

Logical Data Modeling: Part 1

Terry Halpin
INTI International University

My previous fifteen articles on ontology-based approaches to data modeling focused on popular ontology languages for the Semantic Web, such as the Resource Description Framework (RDF), RDF Schema (RDFS), and the Web Ontology Language (OWL). While these languages may be used to specify a data model or ontology, other languages are needed to query such RDF-based data models, e.g. SPARQL (a reflective acronym for “SPARQL Protocol and RDF Query Language”) and OWL-QL. In some future articles, I might discuss some of these semantic web query languages. However, this article instead starts a new series on an alternative, logic-based approach to business data and rules using a single language to both create and query data models.

Introduction

Because of their formal foundation on description logics, semantic web languages such as OWL may be used to perform logical inferences, thus enabling some facts to be derived from other facts. To cater for the possibility that the data for the web fact structures of interest might be incomplete, these languages adopt the *open world assumption* (OWA). This entails that if a given proposition of interest (e.g. `:Phobos :orbits :Mars`) is neither asserted nor inferable from other facts, its truth value taken to be unknown, rather than false.

Logic programming languages are also formally based on logic, and have very strong inferencing capabilities, but unlike OWL, they adopt the *closed world assumption* (CWA), so assume that all the relevant facts are known. Hence a proposition of interest is assumed to be false unless it is either asserted or derivable from other facts. This CWA approach is also typically used when querying relational databases. The two most popular logic programming languages are Prolog and datalog. Prolog (from “*Programming in Logic*”) was originally developed by Alain Colmerauer and others in the early 1970s. Although largely declarative, Prolog includes some non-declarative constructs (e.g. the cut operator), and its programs are not guaranteed to terminate (i.e. execute in a finite time). In spite of these issues, Prolog can be used very productively in applications requiring substantial inferencing power. For a readable overview of prolog systems, see <http://en.wikipedia.org/wiki/Prolog>.

The term “datalog”, coined by David Maier to combine “data” and “logic”, is appropriate for an executable, logic-based language designed for modeling and querying databases. Both Prolog and datalog enable recursive rules and queries (e.g. list all one’s ancestors) to be simply and elegantly expressed, and efficiently executed. Unlike Prolog however, datalog is purely declarative. Moreover, its syntax conforms to safety rules that guarantee that any syntactically well formed datalog program will terminate. For a classical, technical reference on datalog see [1], for a recent tutorial on extended datalog see [5], and for a brief overview of datalog systems see <http://en.wikipedia.org/wiki/Datalog>.

Of the many datalog and datalog-based systems used in practice, *LogiQL* (*Logical Query Language*, pronounced “logical”) is a good example of the state of the art, with a successful track record in industrial business optimization, especially for predictive and control analytics involving large data sets. LogiQL extends traditional datalog in several ways, supporting blocks for modularity, many built-in functions (including aggregate functions), and other advanced features. While retaining much of datalog’s traditional notation, LogiQL provides additional syntax to distinguish between constraints and derivation rules, and to simplify the formulation of various aspects (e.g. declaring a predicate to be functional). Although a commercial version of LogiQL is available from LogicBlox, this series of articles focuses on the free, cloud-based version of LogiQL that is accessible at <https://developer.logicblox.com/playground/>.

A Simple Data Model

Figure 1(a) depicts a simple data model in Object-Role Modeling (ORM) [3] notation, including a sample population. Countries are entities that are identified by their country code, and languages are entities identified by their language name. The sample population shows only six countries: Australia (AU), Canada (CA), France (FR), Luxembourg (LU), the United States (US), and the Vatican City State (VA). The unary fact type Country is large is used to record which of those countries are large in size. The binary fact type Country officially uses Language records for each country the languages it uses in official documents. These languages may or may not be an official language of the country (e.g. neither Australia nor the United States has an official language, but both use English in their official documents).

ORM specifies all facts in terms of roles (depicted by boxes) played by objects. All fact types in ORM are set based, so their associated fact tables never duplicate a tuple. A logical predicate is an ordered set of the roles in a single fact type, so combining the object type names (e.g. 'Country', 'Language') with a predicate reading (e.g. 'is large', 'officially uses') provides a fact type reading. The bar over the roles of the fact type Country officially uses Language depicts a spanning uniqueness constraint, indicating that a given country may officially use many languages, and a given language may be officially used by many countries (so the relationship is many-to-many). The large dots on the role connections for that fact type depict mandatory role constraints, indicating that each country officially uses some language, and each language of interest is officially used by some country.

Figure 1(b) depicts the same data model in the Barker notation [2] for Entity Relationship (ER) modeling, but without the sample data. Figure 1(c) depicts the same data model as a class diagram in the Unified Modeling Language (UML) [6], again without the sample data. Finally, Figure 1(d) depicts a populated relational database model for the same example. There are two relational tables, Country and CountryLanguageUse, with their primary keys underlined. The foreign key reference from CountryLanguageUse.countryCode to Country.countryCode is depicted by an arrowed line. The attribute Country.isLarge uses the bit data type, so here 1 denotes True and 0 denotes False.

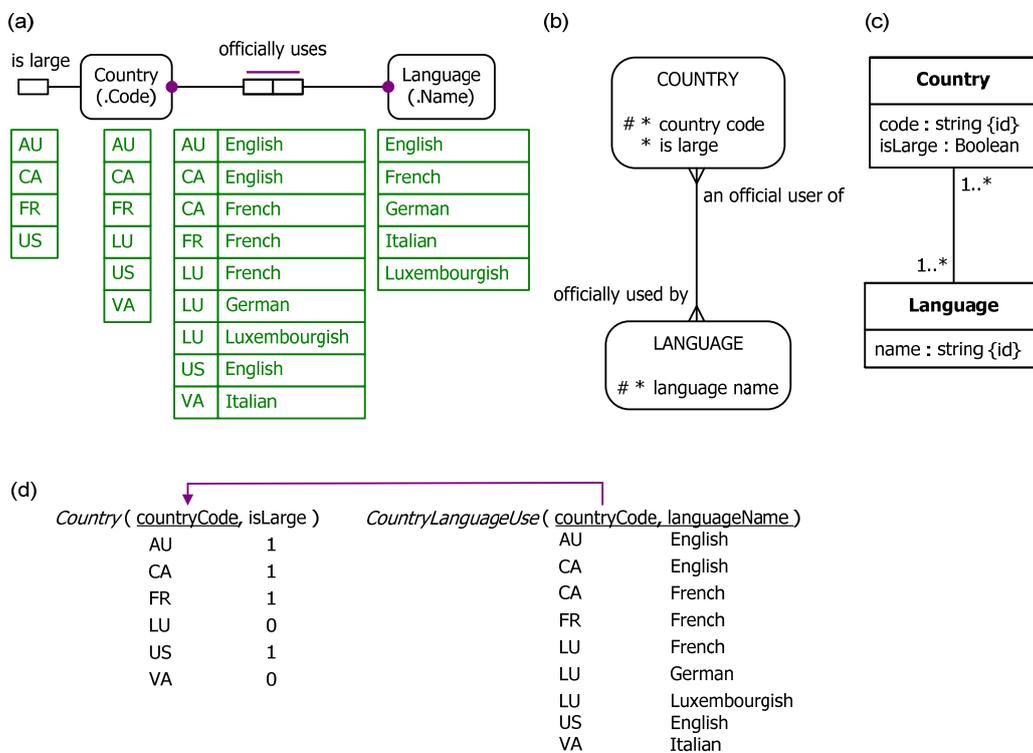


Figure 1 Example data model in (a) ORM, (b) Barker ER, (c) UML, and (d) relational database notation.

The following sections discuss how to code this example in LogiQL, using the free cloud-based “playground” provided at the website mentioned earlier. If you wish to execute LogiQL code yourself on the LogiQL playground, please note that currently the website supports only Chrome and Firefox as web browsers. It is anticipated that other web browsers will be supported in the future.

Declaring the Example Schema in LogiQL

In practice, most entities (e.g. a country or language) can be identified or referenced simply by relating them to a single value (e.g. a country code or language name). The mode or manner in which a value refers to its entity is called a *reference mode* (often abbreviated to *refmode*). In ORM, such simple reference schemes are usually depicted in compact form, by including the reference mode (e.g. Code or Name) in parentheses below the name of the entity type, as shown in Figure 2(a). The dot before the refmode indicates that is a “popular” refmode, ensuring that the compact diagram in Figure 2(a) is equivalent to the expanded diagram shown in Figure 2(b). Other kinds of refmodes exist (e.g. for unit-based reference), but are ignored in this article.

In Figure 2(b) the uniqueness constraint bars over each role of the fact type indicate that they are one-to-one relationships (e.g. each country has at most one country code, and each country code is of at most one country). The large dots on the reference roles hosted by Country and Language indicate that those roles are mandatory (so each country has a country code, and each language has a language name). So the predicates that map Country to CountryCode and Language to LanguageName are *injections* (i.e. mandatory, 1:1 into- relationships). Marking the uniqueness constraints on the roles of CountryCode and LanguageName with a double bar indicates that these injections provide the *preferred reference scheme* for Country and Language.

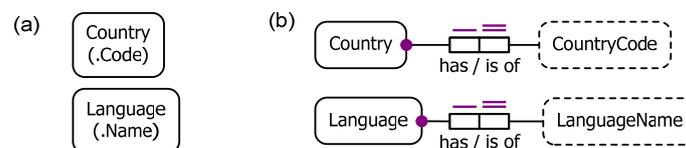


Figure 2 Reference schemes for Country and Language in (a) compact ORM notation and (b) expanded notation.

In LogiQL, the entity type Country as well as its reference schemes may be declared using the following code. The *right arrow* symbol “->”, composed of the two keyboard characters “-” and “>”, stands for the material implication operator “ \rightarrow ” of logic, and is read as “implies”.

```
Country(c), hasCountryCode(c:cc) -> string(cc).
```

Here Country is a unary *predicate* for the entity type Country, string is a unary predicate for the character string datatype, and hasCountryCode is a binary predicate for the reference relationship that relates countries to their country codes (or more correctly, the data values representing those country codes). These predicates are written in prefix notation, with their arguments appended in parentheses. In this case, the arguments c and cc are *individual variables* (which may be assigned some individual item of interest). To save typing, and aid memory, I usually use short names for individual variables based on one or more characters in the name of the variable’s semantic type (e.g. “c” for “Country” and “cc” for CountryCode). Different variables based on the same type may be distinguished by appending a number to their names (e.g. “c1” and “c2” for two country variables). If desired, you may also use longer variable names (e.g. “country” instead of “c”).

The colon “:” in hasCountryCode(c:cc) distinguishes hasCountryCode as a *refmode predicate* (and hence injective), so there is no need to write further code to enforce the mandatory and uniqueness constraints on that predicate. Please note that LogiQL *formulae must always end with a period*. Moreover, LogiQL is *case-sensitive*, so letters entered in lowercase are considered different from letters entered in uppercase. You can choose whatever case you like for any letter in the name of a predicate or variable, but you must consistently use the same casing choice whenever you use that name. For example, if you use “Country” to name an entity type, you cannot reference it later by the name “country”.

As a matter of style, I tend to start *entity predicates* (i.e. unary predicates that denote an entity type) with a capital letter, and start other predicates with a lower case letter. This is consistent with how ORM fact types are declared in English, and it helps me to immediately distinguish entity predicates from other predicates when reading code. You can choose a different style, but be sure to use it consistently.

The formula “Country(c), hasCountryCode(c:cc) -> string(cc).” is treated as an abbreviation for the following four formulae in logic. The first formula declares that Country is an entity type of interest, using “ \forall ” for the *universal quantifier* (read as “for each”). You can read this formula as “For each individual c , if c is a country then c is an entity”.

$$\begin{aligned} &\forall c (\text{Country } c \rightarrow \text{Entity } c) \\ &\forall c, cc [c \text{ hasCountryCode } cc \rightarrow (\text{Country } c \ \& \ \text{string } cc)] \\ &\forall c [\text{Country } c \rightarrow \exists^1 cc \ c \text{ hasCountryCode } cc] \\ &\forall cc \exists^{0..1} c \ c \text{ hasCountryCode } cc \end{aligned}$$

The second logical formula is a *typing constraint* to declare the types of the arguments of the hasCountryCode predicate. LogiQL requires all its predicates to be strongly typed. The third formula is an integrity constraint to ensure that each country has exactly one (i.e. at least one and at most one) country code. The fourth formula constrains each country code to refer to at most one country.

To avoid having to type “ \forall ”, which is unavailable on standard keyboards, LogiQL formulae assume that *variables that occur on both sides of an arrow are implicitly universally quantified*. As you can see, the LogiQL syntax makes it quick and easy to declare an entity type along with an injective reference scheme. However, the brevity of the syntax requires care (e.g. don’t forget to include the colon “:” between the variables of a refmode predicate, and don’t forget to end each formula with a period “.”).

Similarly, the entity type Language can be declared along with its reference scheme as follows. Here I’ve used the letter “l” as an individual variable for a language, and “ln” as a variable for a language name. This brevity saves typing, but if you think “l” might be confused with the numeral for the number one, feel free to choose a different name (e.g. “lang” or even “language”) for the language variable.

```
Language(l), hasLanguageName(l:ln) -> string(ln).
```

The fact types Country is large and Country officially uses Language may be declared in LogiQL as follows.

```
isLarge(c) -> Country(c).
officiallyUses(c, l) -> Country(c), Language(l).
```

In LogiQL, a *comma between two formulae stands for the logical **and**-operator*, often displayed as an ampersand “&” in logic. Hence, these constraints correspond to the following type declarations in logic:

$$\begin{aligned} &\forall c (c \text{ isLarge} \rightarrow \text{Country } c) \\ &\forall c, l [c \text{ officiallyUses } l \rightarrow (\text{Country } c \ \& \ \text{Language } l)] \end{aligned}$$

This means that the predicate named “isLarge” can be applied only to countries, and the predicate named “officiallyUses” can be applied only to country, language pairs. If this is not the case for the business domain being modeled, then you need to choose other, typically longer, names. For example if you wish to record which countries are large as well as which cars are large, then you must distinguish the predicates by naming them differently, e.g. “isLargeCountry” and “isLargeCar”, or “country:isLarge” and “car:isLarge”. In rare cases, you might instead create a common supertype and apply the predicate to that; but for countries and cars it would be very unlikely that we would want to create a supertype of them. At any rate, LogiQL requires *different names for different predicates*.

We have now coded all of the schema in Figure 1(a) in LogiQL, with the exception of the mandatory role constraints that each country officially uses some language, and each language is officially used by some country. Coding of such mandatory role constraints will be discussed in the next article. Ignoring those two constraints, the schema may be declared by just these four lines of LogiQL code. We’ll see how to add the data in the next section.

```
Country(c), hasCountryCode(c:cc) -> string(cc).
Language(l), hasLanguageName(l:ln) -> string(ln).
isLarge(c) -> Country(c).
officiallyUses(c, l) -> Country(c), Language(l).
```

To enter the schema in the free, cloud-based REPL (Read-Eval-Print-Loop) tool available on LogicBlox’s playground website (<https://developer.logicblox.com/playground/>), use a supported browser such as Chrome or Firefox to access the website, then click the “Open in new window” link to give yourself a full screen for entering the code. Schema code is entered in one or more *blocks* of one or more lines of code, using the *addblock* command to enter each block. After the “/>” prompt, type the letter “a”. A drop-down list of available commands starting with the letter “a” now appears. Click the *addblock* option to have the *addblock* command added to the code window. Typing a space character and single quote after the *addblock* command causes a pair of single quotes to be appended, with your cursor placed inside those quotes ready for your block of code (see Figure 3).



Figure 3 Invoking the *addblock* command in the REPL tool.

Now copy the four lines of schema code provided above in this article to the clipboard (e.g. using Ctrl+C), then paste it between the quotes (e.g. using Ctrl+V), and then press the Enter key. You are now notified that the block was successfully added, and a new prompt awaits your next command (see Figure 4). By default, the REPL tool also appends an automatically generated identifier for the code block (e.g. ‘block_1Z1C1DPH’). For simplicity, block identifiers are omitted in this article. Alternatively, you can enter each line of code directly, using a separate *addblock* command for each line.

```

1 Welcome to the LogicBlox playground!
2
/> addblock 'Country(c), hasCountryCode(c:cc) -> string(cc).
.. Language(l), hasLanguageName(l:ln) -> string(ln).
.. isLarge(c) -> Country(c).
..   officiallyUses(c, l) -> Country(c), Language(l).
..
=> Successfully added block
/>

```

Figure 4 Adding a block of code.

Declaring a Derived Fact Type

Now suppose that we are interested in knowing which countries are “multilingual”, in sense of officially using at least two languages. Figure 1(a) shows how to model this requirement in ORM, by adding the *derived fact type* Country is multilingual, along with a *derivation rule* to perform the derivation. In ORM, derived fact types are appended with an asterisk “*”.The ORM derivation rule is specified in FORML (*Fact-Oriented Modeling Language*) [4], a formal textual language for ORM. Here, “<>” (read “is not equal to”) denotes the inequality operator (\neq). Although this derivation rule could have been specified using the count function, I’ve used the more fundamental formulation shown here because discussion of aggregate functions is postponed till a later article.

Figure 1(b) shows how to model this in UML, by adding the derived attribute `isMultilingual` to the `Country` class. In UML, derived attributes are prepended with a slash “/”. In UML, derivation rules are typically specified in the Object Constraint Language (OCL). Since OCL expressions are often difficult for business users to validate no OCL specification is provided here. The Barker ER notation does not support derived attributes or derived relationships. In a relational database, the derivation rule may be specified in a number of ways (e.g. using a view).

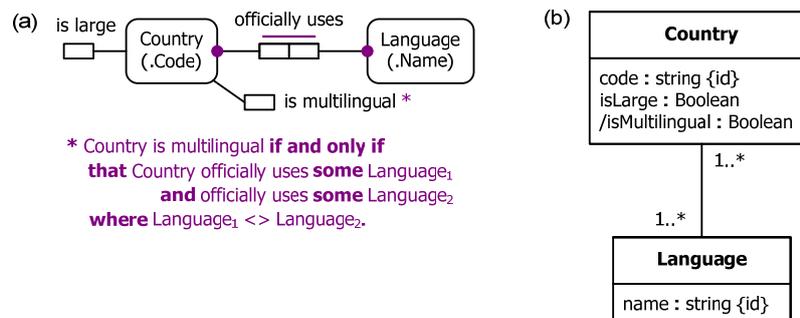


Figure 5 Adding a derived fact type to the sample schema in (a) ORM and (b) UML.

Database systems that use logic-based languages such as Prolog or datalog are called *deductive databases* because they enable even complex deductions to be elegantly specified and efficiently executed. In LogiQL, derivation rules for derived fact type or queries are specified as *left-arrow rules*, using the left arrow symbol “<-” (read as “if”) to denote the inverse material implication operator “←” in logic. Derivation rules are specified as Horn clauses, which take the form *head* <- *body*. As discussed, *variables that occur in both the head and body of the rule are implicitly universally quantified*. Moreover, *variables that occur only in the body of the rule are implicitly existentially quantified*. These requirements are illustrated in the following example.

If you want to reference a derived predicate in a later rule, you need to provide a name for it. For our current example, I’ve used the name “isMultilingual” for the derived predicate corresponding to the derived fact type `Country` is multilingual. The derivation rule in LogiQL may be declared as shown below. The variables *l1* and *l2* are used for languages, and “!=” denotes the inequality operator (≠). Each comma between conditions in the rule body stands for the logical and-operator.

`isMultilingual(c) <- officiallyUses(c, l1), officiallyUses(c, l2), l1 != l2.`

The implicit quantification rules ensure that the head variable *c* is universally quantified, and the variables *l1* and *l2* are existentially quantified because they are introduced in the body. So the LogiQL rule above is equivalent to the following rule in logic, where ∃ denotes the *existential quantifier* (read as “there exists some”). For the sample data, only Canada and Luxembourg satisfy the derived predicate.

$\forall c [c \text{ isMultilingual} \leftarrow \exists l_1, l_2 (c \text{ officiallyUses } l_1 \ \& \ c \text{ officiallyUses } l_2 \ \& \ l_1 \neq l_2)]$

To add the rule in REPL to the existing schema code, simply use an `addblock` command with the above LogiQL code between single quotes after the command. The block structure of LogiQL enables you to build large programs one block at a time.

Adding the Example Data in LogiQL

The data in Figure 1(a) may be entered in LogiQL using the following *delta rules*. A delta rule of the form *+fact* inserts that fact. For example, the delta rule “+isLarge(“AU”).” inserts the fact that Australia is a large country. Because the `isLarge` predicate is known to apply to instances of `Country`, whose reference scheme using country codes is also known, the LogiQL compiler interprets “AU” in this context to be the country

whose country code is represented by the string "AU". Similarly, languages may be represented by their quoted names.

Please note that *plain, double quotes* (i.e. ", ") are needed here. For example, single quotes or smart double quotes (e.g. “, ”) are unacceptable. Various word processors, such as Microsoft Word (which I’m using to write this article) use smart quotes by default. Hence it’s best to use a basic text editor such as WordPad or NotePad to enter code that will later be copied into a LogiQL tool.

```
+isLarge("AU"), +isLarge("CA"), +isLarge("FR"), +isLarge("US").
+officiallyUses("AU", "English"), +officiallyUses("CA", "English").
+officiallyUses("CA", "French"), +officiallyUses("FR", "French").
+officiallyUses("LU", "French"), +officiallyUses("LU", "German").
+officiallyUses("LU", "Luxembourgish").
+officiallyUses("US", "English"), +officiallyUses("VA", "Italian").
```

Here I’ve chosen to enter one or two facts per line of code, ending each line with a period. For entering large amounts of data, there are quicker ways that reduce the typing involved, but this basic approach will suffice for now. Note that there is no need to add delta rules to insert countries and languages because those facts are implied by the typing and mandatory role constraints on the officiallyUses predicate. For example, the assertion +officiallyUses("AU", "English") implies the assertions +Country("AU") and +Language("English"). However, there is no harm in adding those facts separately if you wish.

A delta rule of the form *-fact* is used to retract a fact, and a delta rule of the form *^fact* may be used for an insertion or to update the value component of a functional fact. Those kinds of delta rules will be discussed in a later article.

Delta rules to add or modify data are entered using the exec (for ‘execute’) command rather than the addblock command. To invoke the exec command in the REPL tool, type “e” and then select exec from the drop-down list. Typing a space character and single quote after the exec command causes a pair of single quotes to be appended, with your cursor placed inside those quotes ready for your delta rules.

Now copy the six lines of data code provided above to the clipboard (e.g. using Ctrl+C), then paste it between the quotes (e.g. using Ctrl+V), and then press the Enter key. A new prompt awaits your next command. Both schema and data have now been entered (see Figure 6). Alternatively, you can enter each line of data yourself directly, using a separate exec command for each line.

```

/> addblock 'Country(c), hasCountryCode(c:cc) -> string(cc).
.. Language(l), hasLanguageName(l:ln) -> string(ln).
.. isLarge(c) -> Country(c).
..   officiallyUses(c, l) -> Country(c), Language(l).
.. '
=> Successfully added block
/> addblock 'isMultilingual(c) <- officiallyUses(c, l1), officiallyUses(c, l2), l1 != l2.'
=> Successfully added block
/> exec '+isLarge("AU"), +isLarge("CA"), +isLarge("FR"), +isLarge("US").
.. +officiallyUses("AU", "English"), +officiallyUses("CA", "English").
.. +officiallyUses("CA", "French"), +officiallyUses("FR", "French").
.. +officiallyUses("LU", "French"), +officiallyUses("LU", "German").
.. +officiallyUses("LU", "Luxembourgish").
.. +officiallyUses("US", "English"), +officiallyUses("VA", "Italian").
.. '
/>

```

Figure 6 Adding the data below the program code.

Printing and Querying the Database in LogiQL

Now that the data model (schema plus data) is stored, you can use the *print* command to inspect the contents of any predicate. For example, to list all the recorded countries, type “p” then select print from the drop-down list, and then type a space followed by “C”, then select Country from the drop-down list and press Enter. Alternatively, you can type all of “print Country” yourself and then press Enter. Figure 7

shows the relevant result. By default, the REPL tool also prepends a column listing automatically generated, internal identifiers for the returned values, but for simplicity these identifiers are omitted here.

```

/> print Country
=>


|    |
|----|
| VA |
| US |
| CA |
| AU |
| LU |
| FR |


```

Figure 7 Using the print command to list the extension of a predicate.

To perform a general query, you need to specify a derivation rule to compute the facts requested by the query. If you want to reference a derived predicate in a later rule, you need to provide a name for it, as discussed earlier for the derived fact type Country is multilingual. However, if you simply want to issue a query to derive some facts without needing to reference them later, there is no need to name the derived predicate, so we just use an *anonymous predicate* instead to capture the query result. In LogiQL, an anonymous predicate uses an underscore “_” as a substitute for a normal predicate name. For example, the following rule may be used to derive those large countries that officially use French as a language.

`_ (c) <- isLarge(c) , officiallyUses(c, "French").`

Here the rule’s head `_ (c)` uses an anonymous predicate to capture the result derived from the rule’s body. Since the variable `c` is a head variable, it is implicitly universally quantified. Moreover, the compiler knows that here the language name is intended to refer to its language. So the above rule is understood as shorthand for the following logic formula, where the dummy `isReturned` predicate is satisfied by what is returned by the query, thus standing for an anonymous predicate.

$\forall c [c \text{ isReturned} \leftarrow (c \text{ isLarge} \ \& \ \exists l (c \text{ officiallyUses } l \ \& \ l \text{ hasLanguageName "French"}))]$

In LogiQL, queries are executed by appending their code in single quotes to the `query` command. To do this in the REPL tool, type “q”, choose “query” from the drop-down list, type a space and single quote, then copy and paste the above LogiQL query code between the quotes and press Enter. The relevant query result is now displayed as shown in Figure 8. For simplicity, this figure omits display of internal identifiers for the returned values.

```

/> query '_ (c) <- isLarge(c) , officiallyUses(c, "French").'
=>


|    |
|----|
| CA |
| FR |


```

Figure 8 Using the query command to execute a sample query.

Notice how easy it is to do the equivalent of a relational join between predicates simply by using the same variable (`c`, in this case) in multiple conditions. Compare this with the following SQL code used to perform an equivalent query on the relational tables shown in Figure 1(d).

```

select Country.countryCode
from Country inner join CountryLanguageUse
  on Country.countryCode = CountryLanguageUse.countryCode
where Country.isLarge = 1 and CountryLanguageUse.languageName = 'French'

```

To illustrate a query that relies on closed world semantics, consider the following LogiQL query to list those countries that are not large. LogiQL uses an *exclamation mark* “!” for the *logical negation operator* “*not*”, typically denoted by tilde “~” in logic.

```
_!(c) <- Country(c), !isLarge(c).
```

This is equivalent to the following formulation in logic: $\forall c [c \text{ isReturned} \leftarrow (\text{Country } c \ \& \ \sim c \text{ isLarge})]$. Figure 9 shows the result of running the query in the REPL tool. For simplicity, this figure omits display of internal identifiers for the returned values. The failure to find the Vatican City State and Luxembourg in the list of large countries is taken to mean that they are not large (negation as failure), so those two countries are returned in the result.



```
/>> query '_!(c) <- Country(c), !isLarge(c).  
,  
..  
=> VA  
LU
```

Figure 9 Using the query command to execute a sample query involving negation.

If you omit the condition `Country(c)` from the query by using “`_!(c) <- !isLarge(c)`”, this will generate an error because it violates one of the safety rules that ensure that legal queries execute in a finite time. This erroneous query asks instead for anything (not just any country!) in the universe of discourse that is not a large country. One of the strengths of datalog, including LogiQL, is its guarantee that rules that are syntactically valid will always terminate when executed.

Conclusion

The current article discussed some of the basic concepts used in deductive databases, such as Prolog or datalog-based systems, to model and query data, using the closed world assumption to make inferences involving negation. LogiQL was chosen as a leading edge example of such a system, and used to illustrate how such a logic-based approach can be used to facilitate both modeling and querying of databases. This short article provided only a brief glimpse at LogiQL’s more basic capabilities. Future articles in this series will examine how LogiQL can be used to specify business constraints and rules of a more advanced nature.

References

1. Abiteboul, S., Hull, R. & Vianu, V. 1995, *Foundations of Databases*, Addison-Wesley, Reading, MA.
2. Barker, R. 1990, *CASE*Method: Entity Relationship Modelling*, Addison-Wesley, Wokingham.
3. Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, 2nd edition, Morgan Kaufmann, San Francisco.
4. Halpin, T. & Wijbenga, J. 2010, ‘FORML 2’, *Enterprise, Business-Process and Information Systems Modeling*, eds. I. Bider et al., LNBI 50, Springer-Verlag, Berlin Heidelberg, pp. 247–260.
5. Huang, S., Green, T. & Loo, B. 2011, ‘Datalog and emerging applications: an interactive tutorial’, *Proc. 2011 ACM SIGMOD International Conference on Management of Data*, ACM, New York. <http://dl.acm.org/citation.cfm?id=1989456>.
6. Object Management Group 2013, *OMG Unified Modeling Language (OMG UML)*, version 2.5 RTF Beta 2. Available online at: <http://www.omg.org/spec/UML/2.5/Beta2/PDF/>.