating the practical usefulness of our proposal.

In the rest of the paper, we will motivate the need for algebraic data types in more detail by means of an extended example (Section 2), then describe their syntax and semantics (Section 3) and explore their theoretical properties (Section 4). After a brief discussion of our implementation (Section 5) we present three case studies showing practical applications in Section 6 before surveying related work in Section 7 and concluding in Section 8.

2 BACKGROUND AND MOTIVATION

This section introduces QL by example, and motivates the need for algebraic data types. As our running example we show how to implement SSA conversion (Cytron et al. 1991).

Assume we have encoded a flow-graph representation of a program using the three binary relations described by the schema in Figure 1: `succ` is the successor relation between nodes, while `def` and `use` record definitions and uses of variables, respectively. The columns of these relations are typed using the entity types `@cfg_node` and `@variable`, meaning that the values contained in these columns should be viewed as *entity values*, that is, opaque identifiers modelling some external entities (in this case, flow graph nodes and variables).

The relations `succ`, `def` and `use` are called *extensional* relations, since they are defined explicitly by storing their *extent* (that is, the tuples they contain) in the database. This contrasts with *intensional* relations that are defined implicitly by QL predicates and evaluated on top of the database.

The entity types `@cfg_node` and `@variable` are also extensional relations: they are unary relations, *i.e.* sets, whose elements are entity values. Annotating a column of an extensional with an entity type means that any value stored in that column must be contained in the entity type.

This demonstrates two key principles of QL: types (with the exception of built-in types like **int** and **string**) are unary relations, and for a program entity to be of a type means that all its potential values are contained in the type. Like ordinary predicates, types can be either extensional or intensional: extensional types are entity types, of which we have already seen examples, and intensional types are classes, which we will encounter below.

```
succ(@cfg_node m, @cfg_node n); def(@cfg_node n, @variable v); use(@cfg_node n, @variable v);
```

Fig. 1. Extensional relations encoding a flow-graph representation of a program

```
predicate startsBB(@cfg_node n) {
  not succ(_, n) or
  exists(@cfg_node p, @cfg_node q | succ(p, n) and succ(q, n) and p != q) or
  exists(@cfg_node p, @cfg_node q | succ(p, n) and succ(p, q) and n != q)
}

class BasicBlock extends @cfg_node {
  BasicBlock() { startsBB(this) }
  @cfg_node getNode(int i) {
    i = 0 and result = this or
    succ(getNode(i-1), result) and not startsBB(result)
  }
  BasicBlock getAPredecessor() { exists(int i | succ(result.getNode(i), this)) }
  predicate dominates(BasicBlock that) { /* implementation omitted */ }
  predicate inDominanceFrontierOf(BasicBlock that) {
    that.dominates(getAPredecessor()) and not (that.dominates(this) and this != that)
  }
}
```

Fig. 2. A basic-block program representation defined in QL

As a first step towards SSA conversion, we abstract the node-based flow graph into a graph of basic blocks. So we first define an (intensional) QL predicate `startsBB` shown at the top of Figure 2 that picks out those nodes that start a new basic block: entry nodes (that is, nodes without predecessors), join nodes with more than one predecessor, and branch successor nodes that have a predecessor with more than one successor.

Next, we define a QL class to model basic blocks. In QL, a *class* is simply a set of values for which a set of predicates (that we might think of as *methods*) with a distinguished parameter **this** are defined. Each class extends one or more other types (possibly themselves classes), and may define a *characteristic predicate*, which can also refer to **this** and syntactically looks like a no-argument constructor in Java. The extent of a class contains precisely those values for **this** that are contained in all supertype extents and in the characteristic predicate.

Our class `BasicBlock` is shown in the bottom half of Figure 2: it has `@cfg_node` as its only supertype and its characteristic predicate requires that `startsBB(`**`this`**`)`. In other words, the extent of `BasicBlock` are exactly those `@cfg_nodes` `n` for which `startsBB(n)` holds.

Now we define four member predicates: `getNode`, which looks up a node in a basic block by index; `getAPredecessor`, which navigates the basic block-level flow graph; `dominates`, which determines dominance between basic blocks and whose implementation we have omitted for space reasons; and finally `inDominanceFrontierOf` to computes dominance frontiers, which will be used later to place phi nodes.

Predicates `getNode` and `getAPredecessor` use QL's functional syntax, which allows us to (syntactically) treat them as functions returning a result, as shown in the recursive call `getNode(i-1)`. Inside the body of these predicates, an implicitly declared **result** variable is available that is used to refer to the return value. The functional syntax is purely syntactic sugar, and QL neither assumes nor guarantees that a predicate using this syntax is, in fact, a function: while `getNode` happens to be one, `getAPredecessor` is not. Such "multi-valued" functions are, however, very natural to use in a logic programming language.

```
predicate ssaDef(BasicBlock bb, @variable v) { def(bb.getNode(_), v) or phi(bb, v) }

predicate phi(BasicBlock bb, @variable v) {
  exists(BasicBlock defbb | ssaDef(defbb, v) and bb.inDominanceFrontierOf(defbb))
}
```

Fig. 3. Phi node placement in QL

Note that QL supports arithmetic (cf. predicate `getNode`); in combination with recursion, this makes it possible to write infinite, and hence non-terminating, predicates. QL also supports negation (cf. predicate `inDominanceFrontierOf`), but restricts its use in recursive predicates by requiring *parity stratification*, that is, any recursive cycle between predicates must go through an even number of negations.

Having established a basic block representation of our program, we now proceed to implement SSA conversion proper. In SSA form, each program variable is split into one or more *SSA variables*, each of which have a *single* definitions. A definition of an SSA variable is either an explicit definition of a program variable, or an implicit *phi node* that is inserted into the flow graph at points where two or more definitions of a variable are merged.

As is well known, a phi node for a variable `v` needs to be inserted at the beginning of each basic block `bb` that is in the dominance frontier of another basic block `defbb` that defines `v`, either by an explicit definition or by a previously inserted phi node. Inserting a phi node may in turn trigger the insertion of other phi nodes.

QL's least fixpoint semantics allows a very succinct implementation of phi node placement, shown in Figure 3: predicate `phi(bb, v)` determines if a phi node for `v` is needed at the beginning of `bb` using the dominance frontier criterion, and `ssaDef(bb, v)` records the fact that basic block `bb` contains an SSA definition of `v`.

Elegant as this implementation is, it does not give us a good *representation* of SSA definitions. The best we can do is to treat SSA definitions as tuples `(bb, v)` for which `ssaDef(bb, v)` holds. This is awkward, since tuples are not first-class values in QL, so every variable that may hold SSA definitions would have to be split up into two auxiliary variables to hold the components of the tuple. Care has to be taken to carry around these variables in unison and not to accidentally mix up components from different tuples. With algebraic data types, we can represent tuples as first-class values, which solves this problem.

Another problem is that representing explicit definitions as pairs `(bb, v)` is too imprecise: a single basic block may contain multiple definitions of the same variable, which we would often like to distinguish, but they are conflated in the pair representation. We could include the index of the defining node in our representation, talking about triples `(bb, i, v)` instead of pairs `(bb, v)`, but this representation is not very suitable for phi nodes, which do not correspond to actual flow nodes. We could assign them a dummy index, say -1, but that is a workaround rather than a solution. At the end of the day, the most natural thing to do is to represent explicit definitions by triples, and phi nodes by pairs. With algebraic data types, values arising from different branches of the type can have different arities, which solves this problem.

Borrowing Haskell syntax, we might consider representing SSA definitions using an algebraic data type `SsaDef` with two branches `Def` and `Phi`, defined like this:

```
data SsaDef = Def BasicBlock int @variable | Phi BasicBlock @variable
```

However, this does not fit very well into the conceptual model of QL, where types are just unary predicates: `SsaDef` contains infinitely many values (since the second component of `Def` can be any integer), and thus cannot be evaluated like a normal predicate. We could make special provisions for lazily evaluating algebraic data types, but this would substantially complicate the language semantics and introduce a jarring mismatch between algebraic data types and other QL types.[3]

[3] Of course, primitive types like **int** have similar problems, and they are indeed treated specially in QL, but primitive types are built into the language and there are very few of them, while algebraic data types are user-defined.

```
newtype SsaDef =
    Def(BasicBlock bb, int i, @variable v) { def(bb.getNode(i), v) }
  or Phi(BasicBlock bb, @variable v) {
      exists(SsaDefinition def | def.getVariable() = v and bb.inDominanceFrontierOf(def.getBasicBlock()))
    }
class SsaDefinition extends SsaDef {
  abstract BasicBlock getBasicBlock();
  abstract @variable getVariable();
}
class ExplicitDefinition extends SsaDefinition, Def {
  BasicBlock getBasicBlock() { this = Def(result, _, _) }
  @variable getVariable() { this = Def(_, _, result) }
}
class PhiNode extends SsaDefinition, Phi {
  BasicBlock getBasicBlock() { this = Phi(result, _) }
  @variable getVariable() { this = Phi(_, result) }
}
```

Fig. 4. An algebraic data type for describing SSA variables

Instead, we observe that while there are infinitely many `SsaDef` values, we are only interested in finitely many of them, namely those that represent actual SSA definitions. If we allow the branches of algebraic data types to restrict the possible values of their parameters so as to construct only those values that are actually needed, then algebraic data types can be evaluated like any other predicates and harmony is restored.

To this end, the branches of an algebraic data type in QL may have a body that computes the set of tuples that the branch ranges over, as shown at the top of Figure 4: branch `Def` of type `SsaDef` is defined as

```
Def(BasicBlock bb, int i, @variable v) { def(bb.getNode(i), v) }
```

meaning that it ranges over those tuples `(bb, i,v)` for which `def(bb.getNode(i), v)` holds, and no other tuples. The branch body of `Phi` implements the phi node placement algorithm discussed above.

To make it easier to implement, we define classes `SsaDefinition`, `ExplicitDefinition` and `PhiNode` that correspond, respectively, to the algebraic data type `SsaDef` as a whole, and to the two branch types `Def` and `Phi`. These classes define member predicates `getBasicBlock` and `getVariable` for extracting the relevant bits of information from SSA definitions, which are used in `Phi` to determine basic blocks that need a phi node.

Note that we use a branch name like `Def` for two distinct purposes: it can act as a *branch type*, that is, a unary predicate, or as an *injector predicate* with four parameters (three explicitly declared ones and an implicit result parameter). In fact, these two are different predicates that happen to both be called `Def`. In QL syntax, there is never any ambiguity between the two.

The branch type `Def` is a subtype of `SsaDef` and can be used in declarations, such as the **extends** clause of its corresponding class `ExplicitDefinition`. The injector predicate `Def` can either be thought of as a value "constructor" that creates elements of the branch type `Def` given values for its parameters `bb`, `i` and `v`, or as a "destructor" that extracts the components `bb`, `i` and `v` from a given value of `Def`. It is in this latter role that `Def` is used in the member predicate definitions of class `ExplicitDefinition`.

Unlike the distinction between branch types and injector predicates, however, the distinction between "constructors" and "destructors" is purely pedagogical; there is only one injector predicate.

Finally, we note that branch bodies can be recursive with each other and with normal predicates, and this is indeed the case in our example: `Phi` calls `SsaDefinition.getVariable`, which is overridden by class `PhiNode` to call `Phi`. As noted above, language extensions for modelling structured data in other Datalog dialects do not permit such recursion, but we have found it to be a very useful and powerful tool in practice.

$$
\begin{array}{lcll}
\mathit{prog} & ::= & \overline{cd}\;\overline{td}\;\overline{pd} & \text{program} \\
\mathit{td} & ::= & \textbf{newtype}\ A = \overline{bd} & \text{algebraic data type definition} \\
\mathit{bd} & ::= & B(\overline{T x})\{f\} & \text{branch definition} \\
f, g & ::= & \ldots \mid y = B(\overline{x}) & \text{formula} \\
S, T & ::= & \ldots \mid A \mid B & \text{type reference}
\end{array}
$$

Fig. 5. Extensions to CoreQL (Avgustinov et al. 2016) to incorporate algebraic data types

3 SYNTAX AND SEMANTICS

We now proceed to give a precise description of the syntax and semantics of our algebraic data types. Building on a previous description of the semantics of QL (Avgustinov et al. 2016) in terms of a core calculus CoreQL and its translation to untyped Datalog, we extend CoreQL with algebraic data types and show how it translates to Datalog with a novel tuple numbering operator for creating new values at runtime. For the convenience of the reader, we reproduce the definition of CoreQL in Appendix A.

3.1 Syntax and validity rules

Figure 5 shows the syntax of our algebraic data types as an extension of CoreQL: in addition to classes and predicates, programs can also declare algebraic data types. Each such declaration associates a type name A with a list of branch declarations $\overline{bd}$. Each branch, in turn, has a name B, a list of parameters $\overline{T\ x}$ and a body, which is a formula f. In full QL, the body may be omitted, in which case it defaults to the always-true formula `any()`.

We also add a new kind of formulas of the form $y = B(\overline{x})$, where y a variable name, B a branch name, and $\overline{x}$ a list of variable names. In full QL, we instead introduce a new kind of expression $B(\overline{e})$, where $\overline{e}$ a list of argument expressions, which can be desugared into its CoreQL counterpart by introducing temporary variables.

Finally, data type names A and branch names B are added to the set of type names, so they can appear in variable declarations and the **extends** clauses of classes.

In addition to CoreQL's syntactic validity requirements (cf. Appendix A), we require that no two types (whether they be classes, data types or branches) and no two parameters of the same branch have the same name, and that each data type have at least one branch; type references to data types and branches must correspond to a type or branch definition; and for each formula $y = B(\overline{x})$ there must be a branch named B with the appropriate arity.

Additionally, we introduce consistency rules that ensure programs do not mix up values from different algebraic data types. Besides preventing logic errors, this also allows the implementation to reuse identifiers across tuple numberings. We introduce a *universe* U_A for each algebraic data type A, which is the set of all values of that type, and one additional universe U_0 containing all values that do not belong to algebraic data types.

Definition 3.1 (Universe assignment). To each type in a translatable CoreQL program (cf. again Appendix A), we assign at most one universe:

- The universe of an algebraic data type A is U_A.
- The universe of a branch type B is the universe of its enclosing algebraic data type.
- The universe of an entity type $@b$ is U_0.
- For a class C with supertypes $\overline{T}$, if all supertypes are from the same universe U, then that is also the universe of C and C.`domain`.

Note that the last clause is well-defined, since translatable CoreQL programs have acyclic type hierarchies.

We require CoreQL programs to be universe consistent in the following sense:

Definition 3.2 (Universe consistency). A translatable CoreQL program is *universe consistent* if:

(1) For each class C with supertypes $\overline{T}$, all supertypes are from the same universe U (thus guaranteeing that each class has a universe).
(2) For each call $p(\overline{x})$ or $y.p(\overline{x})$, the type of each argument variable x_i is from the same universe as the type of the corresponding parameter z_i of the called predicate.
(3) For each formula $y = B(\overline{x})$, the type of y is from the same universe as B, and the type of each argument variable x_i is from the same universe as the type of the corresponding parameter z_i of B.
(4) For each member predicate $p(\overline{S\ x})$ that overrides another predicate $p(\overline{T\ z})$, each S_i is from the same universe as the corresponding T_i.

In our implementation for full QL we use the QL compiler's type inference mechanism (Schäfer and de Moor 2010) to detect additional type errors. In general, the QL compiler considers any part of the program that it can show to be unsatisfiable as a type error (even if there is no consistency violation). Its type inference algorithm is parameterised over a *type hierarchy* that allows stating relationships between types as arbitrary monadic first-order formulas. For an algebraic data type A with branch types $B_1, \dots, B_n$, we augment the type hierarchy with inclusion facts $\forall x\colon B_i(x) \implies A(x)$ and disjointness facts $\neg\exists x\colon B_i(x) \wedge B_j(x)$ (where $i \neq j$). This allows us, for instance, to detect code that erroneously attempts to treat a value from one branch as belonging to a different branch, which will never yield any results at runtime.

3.2 Datalog with tuple numbering

The target language of our translation is an untyped variant of Datalog extended with a tuple numbering operator, which we now describe in more detail.

A Datalog program is a set of rules of the form $p(\overline{x}) \leftarrow \varphi$ where p belongs to the set $\mathbb{I}$ of *intensional relation symbols* each of which is associated with an arity; $\overline{x}$ is a vector drawn from the set $\mathbb{V}$ of *element variables*, whose length is the same as the arity of p; and φ is a formula of first-order logic. The set of free variables of φ must be exactly $\overline{x}$, so every parameter of p is free in the body and vice versa. φ may make use of constant symbols (but no function symbols) and refer to relations both from $\mathbb{I}$ and the set $\mathbb{E}$ of *extensional relation symbols* (which is disjoint from $\mathbb{I}$), subject to parity stratification. It may also use equality and the usual connectives and quantifiers of first-order logic. Additionally, φ may contain sub-formulas of the form $z = \#r(\overline{y})$, where $\overline{y}$ and z are element variables and $r \in \mathbb{I} \cup \mathbb{E}$ is a relation symbol of the appropriate arity.

The semantics of a Datalog program is computed over a *structure* $\langle \mathcal{D}, \mathcal{E}, \#^i \rangle$, where $\mathcal{D}$ is a non-empty set (also called the *domain*); $\mathcal{E}$ is an *interpretation* that assigns to each n-ary extensional relation symbol $e \in \mathbb{E}$ a set of n-tuples over $\mathcal{D}$; and $\#^i$ is a family of injective *tuple-numbering functions* from $\mathcal{D}^i$ to $\mathcal{D}$, one for each natural number i. Note that we do *not* require the ranges of different tuple-numbering functions to be disjoint.

To define the meaning of formulas φ, we additionally need a *relation assignment* $\mathcal{I}$ and a *variable assignment* σ; the former is similar to $\mathcal{E}$ in that it assigns sets of domain tuples to relation symbols, but for intensional relation symbols from $\mathbb{I}$; the latter maps element variables to elements of $\mathcal{D}$. A satisfaction judgment $\langle \mathcal{D}, \mathcal{E}, \#^i \rangle \models_{\mathcal{I},\sigma} \varphi$ can now be defined by structural induction on φ in the usual way, using $\mathcal{I}$ and $\mathcal{E}$ to look up relation symbols and σ for element variables. The only new case is for tuple numbering: writing $\sigma[\overline{y}]$ for the n-tuple of values assigned to $\overline{y}$ by σ, we define $\langle \mathcal{D}, \mathcal{E}, \#^i \rangle \models_{\mathcal{I},\sigma} z = \#r(\overline{y})$ to hold if $\sigma[\overline{y}] \in (\mathcal{I} \cup \mathcal{E})(r)$ and $\sigma(z) = \#^n(\sigma[\overline{y}])$.

Assuming that the program is stratified, rule bodies can be interpreted as monotonic maps over their free relation variables. This is a well-known result for standard Datalog, and our definition of the semantics of tuple-numbering is monotonic as well, as we will discuss in more detail below. Hence, intensional predicates can be semantically interpreted as the least fixpoints of their defining rules, yielding the overall semantics of the Datalog program.

3.3 Translating algebraic data types to Datalog

Finally, we show the translation from CoreQL with algebraic data types to Datalog with tuple numbering. For ease of reference, Figures 10 and 11 in Appendix A reproduces the translation from plain CoreQL to Datalog (Avgustinov et al. 2016), on which we base our definitions.

Intuitively, the idea is to first treat each branch B of a data type A as a normal predicate B_{dom} that computes all tuples that satisfy the branch body. Then we tuple-number B_{dom} to obtain a predicate $B_{\#}$ that assigns identifiers to the tuples in B_{dom}. We cannot directly gather up the identifiers produced by the $B_{\#}$ predicates for the various branches to obtain A, since different tuple numberings are not guaranteed to produce disjoint identifiers, so two branches $B_{\#}$ and $C_{\#}$ might produce overlapping identifiers. Instead, we define a predicate A_{dom} containing of all pairs (b, i), where i is an identifier produced by some $B_{\#}$, and b is a constant uniquely representing that branch B among all other branches of A. For concreteness, we will choose the string "B" for this purpose, but any other constant would do just as well. Finally, we tuple-number A_{dom}, yielding a predicate $A.A$ whose output enumerates the set representing A. The two steps of this encoding process correspond to the two type-forming operations of product and disjoint sum types that together form the basis of algebraic data types.

Thinking operationally for a moment, to "construct" a value $B(\overline{v})$ of A we first use $B_{\#}$ to compute an inner identifier i for $\overline{v}$, and then apply $A.A$ to the pair ("B", i) to obtain its outer identifier, which is the value representing $B(\overline{v})$ as an element of A. Conversely, to "destruct" an element of A we can apply $A.A$ in reverse to decode it into a pair ("B", i) that tells us which branch it came from and what its inner identifier in that branch is, at which point we can use $B_{\#}$ to recover the underlying tuple. In a logic programming language, of course, predicates are not "applied" forwards or backwards, they statically describe a relation that can be navigated in any direction; hence the same predicates can be viewed as constructors or destructors, depending on context.

Making our informal description precise, each branch definition $B(\overline{T\ x})\{f\}$ of a data type A gives rise to four Datalog predicates: B_{dom}, which interprets f as a normal predicate body; $B_{\#}$, which tuple-numbers B_{dom} to obtain inner identifiers for all its tuples; $B.B$, which maps those inner identifiers to outer identifiers belonging to the enclosing data type A; and B, which projects $B.B$ onto its co-domain and hence contains those elements of A that are generated by B. Formally, this looks as follows:[4]

$$\begin{array}{lcl}
B_{\text{dom}}(\overline{x}) & \leftarrow & \mathcal{T}_b(f, \langle\overline{x_i := T_i}\rangle). \\
B_{\#}(\overline{x}, y) & \leftarrow & y = \#B_{\text{dom}}(\overline{x}). \\
B.B(\overline{x}, z) & \leftarrow & \exists y\colon B_{\#}(\overline{x}, y) \wedge A.A(\text{“B”}, y, z). \\
B(z) & \leftarrow & \exists \overline{x}\colon B.B(\overline{x}, z).
\end{array}$$

Each data type definition A, in turn, induces three Datalog predicates: A_{dom} collects the tuple numbers assigned by the $B_{\#}$ predicates of the branches into one set, tagging each with the name of the branch it came from. $A.A$ tuple-numbers A_{dom} to obtain identifiers for these tagged inner tuple numbers, and A again projects $A.A$ down to its last column, yielding the set of all elements in A.

$$\begin{array}{lcl}
A_{\text{dom}}(b, y) & \leftarrow & \bigvee_B b = \text{“B”} \wedge \exists \overline{x}\colon B_{\#}(\overline{x}, y). \\
A.A(b, y, z) & \leftarrow & z = \#A_{\text{dom}}(b, y). \\
A(z) & \leftarrow & \exists b, y\colon A.A(b, y, z).
\end{array}$$

Finally, we define how to translate $y = B(\overline{x})$ to Datalog:

$$\mathcal{T}_b(y = B(\overline{x}), \Gamma) \quad = \quad B.B(\overline{x}, y)$$

[4] See Appendix A for the definition of the translation function $\mathcal{T}_b(f, \Gamma)$, taken from (Avgustinov et al. 2016), which translates a QL formula f to a corresponding Datalog formula, with the type environment Γ mapping QL variables to their declared types.

The existing rules for translating variable declarations in CoreQL already ensure that a variable with type A or B is constrained to range over the elements in the unary predicates of the same name, so no special rules are needed to handle variables declared to be of an algebraic data type or branch type.

It is perhaps worth noting that non-recursive algebraic data types can be encoded directly in Datalog without the need for a tuple numbering operator: assuming that all branches of the data type A have the same arity n (which can be achieved by padding with dummy values or repeating tuple components) each variable x of type A can be represented as $n + 1$ component variables $x_0, x_1, \ldots, x_n$, where $x_1, \ldots, x_n$ represents a tuple of values and x_0 is a tag indicating which branch it is from. This does not work if A is recursive, as in that case one of the components could itself be of type A (or another type depending on A).

3.4 Algebraic data types and classes

Algebraic data types and classes are semantically completely orthogonal. QL classes do not create new values; they simply describe subsets of already existing values and provide an interface for working with them. Algebraic data types, on the other hand, do create new values, but offer no data abstraction features. Indeed, they do not need to, since we can simply define a class that extends the type and defines member predicates on it.

In practice, a common pattern is to have one class for each branch type and a superclass for the overall type as in the SSA example of Section 2. The latter usually declares abstract predicates which are then implemented by the former, with one implementation per branch. Sometimes multiple branches use the same implementation, which can be accommodated by factoring out an intermediate class to hold the shared predicate.

Because QL classes can overlap, they can implement different interfaces for the same set of values. This allows us to implement pattern matching on algebraic data types using use virtual dispatch, similar to Scala's case classes (Odersky and Zenger 2005).

Assume we want to match on a value a from an algebraic data type A, with clauses $f_1, \ldots, f_n$ corresponding to the branches $B_1, \ldots, B_n$ of A. Each clause f_i is a formula that may refer to the branch parameters of B_i, but initially we assume it has no other free variables. To encode this in QL, we first define a new subclass of A that declares a single abstract predicate representing the pattern matching:

```
class AMatcher extends A { abstract predicate match(); }
```

Then, for each branch $B_i(T_1\ x_1, \ldots)$ we define a subclass of `AMatcher` that overrides `match` to apply f_i:

```
class B_iMatcher extends AMatcher, B_i { predicate match() { exists(T_1 x_1, ... | this = B_i(x_1, ...) and f_i) } }
```

The matching can now be encoded as a QL formula `a.(AMatcher).match()`:[5] at runtime, QL's normal dispatch machinery will choose the implementation of `match` from the most specific subclass of `AMatcher` that contains `a`, so if `a` belongs to branch type B$_i$, its implementation B$_i$`.match`, which simply wraps f_i, will be evaluated.

Additional free variables in match clauses can be accommodated by lifting them to parameters of `match`. If we want to add a catch-all clause f_0 that applies if no other branch matches, we can simply turn `AMatcher.match` into a concrete predicate with body f_0; dispatch semantics ensures that f_0 is only evaluated if no more specific definition applies. Note that our encoding does not provide exhaustiveness checking, which would need special support from the compiler.

4 METATHEORY

In this section, we prove a few results about the tuple numbering operator we have added to Datalog: we show that tuple numbering is monotonic and hence needs no special semantic treatment; it admits context-pushing optimisations; and it is Turing complete.

Theorem 4.1. *Tuple numbering is monotonic in the sense that if $\langle \mathcal{D}, \mathcal{E}, \#^i \rangle \models_{\mathcal{I},\sigma} z = \#r(\overline{y})$ holds and $\mathcal{I}'$ is an assignment such that $\mathcal{I}(r) \subseteq \mathcal{I}'(r)$ for all $r \in \mathrm{dom}(\mathcal{I})$, then $\langle \mathcal{D}, \mathcal{E}, \#^i \rangle \models_{\mathcal{I}',\sigma} z = \#r(\overline{y})$ holds as well.*

[5] Recall that QL uses postfix casts, so `a.(AMatcher)` means "the value of `a`, considered as an element of class `AMatcher`".

Proof. Immediate from the definition. As mentioned above, this property is important because it means that rules in Datalog with tuple numbering are monotonic maps over assignments like in plain Datalog, so they have a well-defined least fixpoint semantics that can be computed by bottom-up evaluation as usual. □

The QL compiler performs a variety of whole-program optimisations on the Datalog program it generates. Most of these optimisations amount to logical rewrites, so we need to clarify the interaction between tuple numbering and other logical operators.

Our first result concerns the interaction between conjunction and tuple numbering.

Theorem 4.2. *Tuple numbering commutes with conjunction: replacing a formula $\varphi \wedge z = \#r(\overline{y})$, where the free variables of φ are contained in $\overline{y}$, with $z = \#r'(\overline{y})$, where r' is a newly defined intensional predicate $r'(\overline{y}) \leftarrow \varphi \wedge r(\overline{y})$, does not change program semantics.*

Proof. Without loss of generality, we assume that $\varphi \wedge z = \#r(\overline{y})$ is itself the body of a rule defining an intensional predicate. Then it is easy to check that the least-fixpoint model of the old program can be extended to the least-fixpoint model of the new program by assigning r' all those tuples that satisfy $\varphi \wedge r(\overline{y})$ under the model of the old program. Both models assign the same meaning to all relation symbols except for r', which does not exist in the old program. □

This is important because many of the most important whole-program optimisations the QL compiler performs rely on context pushing: a predicate q that is used in a conjunction together with some other predicate p can be specialised by pushing the call to p into the body of q, thereby making q smaller and less expensive to compute. The theorem states that this is safe even if q uses tuple numbering.

However, the same is not true of other logical operators, which, fortunately, are not used for inter-procedural optimisations by the QL compiler.

Theorem 4.3. *Tuple numbering does not commute with disjunction, negation, or existential quantification.*

Proof. To ease notation, we write $y = \#\varphi$ for arbitrary formulas φ, meaning the program obtained by lifting φ into a new intensional predicate.

Counterexample for disjunction: $\top \vee x = \#\top \equiv \top \not\equiv x = \#\top$; for negation: $\neg(x = \#\top) \not\equiv \bot \equiv x = \#(\neg\top)$.

For existential quantification, let $p(x)$ be a predicate that holds for at least two values of x. Then $\exists x\colon y = \#p(x)$ holds for at least two values of y, while $y = \#(\exists x\colon p(x)) \equiv y = \#\top$ holds for only one value of y. □

Next, we investigate the expressive power of tuple numbering. Recall that pure Datalog can only express polynomial algorithms (and is, in fact, PTIME-complete). As it turns out, adding tuple numbering has a rather dramatic impact on its expressiveness:

Theorem 4.4. *Tuple numbering makes Datalog (without primitive types, equality or negation) Turing complete.*

Proof. Figure 6 shows how to implement SK combinators in QL with algebraic data types. The implementation is parameterised over a binary predicate `initial(l, r)` that encodes any input term `l r` (that is, the application of term `l` to term `r`) that we want to reduce

Type `Term` represents combinator terms, including the combinators S and K themselves, as well as a judiciously chosen set of applicative terms `l r` that is just large enough to include the input term and all its reducts (remember that we cannot just include all applicative terms in `Term`, as that would make it infinite). Predicate `red` implements one-step reduction, while `eval` is multi-step reduction of a term to its normal form, if it exists.

The precise statement of Turing completeness thus is: given an SK combinator term `l r`, we can construct a QL program using algebraic data types (and hence a Datalog program using tuple numbering) such that reduction of `l r` terminates at some normal form `n` if and only if the iterative bottom-up evaluation of the QL program terminates with (an encoding of) the same normal form.

```
newtype Term = S() or K() or App(Term l, Term r) {
       initial(l, r) // inject input terms
    or exists(Term x, Term y, Term z | exists(App(App(App(S(), x), y), z) | // if 'S x y z' is a term ...
         l = x and r = z                                            // ... then so is 'x z' ...
      or l = y and r = z                                            // ... and 'y z' ...
      or l = App(x, z) and r = App(y, z)))                          // ... and 'x z (y z)'
  or exists(Term lprev | exists(App(lprev, r)) | l = red(lprev)) // congruence closure on the left
  or exists(Term rprev | exists(App(l, rprev)) | r = red(rprev)) // congruence closure on the right
}

Term red(Term t) {
     exists(Term x, Term y | t = App(App(K(), x), y) and result = y)
  or exists(Term x, Term y, Term z | t = App(App(App(S(), x), y), z) and result = App(App(x, z), App(y, z)))
  or exists(Term l, Term r | t = App(l, r) and (result = App(red(l), r) or result = App(l, red(r)))))
}

Term eval(Term t) { result = eval(red(t)) or (not exists(red(t)) and result = t)}
```

Fig. 6. SK combinators in QL with algebraic data types; the predicate `initial` encodes the term to reduce

As an example, assume we want to reduce the term $K\ S\ K$. We encode it by providing an appropriate implementation of `initial` (the first disjunct guarantees the existence of the terms used in the second disjunct):

```
predicate initial(Term l, Term r) {
  l = K() and r = S() or
  l = App(K(), S()) and r = K()
}
```

Now we can compute `eval(App(K(), S()), K())`, which yields the result `S()`, as expected.

Note that our implementation does not use primitive types. While it uses equality in a few places, most of these equalities are syntactic and disappear when translating to Datalog, except for the equality **`result`** `= t` in `eval`. However, it is easy to define a predicate `equals(Term s, Term t)` that computes equality of terms, so we can eliminate this equality as well. Finally, the single use of negation can also be eliminated by implementing a predicate `nf(Term t)` that holds for exactly those terms `t` that are in normal form, which can be done without using equality or negation. □

5 IMPLEMENTATION

In this section, we briefly describe how we have extended the Semmle runtime system to support tuple numbering.

The theoretically cleanest way of implementing tuple numberings would be as Gödel numberings. Hash functions could be used as a pragmatic alternative, but sufficiently strong hashes are too long for the engine to operate on them directly; for example, a SHA-1 hash needs 160 bits, while the Semmle engine expects primitive values to be 32-bit or 64-bit quantities.

For strings, this problem is solved by maintaining a *string pool* that maps strings to unique 32-bit identifiers. We follow the same strategy for tuple numberings, allocating *tuple pools* that map tuples of values to 32-bit identifiers. To avoid exhausting the space of available identifiers, we use one tuple pool per *universe signature*, where the universe signature of a relation is the ordered list of universes its parameters belong to. Thus, when tuple numbering two relations $p(x, y)$ and $q(x', y')$ we will use the same tuple pool if x is from the same universe as x' and y from the same universe as y'. This overlap is not observable by universe-consistent QL programs.

Non-recursive tuple numbering can be implemented much more easily by computing the relation to be numbered in full, sorting its tuples, and then assigning tuple identifiers in order. This does not work for the recursive case where tuple elements might themselves be from the relation to be numbered or from another relation that depends on it, so identifiers have to be assigned on the fly as tuples are added to the relation.

6 CASE STUDIES

To demonstrate the usefulness of our proposed algebraic data types, we now present three case studies that put them to work on three practical analysis problems: context-sensitive flow analysis for JavaScript using the Cartesian Product Algorithm as an example of a classic program analysis algorithm; control flow graph construction for Java as an example of a supporting algorithm; and regular expression parsing as a somewhat unconventional application.

For space reasons, we only describe the most salient parts of each case study in detail; links to full implementations are provided on our website (Semmle 2017b).

6.1 Implementing the Cartesian Product Algorithm

A typical use case for structured values in Datalog is the implementation of context-sensitive flow analyses, a particularly interesting example of which is the Cartesian Product Algorithm (CPA) (Agesen 1995). CPA contexts are tuples of abstract values representing the arguments passed at some call site. To analyse a call $f(e_1, \ldots, e_n)$, we (i) analyse the argument expressions $e_1, \ldots, e_n$ yielding abstract values $v_1, \ldots, v_n$; (ii) analyse the body of f in the context $(v_1, \ldots, v_n)$, which means that we assume each parameter x_i to have the corresponding abstract value v_i; and (iii) use the abstract return value of f in this context as the abstract value of the call. As the analysis proceeds, more possible abstract argument values v_i may be discovered, which may induce more possible contexts for f, possibly yielding new abstract return values. These changes are monotonic, so the analysis will terminate.

We outline an implementation of CPA for JavaScript in QL, concentrating on the handling of contexts and omitting most of the rules for handling individual language constructs, which are not interesting for our purposes.

At the heart of the analysis is a QL predicate `eval` that maps pairs of a `Context` and an `Expr` (that is, a JavaScript expression) to one or more abstract values, represented by the abstract data type `AbstractValue`, which is a straightforward enumeration of the various kinds of values tracked by the analysis:

```
newtype AbstractValue = Undefined() or Number() or AbstractFunction(Function f) or ...
```

`Undefined` is the abstract value representing the JavaScript `undefined` value; `Number` represents all numeric values; while `AbstractFunction` defines one abstract value for each function `f`, representing all concrete function objects created by evaluating `f`. The remaining branches are similar and have been omitted for brevity.

A context is a tuple of abstract values, which we represent as a cons-list:

```
newtype Context = Nil() or Cons(AbstractValue car, Context cdr) { evalStep(_, _, _, _, car, cdr) }
```

The branch body of `Cons` restricts it to only construct lists that correspond to arguments for an actually observed call site, using the predicate `evalStep` which we will discuss next. If `Cons` were left unrestricted, the set of contexts would become infinite, and the analysis would never terminate.

Predicate `evalStep(ctxt, c, f, i, car, cdr)` models one step in the left-to-right evaluation of the arguments of call site `c`; it holds if `c` in context `ctxt` may call function `f`, its `i`th argument evaluates to `car`, and the remaining arguments to `cdr`. It is implemented by mutual recursion with another predicate `evalArgs` that iteratively builds up the context corresponding to some call site `c` under a context `ctxt`, and relies on an auxiliary predicate `calls` that models the call graph:

```
predicate evalStep(Context ctxt, CallExpr c, Function f, int i, AbstractValue car, Context cdr) {
  car = eval(ctxt, c.getArgument(i)) and cdr = evalArgs(ctxt, c, f, i+1)
}
```

```
Context evalArgs(Context ctxt, CallExpr c, Function f, int i) {
    calls(ctxt, c, f) and i = c.getNumArgument() and result = Nil()
  or exists(AbstractValue car, Context cdr | evalStep(ctxt, c, f, i, car, cdr) and result = Cons(car, cdr))
}
```

Next, we define a helper predicate `appliesTo` that determines whether a context applies to a function, and a predicate `lookup` that looks up the abstract value corresponding to some parameter `p` in a context, using another auxiliary predicate `get(ctxt, i)` that retrieves the `i`th element of context `ctxt`:

```
predicate appliesTo(Context ctxt, Function f) { ctxt = evalArgs(_, _, f, 0) }

AbstractValue lookup(Context ctxt, Parameter p) {
  exists(Function f, int i | appliesTo(ctxt, f) and p = f.getParameter(i) and result = get(ctxt, i))
}
```

With these preparations out of the way, we now show three representative clauses of the `eval` predicate: analysis of numeric literals as an example of a simple base case (where we use `appliesTo` to restrict the range of the otherwise unused parameter `ctxt`), and the two crucial cases of parameter and return value passing. To analyse a use of a parameter `p` we look it up in the context; to analyse a function call, we construct the appropriate calling context, and use the auxiliary predicate `retval` to determine the possible return values of the callee in that context. That predicate, in turn, considers the `return` statements in the function to determine possible return values:

```
AbstractValue eval(Context ctxt, Expr e) {
    e instanceof NumberLiteral and appliesTo(ctxt, e.getEnclosingFunction()) and result = Number()
  or exists(SimpleParameter p | e = p.getVariable().getAnAccess() and result = ctxt.lookup(p))
  or exists(Function f | calls(ctxt, e, f) and result = retval(evalArgs(ctxt, e, f, 0), f))
  or ...
}

AbstractValue retval(Context ctxt, Function f) {
  exists(ReturnStmt ret | ret = f.getAReturnStmt() and result = eval(ctxt, ret.getExpr()))
}
```

Extending these predicates to model all of ECMAScript 2016 is a non-trivial task, but takes no more than about 500 lines of QL. We want to emphasise, however, that we do not mean to claim that CPA is a silver bullet for the analysis of JavaScript, the challenges of which are manifold and extensively documented in the literature (Jensen et al. 2009; Kashyap et al. 2014; Park and Ryu 2015; Schäfer et al. 2013; Sridharan et al. 2012), we simply use it as an example of a non-trivial context sensitivity policy that nicely demonstrates the use of recursive algebraic data types: since contexts can be lists of arbitrary length, it is not clear how they could be represented in plain QL.

6.2 Constructing control flow graphs

As our next example, we show how to construct intra-procedural control flow graphs for programs written in a very small subset of Java, comprising just four kinds of statements: expression statements, `throw`, `try` with `catch` (but no `finally`) and block statements. We ignore any control flow resulting from expression evaluation.

The CFG contains one node for each statement, plus one entry node and one exit node for each callable (that is, method or constructor):

```
newtype CfgNode = StmtNode(Stmt s) or EntryNode(Callable c) or ExitNode(Callable c)
```

CFG edges are labelled with *completions* that indicate the reason for the flow. We have two kinds of completions: the normal completion indicating normal termination of a statement; and throw completions, indicating that a statement has thrown an exception:

```
predicate final(Stmt s, CfgNode f, Completion c) {
     s instanceof ThrowStmt and f = StmtNode(s) and c = Throw(s.(ThrowStmt).getExpr().getType())
  or exists(Block b | b = s | final(b.getLastStmt(), f, c) or final(b.getAStmt(), f, c) and c != Normal())
  or exists(TryStmt ts | ts = s |
       final(ts.getBlock(), f, c) and
       not exists(RefType thr | c = Throw(thr) | thr.getASupertype*() = ts.getACatchClause().getACaughtType())
    or final(ts.getACatchClause(), f, c)
  )
}

predicate succ(CfgNode s, CfgNode t) {
     exists(Callable c |
       s = EntryNode(c) and t = initial(c.getBody()) or
       final(c.getBody(), s, _) and t = ExitNode(c)
     )
  or exists(Block blk, int i | final(blk.getStmt(i), s, Normal()) and t = initial(blk.getStmt(i+1))
  or exists(TryStmt ts, RefType thr, CatchClause cc |
       final(ts.getBlock(), s, Throw(thr)) and cc = ts.getACatchClause() and
       exists(cc.getACaughtType().commonSubtype(thr)) and t = initial(cc.getBlock())
     )
}
```

Fig. 7. Excerpts from a library for computing control flow graphs for Java.

```
newtype Completion = Normal() or Throw(RefType t) {t.getASupertype*().hasQualifiedName("java.lang","Throwable")}
```

Note that we use the branch body of `Throw` to restrict its parameter `t` to legal exception types. In full Java, we additionally need break and continue completions (optionally including labels), and a return completion.

The CFG is now computed bottom-up, constructing partial CFGs for subtrees of the AST and then gradually combining them into larger CFGs. The partial CFG for a subtree *t* has a unique initial node and one or more final nodes, each associated with a completion.

The initial node `initial(s)` of a statement `s` is simply the corresponding CFG node `StmtNode(s)`. Final nodes are computed by a predicate `final(s, f, c)`, parts of which are shown in Figure 7, that holds if `f` is a final node of the subtree `s` when terminating with completion `c`. In particular:

- Expression statements and `throw` statements are their own final nodes, the former having a normal completion, the latter the `Throw` completion of its thrown exception.
- For blocks, any final node of the last statement is a final node of the block, as are the final nodes of other statements with throw completions; this models **throw** breaking out of a block.
- For `try` statements, any final node of a `catch` block is a final node of the `try`, as is any final node of its body, except if it throws an exception that is a subtype of the type declared by a `catch` clause (and hence guaranteed to be caught).

Finally, Figure 7 also shows parts of the implementation of predicate `succ`, which matches up final nodes with initial nodes to compute the CFG successor relation itself:

- The successor of an entry node is the initial node of the callable's body; the successor of any final node of the body (regardless of its completion) is the exit node.
- For blocks, the successor of a final node of the *i*-th statement with a normal completion is the initial node of the $(i + 1)$-th statement.

```
alt    ::=  cat "|" alt | cat          cat    ::=  term cat | term
term   ::=  atom "*" | atom            atom   ::=  plainchar | "("alt")"
```

Fig. 8. A context-free grammar for a simple regular expression language

- For `try` statements, the successor of a final node in the body that may throw an exception is the initial node of any catch clause that may catch this exception at runtime.

This approach generalises to full languages: we have implemented CFG construction for all of Java 8 (500 LoC) and C# 6 (800 LoC) in QL, modelling not only statement-level control flow as shown here, but also expression-level flow. For Java, we use AST nodes as CFG nodes, while the C# library has proper CFG nodes as shown here. Both libraries scale to real-world code bases with millions of lines of code and are in production use.

Unlike the previous example, this approach to CFG construction could be implemented without algebraic data types: an early version of the Java CFG library did so by encoding completions as integers and overloading AST nodes to serve as CFG nodes. We found, however, that switching to algebraic data types made the code much easier to understand and maintain.

6.3 Parsing regular expressions

Our last example is a parser for regular expressions that produces an AST representation which can be used as for writing analyses. Regular expressions are ubiquitous in modern programming languages and can be a source of bugs, ranging from simple logical errors such as using a start-of-input assertion "`^`" at a position where it cannot possibly match to more complex problems such as regular expressions that are prone to exponential backtracking, which can leave an application vulnerable to ReDoS attacks (Kirrage et al. 2013).

Admittedly, parsing regular expressions is not usually thought of as an analysis task, and certainly not a problem to be solved with Datalog. In languages with built-in regular expression literals such as JavaScript or Perl, the extractor can easily parse them and store an AST representation in the database. In other languages, however, regular expression support is provided by a library, so the extractor cannot easily know which string literals should be interpreted as regular expressions. In fact, in a dynamically typed language such as Python it may take non-trivial points-to analysis to detect calls to the regular expression library in the first place.

With algebraic data types, we can implement the parser in QL instead, with its input coming either from a database extensional or some ancillary analysis; in fact, the parser could even be recursive with the analysis computing its input, if desired. In what follows, we assume that `RegExp` is a suitably defined QL class comprising all strings that may represent regular expressions.

As with the other examples, we only discuss a small part of the implementation in detail and refer to our website (Semmle 2017b) for the full version. We restrict ourselves to those regular expressions described by the context-free grammar in Figure 8, comprising alternation, concatenation, Kleene star and grouping. The terminal symbol `plainchar` represents any character other than the operators "`|`", "`*`", "`(`" and "`)`"; each `plainchar` represents itself, except for the anchors "`^`" and "`$`" which represent start and end of input, respectively. As a matter of terminology, we will call a rule whose right hand side consists of a single non-terminal, such as alt ::= cat, a *chain rule*, and a more complex rule a *production rule*.

Implementing a recogniser, that is, a parser that does not produce any output, is straightforward: for each non-terminal n with rules $r_1, \ldots, r_m$, define ternary predicates $r_i(s, b, e)$ corresponding to the rules and another ternary predicate $n(s, b, e)$ corresponding to the non-terminal itself, which is simply the disjunction of the rule predicates. Each rule predicate $r_i(s, b, e)$ is defined in such a way that it holds if the string s, which is the input to be parsed, contains a substring from index b (inclusive) to index e (exclusive) that can be derived using rule r_i.

For example, the rule alt ::= cat "`|`" alt, is implemented as a Datalog predicate `dis`:

```
predicate dis(RegExp s, int b, int e) { exists(int m | cat(s, b, m) and s.charAt(m) = "|" and alt(s, m+1, e)) }
```

This rule forms one disjunct of predicate `alt`, which represents the non-terminal of the same name:

```
predicate alt(RegExp s, int b, int e) { dis(s, b, e) or cat(s, b, e) }
```

Predicates for the other non-terminals can be defined similarly, yielding a recogniser that checks whether a regular expression can be parsed according to our grammar. However, it does not reify the results of the parse in any useful way, and hence is of limited use for further analysis.

We convert it to a proper parser in five steps:

(1) Turn the predicates for production rules into branches of a new data type `Pattern` like this:

```
newtype Pattern =
  MkDis(RegExp s, int b, int e) {exists(int m | cat(s, b, m) and s.charAt(m)="|" and alt(s, m+1, e))} or ...
```

(2) Introduce a corresponding (concrete) QL class with a predicate `at` for accessing `s`, `b` and `e`:

```
class Dis extends MkDis { predicate at(RegExp s, int b, int e) { this = MkDis(s, b, e) } }
```

(3) For each non-terminal, introduce an abstract class that declares the `at` predicate and is extended both by the concrete classes corresponding to its production rules and the abstract classes corresponding to its chain rules. In our example, we introduce abstract classes `Alt` and `Cat` such that `Cat` and `Dis` both extend `Alt`, making `Alt` the union of `Cat` and `Dis`.

(4) Each call to a non-terminal *n* in a production rule predicate is replaced by a call to *x*.`at`, where *x* is an existentially quantified variable of type *n*:

```
MkDis(RegExp s, int b, int e) {
  exists(Cat l, int m, Alt r | l.at(s, b, m) and s.charAt(m) = "|" and r.at(s, m+1, e))
}
```

Note how the declared types of `l` and `r` enforce the correct precedence and associativity: `l` is restricted to be an element of `Cat` and not, say, of `Dis`, ensuring that alternation is given lower precedence than concatenation, and associates to the right.

(5) Promote the existentially quantified variables thus introduced to parameters of the branch predicate and define getters for them on the QL class, turning it into a full-fledged AST class:

```
newtype Pattern =
    MkDis(RegExp s, int b, int e, Cat l, Alt r) {
      exists(int m | l.at(s, b, m) and s.charAt(m) = "|" and r.at(s, m+1, e))
    } or ...

class Dis extends MkDis, Alt {
  predicate at(RegExp s, int b, int e) { this = MkDis(s, b, e, _, _) }
  Cat getLeft() { this = MkDis(_, _, _, result, _) }
  Alt getRight() { this = MkDis(_, _, _, _, result) }
}
```

Apart from the classes introduced in this way, we can define additional classes to model semantic properties; for instance, anchors are different from other atoms, so we might want to subclass the class `CharAtom` (which we assume has a member predicate `getText` to extract its source text) as follows:

```
abstract class Anchor extends CharAtom {}
class Caret extends Anchor { Caret() { getText() = "^" } }
class Dollar extends Anchor { Caret() { getText() = "$" } }
```

This refinement will be useful in the example analysis we discuss next: assume that we want to find caret assertions which are preceded by at least one term that cannot match the empty string, and hence are unmatchable.

Starting with the latter condition, we define a member predicate `isNullable` that holds for those `Patterns` that can match the empty string. For example, a disjunction is nullable if either of its children is, so `Dis.isNullable` is defined as `getLeft().isNullable() or getRight().isNullable()`.

Character atoms are not nullable, except for anchors. This is elegantly expressed by overriding `isNullable` once in `CharAtom` with body `none()` to model the fact that most character atoms are not nullable, and then again in `Anchor` with body `any()` to model the fact that anchors are an exception.

Now we define a recursive predicate `pred(l, r)` that holds if `l` is matched immediately before `r`:

```
predicate pred(Pattern l, Pattern r) {
  exists(Seq s | l = s.getLeft() and r = s.getRight()) or
  exists(Dis d | pred(l, d) and (r = d.getLeft() or r = d.getRight())) or
  exists(Group g | pred(l, g) and r = g.getBody())
}
```

The unmatchable caret assertions can now be identified by looking for `Caret` nodes that are transitively preceded by a non-nullable pattern (note that in QL `p+` denotes the transitive closure of predicate `p`):

```
from Caret c, Pattern p
where pred+(p, c) and not p.isNullable()
select c, "This assertion can never match."
```

As it stands, our parser is quite inefficient, since it effectively uses a brute-force bottom-up approach that wastes a lot of time building partial ASTs that are not part of a successful parse. To improve performance, the `b` and `e` parameters of the rule predicates have to be restricted to candidates where a successful parse is possible. Kanazawa (2007) observed that this can be achieved by applying the well-known *magic sets* transformation (Abiteboul et al. 1995) to push calling contexts into predicates. This transforms what is essentially a CYK parser into an Earley parser.

Based on these techniques, the miniature parser outlined above can be extended to a full-fledged parser for JavaScript regular expressions, totalling about 600 LoC.

7 RELATED WORK

As mentioned in the introduction, LogiQL's constructor predicates (LogicBlox 2017) are closely related to our algebraic data types. Essentially, they provide a non-recursive variant of tuple numbering. Multiple constructor predicates can contribute values to the same type, so there is no need for a two-stage encoding like the one we have presented. Constructor predicates are heavily used by Doop (Bravenboer and Smaragdakis 2009), a points-to analysis framework for Java implemented in LogiQL. The absence of recursion, however, makes it impossible to encode some of the more complex examples of algebraic data types shown in Section 6; in particular, Doop does not support CPA, and we conjecture that it would need to make use of other extra-logical language features of LogiQL in order to implement it.

Tuple numbering can be viewed as an extension of Datalog with *existential rules*, where the head of the rule may existentially quantify some of its variables (as opposed to normal rules, where each variable in the head has to appear in the body at least once). Such rules were first studied in the database community to express tuple-generating dependencies (Abiteboul et al. 1995), a very general class of integrity constraints on extensional databases that allow asserting the existence of database entities based on logical conditions. For example, in a database that encodes the AST of a Java program we might want to assert that for every entity representing a `if` statement there is an entity representing its "then" branch. Tuple-generating dependencies have been the subject of much theoretical investigation, chiefly concerned with problems such as repairing databases that fail to satisfy a dependency, or optimising query execution based on known constraints. More recently, existential rules have been studied in ontology-based reasoning (Baget et al. 2011; Calì et al. 2009). Besides a difference in focus, the key difference between tuple numbering and more general existential rules is that the latter do not require the existentially quantified variables to be instantiated with freshly constructed values. As such, they are more general, but not suited for representing structured values.

Algebraic data types, first introduced in Hope (Burstall et al. 1980), have established themselves as an essential feature of many functional programming languages, notably ML (Milner et al. 1997) and Haskell (Hudak et al. 2007). The OCaml dialect of ML (Doligez et al. 2016) additionally supports standard object-oriented features, which, however, are orthogonal to algebraic data types. Scala (Odersky and Zenger 2005) also has both object orientation and algebraic data types and uses the former to express the latter, viewing algebraic data types as classes and expressing pattern matching through virtual dispatch similar to what we have shown in Section 3. In QL, the two concepts are independent, but the great flexibility of classes makes it easy to combine them in fruitful ways, as we have shown in our examples.

We have mentioned above that our algebraic data types do not support polymorphism, which is offered by both Haskell and ML. Haskell supports further extensions like higher-kinded type parameters and generalised algebraic data types that bring it a step closer to the very powerful inductive data types supported by theorem provers such as Coq (Bertot and Castéran 2004) or Agda (Norell 2008). These data types offer not only a very general notion of polymorphism, but also dependent types, which gives them much of the flexibility that QL's branch bodies provide, but with stronger static guarantees. Agda in particular has pioneered the practical use of induction-recursion (Dybjer 2000), whereby the definition of an inductive data type uses a recursively defined function over that same data type. In QL, such recursion between data types and other predicates is a natural consequence of viewing types as unary predicates.

We are not aware of any previous work on adding algebraic data types to Datalog, but several other logic programming languages such as SWI Prolog (Wielemaker et al. 2012) and hybrid functional-logic languages such as Mercury (Somogyi et al. 1994) and Datafun (Arntzenius and Krishnaswami 2016) do have support for them. Their notion of types, however, embodies the more usual notion of types as meta-level descriptions of sets, rather than QL's view of types as unary predicates defined within the language itself. Datafun uses types to track monotonicity and establish finiteness of predicates, so programs can freely construct and use structured values as long as they can be typed, thereby proving that only a finite number of values is constructed in any given execution. SWI Prolog and Mercury, on the other hand, follow a Prolog-style semantics, which supports arbitrary structured values.

As mentioned above, QL's type system (Schäfer and de Moor 2010) can accommodate algebraic data types without any changes, since it supports arbitrary types defined by unary predicates as long as their mutual relationship can be described in terms of first-order statements such as inclusion and disjointness, which is the case for our monomorphic algebraic data types. LogiQL's type system (Zook et al. 2009) also supports inclusion constraints between types, but not disjointness, which is important for algebraic data types.

8 CONCLUSION

We have presented an extension of QL with monomorphic algebraic data types that allow programs to work with structured values, which was previously impossible. Like their counterparts in functional languages, these types offer a flexible way of describing tree-structured data using disjoint union and tupling, and they can be recursive. Like other types in QL, algebraic data types are just unary predicates, so they can be recursive not just with each other but with other predicates as well. While the new types are orthogonal to QL's existing class system, the two can be mixed freely, effortlessly combining object-oriented and algebraic programming idioms.

Algebraic data types can be implemented by compiling QL to plain Datalog extended with a tuple numbering operator that manages the creation of fresh identifiers for structured values. This operator adds considerable expressive power to Datalog, yet is easy to implement and interacts well with common optimisations.

Algebraic data types bring the simplicity and elegance of QL to bear on problems that previously were out of its scope, as shown by the case studies. In future, we are particularly interested in further exploring its applications in context-sensitive analyses like CPA and analysis support for DSLs like regular expressions.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman, Boston, MA, USA.

Ole Agesen. 1995. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *ECOOP*.

Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*.

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: a Functional Datalog. In *ICFP*.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *ECOOP*.

Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. 2011. On Rules with Existential Variables: Walking the Decidability Line. *Artificial Intelligence* 175, 9-10 (2011).

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development.* Springer, Berlin Heidelberg.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA*.

R. M. Burstall, D. B. MacQueen, and D. T. Sannella. 1980. HOPE: An Experimental Applicative Language. In *LFP*.

Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009. A General Datalog-Based Framework for Tractable Query Answering over Ontologies. In *PODS*.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *TOPLAS* 13, 4 (Oct. 1991).

Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2016. The OCaml System: Documentation and User's Manual. (2016). http://caml.inria.fr/pub/docs/manual-ocaml/

Peter Dybjer. 2000. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *Journal of Symbolic Logic* 65, 2 (2000).

Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. 2012. An overview of Ciao and its design philosophy. *TPLP* 12, 1-2 (2012).

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *HOPL*.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *SAS*.

Makoto Kanazawa. 2007. Parsing and Generation as Datalog Queries. In *ACL*.

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *FSE*.

James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static Analysis for Regular Expression Denial-of-Service Attacks. In *NSS*.

Ondrej Lhoták and Laurie J. Hendren. 2004. Jedd: A BDD-based relational extension of Java. In *PLDI*.

LogicBlox. 2017. LogicBlox 4 Reference Manual. (2017). https://developer.logicblox.com/content/docs4/core-reference/webhelp/

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *AFP*.

Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. In *OOPSLA*.

Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *ECOOP*.

Max Schäfer and Oege de Moor. 2010. Type Inference for Datalog with Complex Type Hierarchies. In *POPL*.

Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic Determinacy Analysis. In *PLDI*.

Semmle. 2017a. Code Exploration. (2017). https://semmle.com/products/semmle-ql

Semmle. 2017b. Publications Page. (2017). https://semmle.com/publications

Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. 1994. The Implementation of Mercury, an Efficient Purely Declarative Logic Programming Language. In *ILPS*.

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*.

Terrance Swift and David S. Warren. 2012. XSB: Extending Prolog with Tabled Logic Programming. *TPLP* 12, 1-2 (2012).

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *APLAS*.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012).

Niklaus Wirth. 1976. *Algorithms + Data Structures = Programs.* Prentice Hall, Upper Saddle River, NJ, USA.

David Zook, Emir Pasalic, and Beata Sarna-Starosta. 2009. Typed Datalog. In *PADL*.

A COREQL

To make our presentation self-contained, we reproduce the definition of the syntax and semantics of CoreQL from Avgustinov et al. (2016): Figure 9 gives the syntax, Figure 10 and Figure 11 the translation from CoreQL to Datalog.

The following definitions establish notation used in the figures:

Definition A.1 (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.

Definition A.2 (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 A.3 (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 A.4 (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:

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

Definition A.5 (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.

Definition A.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 $\lambda(D, p, |\overline{x}|) \neq \emptyset$.

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

Fig. 9. Syntax of Core QL; $\bar{\cdot}$ denotes (possibly empty) sequences, $\cdot^{?}$ optional elements

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

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

$$\begin{array}{lcll}
\mathcal{K}(cd) & := & \bigvee_{D<:C} D(\textbf{this}) & \text{if } cd \text{ is abstract}\\
\mathcal{K}(cd) & := & C.C(\textbf{this}) & \text{if } cd \text{ is concrete}
\end{array}$$

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

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

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

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

Fig. 10. Translation from Core QL predicates to Datalog; for readability, we write $C <: T$ to mean $C.\mathtt{domain} <: T$

Translation of a predicate or character body f:

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

Translation of a predicate call:

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

Translation of other formulas:

$$\begin{aligned}
\mathcal{T}_f(\mathbf{not}\ f, \Gamma) &:= \neg\mathcal{T}_f(f, \Gamma) \\
\mathcal{T}_f(f\ \mathbf{and}\ g, \Gamma) &:= \mathcal{T}_f(f, \Gamma) \wedge \mathcal{T}_f(g, \Gamma) \\
\mathcal{T}_f(f\ \mathbf{or}\ g, \Gamma) &:= \mathcal{T}_f(f, \Gamma) \vee \mathcal{T}_f(g, \Gamma) \\
\mathcal{T}_f(\mathbf{exists}(C\ x \mid f), \Gamma) &:= \exists x\colon \big(C(x) \wedge \mathcal{T}_f(f, \Gamma[x := C])\big)
\end{aligned}$$

Fig. 11. Translation from Core QL formulas to Datalog