Logical Data Modeling: Part 9

Terry Halpin
INTI International University

This is the ninth article in a series on logic-based approaches to data modeling. The first article [5] briefly overviewed deductive databases, and illustrated how simple data models with asserted and derived facts may be declared and queried in LogiQL [2, 16, 18], a leading edge deductive database language based on extended datalog [1]. The second article [6] discussed how to declare inverse predicates, simple mandatory role constraints and internal uniqueness constraints on binary fact types in LogiQL. The third article [7] explained how to declare *n*-ary predicates and apply simple mandatory role constraints and internal uniqueness constraints to them. The fourth article [8] discussed how to declare external uniqueness constraints. The fifth article [9] covered derivation rules in a little more detail, and showed how to declare inclusive-or constraints. The sixth article [10] discussed how to declare simple set-comparison constraints (i.e. subset, exclusion, and equality constraints), and exclusive-or constraints. The seventh article [11] explained how to declare subset, constraints between compound role sequences, including cases involving join paths. The eighth article [12] discussed how to declare exclusion and equality constraints between compound role sequences. The current article explains how to declare basic subtyping in LogiQL. The LogiQL code examples are implemented using the free, cloud-based REPL (Read-Eval-Print-Loop) tool for LogiQL that is accessible at https://developer.logicblox.com/playground/.

Asserted Subtypes

Table 1 shows an extract from report of about hospital patients. Each patient is identified by a patient number, has a name (not necessarily unique), and is classified as an inpatient (currently staying in the hospital) or an outpatient. Each inpatient is assigned to exactly one ward that includes a bed for that patient. Optionally, an outpatient may have a next hospital appointment scheduled. Other details about patients are maintained but are not of interest for this discussion. As an optional exercise, you might like to specify a data model for this example, including all relevant constraints, before reading on.

Table 1 Report extract of data about hospital patients

Patient Nr	*Patient Name*	*InPatient*	*OutPatient*	*Ward*	*Next Appointment*
101	John Smith	✓		A3	
102	Sue Jones		✓		
103	John Smith		✓		2017-07-22
104	Ann Green	✓		A3	

Figure 1 shows one way to model this example with an Object-Role Modeling (ORM) [13, 14, 15] diagram, using constructs discussed in previous articles. The preferred reference schemes for Patient, Ward and Date are depicted by the reference modes ".Nr", ".Code" and "ymd" (for year-month-day format) shown in parentheses. The patient classification is modeled using two unary fact types, and the ward and next appointment details are modeled using three binary fact types. The uniqueness constraint bar spanning Patient's role in the binary fact types Patient has PatientName, Patient is assigned to Ward and Patient has next appointment on Date indicates that each of these relationships is many-to-one. The mandatory role dot on Patient ensures that each patient has a name, and the exclusive-or constraint (lifebuoy symbol) declares that each patient is an inpatient or an outpatient but not both. The equality constraint (circled "=") ensures that a patient is an inpatient if and only if that patient is assigned to a ward, and the subset constraint (circled "⊆") declares that if a patient has a next appointment date then that patient is an outpatient.

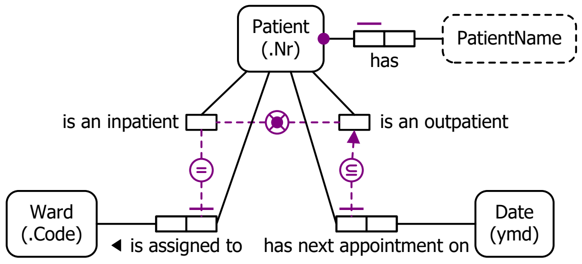


Figure 1 One way to model Table 1 in ORM notation, without subtyping.

The ORM schema in Figure 1 makes no use of subtyping, and may be coded in LogiQL using techniques discussed in previous articles. The populated ORM model depicted in Figure 2 shows an alternative way to model Table 1 using subtyping. The sample facts are shown in fact tables next to the relevant fact types and object types. The patient classification scheme is now depicted using InPatient and OutPatient subtypes, rather than unary fact types. The solid arrows declare InPatient and OutPatient as subtypes of Patient (their supertype) that inherit the same reference scheme as Patient. The exclusive-or constraint between the subtyping arrows ensures that each patient is an inpatient or an outpatient but not both. The mandatory role dot of InPatient ensures that each patient is assigned to a ward, and the lack of a mandatory role dot on OutPatient means that it is optional for outpatients to have a next appointment date.

If a given patient is an inpatient, this must be explicitly asserted. Similarly, if a given patient is an outpatient, this is simply asserted. Hence these subtypes are called *asserted subtypes*. Using these subtypes instead of simple unary predicates intuitively captures the intent to think of InPatient and OutPatient as object types that are useful for classifying things in the business domain. It also enables subtype-specific properties to be directly attached to the subtypes, which tends to simplify the model—this is especially the case if the subtypes have multiple specific properties.

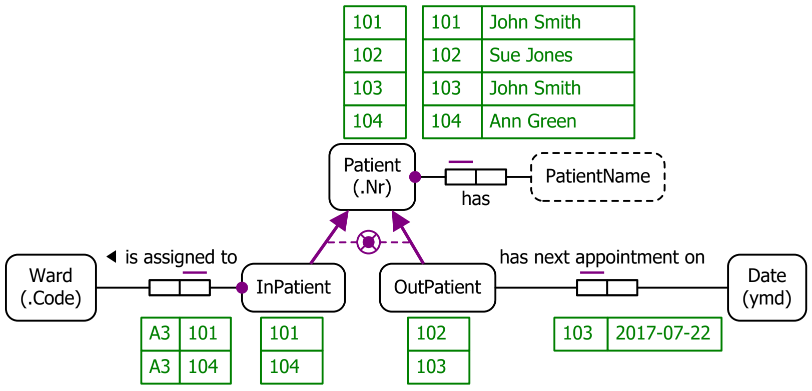


Figure 2 Another way to model Table 1 in ORM notation, using asserted subtypes (population included).

Figure 3(a) schematizes the same universe of discourse depicted in Table 1 as an Entity Relationship diagram in Barker notation (Barker ER) [3]. The Barker ER schema depicts the primary reference scheme for Patient and Ward by prepending an octothorpe "#" to the patient nr and ward code attributes respectively. The asterisk "*" prepended to the patient nr, patient name and ward code attributes indicates that these attributes are mandatory. The small "o" prepended to the next appointment date attribute indicates that this attribute is optional for outpatients. The two subtypes are depicted inside their supertype, and assumed to be mutually exclusive and collectively exhaustive. The half solid, half dashed line indicates that the ward assignment relationship is mandatory for InPatient and optional for Ward. The crowsfoot at just the Inpatient end of the relationship line indicates that the relationships are many-to-one. Predicate readings are provided at each end of the ward assignment relationship.

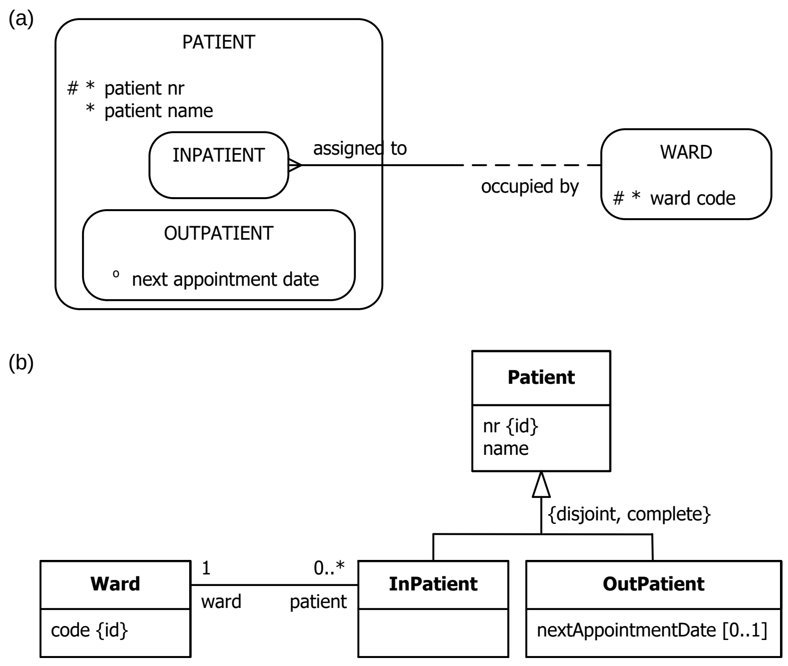


Figure 3 Data schema for Table 1 in (a) Barker ER, and (b) UML notation, using asserted subtypes.

Figure 3(b) schematizes Table 1 as a class diagram in the Unified Modeling Language (UML) [19]. The UML class diagram depicts the primary identification for Patient and Ward by appending "{id}" to the nr and code attributes respectively. The nr, name and code attributes by default have a multiplicity of 1, so are mandatory and single-valued. The multiplicity of 0..1 on nextAppoitnmentDate indicates that this is optianl and single-valued. The multiplicities at the end of the ward assignment association indicate that each inpatient has exactly one ward and each ward has zero or more patients. Names are provided for each association role. The arrowed lines from InPatient and OutPatient declare them as subclasses of Patient. The {disjoint, complete} qualification ensures that this subclassing scheme partitions Patient (i.e. each patient is an inpatient or outpatient but not both).

The ORM schema in Figure 2 may be coded in LogiQL as shown below. The right arrow symbol "->" stands for the material implication operator "$\rightarrow$" of logic, and is read as "implies". An exclamation mark "!" denote the logical negation operator and is read as "it is not the case that". A comma "," denotes the logical conjunction operator "&", and is read as "and". A semicolon ";" denotes the logical disjunction operator "$\vee$", and is read as "or" (in the inclusive-or sense). Recall that LogiQL is case-sensitive, each formula must end with a period, and head variables are implicitly universally quantified.

The first line of the LogiQL code declares Patient as an entity type whose instances are referenced by patient numbers that are coded as integers. The colon ":" in hasPatientNr(p:pNr) distinguishes hasPatientNr as a refmode predicate, so this predicate is injective (mandatory, 1:1). Similarly, the next line declares the entity type Ward along with its reference mode. Comments are prepended by "//", and describe the constraint declared in the code immediately following the comment. An underscore "_" denotes the anonymous variable, is used for existential quantification, and is read as "something". For example, the mandatory role constraint expressed in LogiQL as "Patient(p) -> patientNameOf[p] = _." corresponds to the logical formula "$\forall p$(Patient p $\rightarrow$ $\exists x$ p hasPatientName x)".

For explanatory purposes, the LogiQL code for the subtyping aspects is colored blue. The declarations lang:entity(`InPatient) and lang:entity(`OutPatient) ensure that InPatient and OutPatient are treated as entity types rather than just unary property predicates.

```
Patient(p), hasPatientNr(p:pNr) -> int(pNr).
Ward(w), hasWardCode(w:wc) -> string(wc).
InPatient(p) -> Patient(p).          // Each inpatient is a patient
```

```
lang:entity(`InPatient).                 // InPatient is an entity type
OutPatient(p)  ->  Patient(p).           // Each outpatient is a patient
lang:entity(`OutPatient).                // OutPatient is an entity type
Patient(p) ->  InPatient(p); OutPatient(p).    // Each patient is an inpatient or an outpatient
InPatient(p)  ->  !OutPatient(p).              // No inpatient is an outpatient

patientNameOf[p] = pn  ->  Patient(p), string(pn).
wardAssignedTo[p] = w  ->  Patient(p), Ward(w).
nextAppointmentDateOf[p] = d  ->  Patient(p), datetime(d).

Patient(p)  ->  patientNameOf[p] = _.          // Each patient has a name
InPatient(p)  ->  wardAssignedTo[p] = _.       // Each inpatient is assigned a ward
```

To enter the schema in the free, cloud-based REPL tool, use a supported browser such as Chrome, Firefox or Internet Explorer to access the website https://repl.logicblox.com. Alternatively, you can access https://developer.logicblox.com/playground/, then click the “Open in new window” link to show 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”, and click the addblock option that then appears. This causes the addblock command (followed by a space) to be added to the code window. Typing a 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 4).

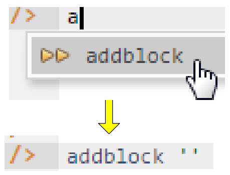
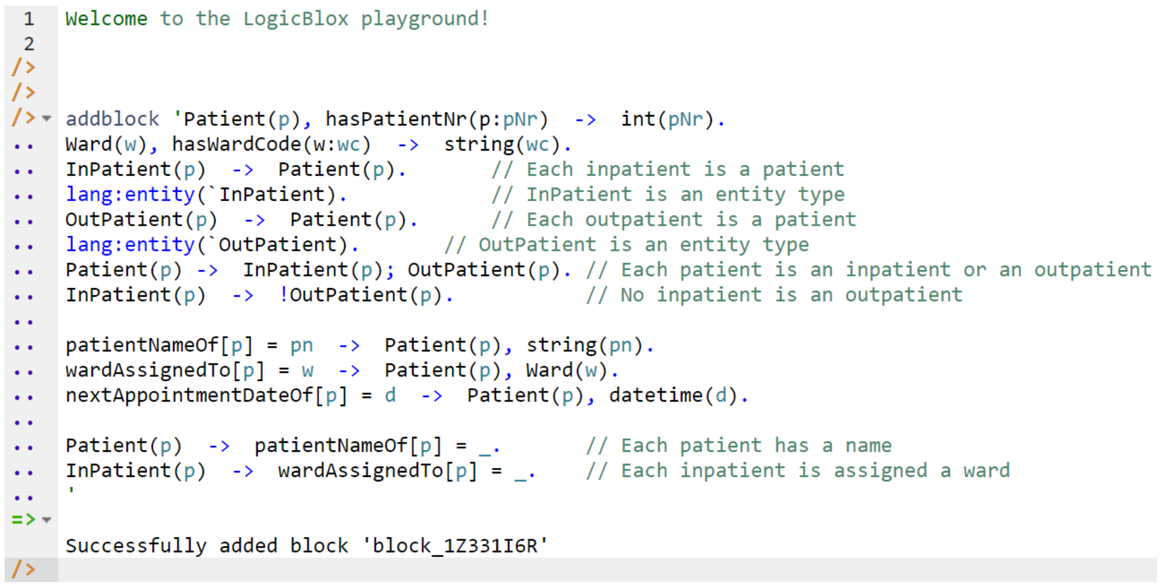


Figure 4 Invoking the addblock command in the REPL tool.

Now copy the schema 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. You are now notified that the block was successfully added, and a new prompt awaits your next command (see Figure 5). By default, the REPL tool also appends an automatically generated identifier for the code block. Alternatively, you can enter each line of code directly, using a separate addblock command for each line.



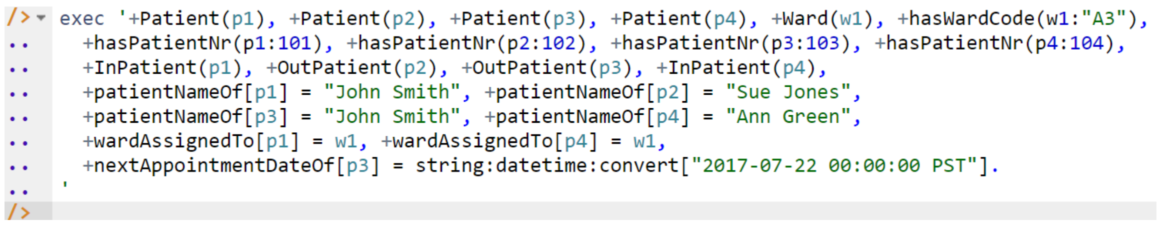
```
Welcome to the LogicBlox playground!

/>
/>
/> addblock 'Patient(p), hasPatientNr(p:pNr)  ->  int(pNr).
.. Ward(w), hasWardCode(w:wc)  ->  string(wc).
.. InPatient(p)  ->  Patient(p).        // Each inpatient is a patient
.. lang:entity(`InPatient).             // InPatient is an entity type
.. OutPatient(p)  ->  Patient(p).       // Each outpatient is a patient
.. lang:entity(`OutPatient).        // OutPatient is an entity type
.. Patient(p) ->  InPatient(p); OutPatient(p). // Each patient is an inpatient or an outpatient
.. InPatient(p)  ->  !OutPatient(p).           // No inpatient is an outpatient
..
.. patientNameOf[p] = pn  ->  Patient(p), string(pn).
.. wardAssignedTo[p] = w  ->  Patient(p), Ward(w).
.. nextAppointmentDateOf[p] = d  ->  Patient(p), datetime(d).
..
.. Patient(p)  ->  patientNameOf[p] = _.       // Each patient has a name
.. InPatient(p)  ->  wardAssignedTo[p] = _.    // Each inpatient is assigned a ward
.. '
=>
Successfully added block 'block_1Z331I6R'
/>
```

Figure 5 Adding a block of schema code.

The data in Table 1 may be entered in LogiQL using the following delta rules. A delta rule of the form +*fact* inserts that fact. Recall that *plain, double quotes* (i.e. ",") are needed here, not single quotes or smart double quotes. 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. Dates are entered using the string:datetime:convert function. Note that a time of day must be included with the date, even if this is not of interest. In this example, I've chosen 00:00:00 (0 hours, 0 minutes, 0 seconds) as the time of day, and declared Pacific Standard Time (PST) as the time zone If an explicit time zone is omitted, the time zone is assumed to be UTC by default.

```
+Patient(p1), +Patient(p2), +Patient(p3), +Patient(p4), +Ward(w1), +hasWardCode(w1:"A3"),
 +hasPatientNr(p1:101), +hasPatientNr(p2:102), +hasPatientNr(p3:103), +hasPatientNr(p4:104),
 +InPatient(p1), +OutPatient(p2), +OutPatient(p3), +InPatient(p4),
 +patientNameOf[p1] = "John Smith", +patientNameOf[p2] = "Sue Jones",
 +patientNameOf[p3] = "John Smith", +patientNameOf[p4] = "Ann Green",
 +wardAssignedTo[p1] = w1, +wardAssignedTo[p4] = w1,
 +nextAppointmentDateOf[p3] = string:datetime:convert["2017-07-22 00:00:00 PST"].
```

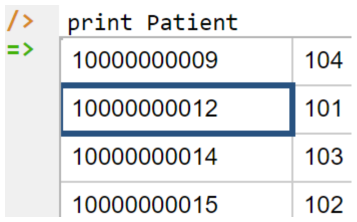
Delta rules to add or modify data are entered using the exec (for 'execute') command. To invoke the exec command in the REPL tool, type "e" and then select exec from the drop-down list. A space character is automatically appended. Typing a single quote after the exec command and space causes a pair of single quotes to be appended, with your cursor placed inside those quotes ready for your delta rules. Now copy the 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 (see Figure 6).



```
/> exec '+Patient(p1), +Patient(p2), +Patient(p3), +Patient(p4), +Ward(w1), +hasWardCode(w1:"A3"),
..   +hasPatientNr(p1:101), +hasPatientNr(p2:102), +hasPatientNr(p3:103), +hasPatientNr(p4:104),
..   +InPatient(p1), +OutPatient(p2), +OutPatient(p3), +InPatient(p4),
..   +patientNameOf[p1] = "John Smith", +patientNameOf[p2] = "Sue Jones",
..   +patientNameOf[p3] = "John Smith", +patientNameOf[p4] = "Ann Green",
..   +wardAssignedTo[p1] = w1, +wardAssignedTo[p4] = w1,
..   +nextAppointmentDateOf[p3] = string:datetime:convert["2017-07-22 00:00:00 PST"].
..  '
/>
```

Figure 6 Adding the data.

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 the patient numbers of all the recorded patients, type "p" then select print from the drop-down list, then type a space followed by "P", then select Patient from the drop-down list and press Enter. Alternatively, type "print Patient" yourself and press Enter. Figure 7 shows the result. By default, the REPL tool prepends a column listing automatically generated, internal identifiers for the returned entities. Similarly, you can use the print command to print the extension of the other predicates.



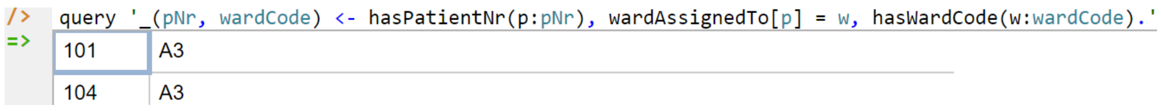
```
/> print Patient
=> 10000000009   104
   10000000012   101
   10000000014   103
   10000000015   102
```

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

As discussed in previous articles, to perform a query, you specify a derivation rule to compute the facts requested by the query. For example, the following query may be used to list the patient number and ward code for those patients assigned to a ward. The rule's head uses an anonymous predicate to capture the result derived from the rule's body. The head variables *pNr* and *wardCode* are implicitly universally quantified. The variables *p* and *w* introduced in the rule body are implicitly existentially quantified.

```
_ (pNr, wardCode) <-  hasPatientNr(p:pNr), wardAssignedTo[p]=w, hasWardCode(w:wardCode).
```

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 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.



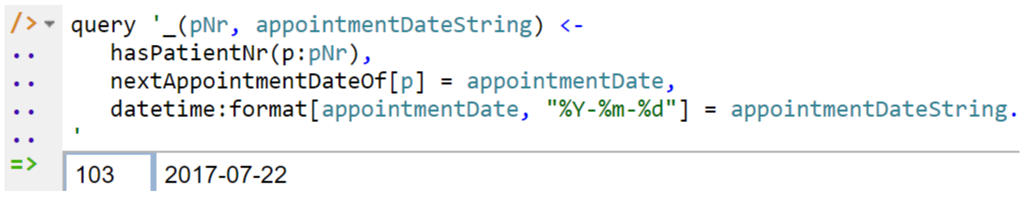
```
/> query '_(pNr, wardCode) <- hasPatientNr(p:pNr), wardAssignedTo[p] = w, hasWardCode(w:wardCode).'
=> 101  A3
   104  A3
```

Figure 8 A query to list the patient number and ward code for those patients assigned to a ward.

As an example query that requests temporal data, the following query will list the patient number and next appointment date of those patients who have a next appointment scheduled. The datetime:format function is used to display the date in the desired format. In this case, extended ISO format is specified using the format string "%Y-%m-%d". For other datetime formats and further details on a wide range of datetime functions, see the core reference manual at https://developer.logicblox.com/content/docs4/core-reference/.

```
_(pNr, appointmentDateString) <-
  hasPatientNr(p:pNr),
  nextAppointmentDateOf[p] = appointmentDate,
  datetime:format[appointmentDate, "%Y-%m-%d"] = appointmentDateString.
```

Figure 9 shows a screenshot of the query and the result returned.



```
/> query '_(pNr, appointmentDateString) <-
..    hasPatientNr(p:pNr),
..    nextAppointmentDateOf[p] = appointmentDate,
..    datetime:format[appointmentDate, "%Y-%m-%d"] = appointmentDateString.
..  '
=> 103  2017-07-22
```

Figure 9 A query to list the patient number and next appointment date for those patients with an appointment.

Derived Subtypes

Table 2 shows a slightly different report extract for the hospital example under discussion. Instead of using InPatient and OutPatient columns, it uses a single "Patient Kind" column where the entries "In" and "Out" indicate inpatient and outpatient status respectively.

Table 2 A slightly different report extract of data about hospital patients

Patient Nr	*Patient Name*	*Patient Kind*	*Ward*	*Next Appointment*
101	John Smith	In	A3	
102	Sue Jones	Out		
103	John Smith	Out		2017-07-22
104	Ann Green	In	A3	

An ORM schema for Table 2 is shown in Figure 10. The fact type "Patient is of PatientKind" is used to record the patient kind information, enabling us to derive which patients are inpatients and which patients are outpatients using the subtype derivation rules displayed at the bottom on the figure. InPatient and OutPatient are now *derived subtypes*. The derived nature of the subtypes is indicated by appending an asterisk "*" to their names, and providing the subtype derivation rules used to define them.

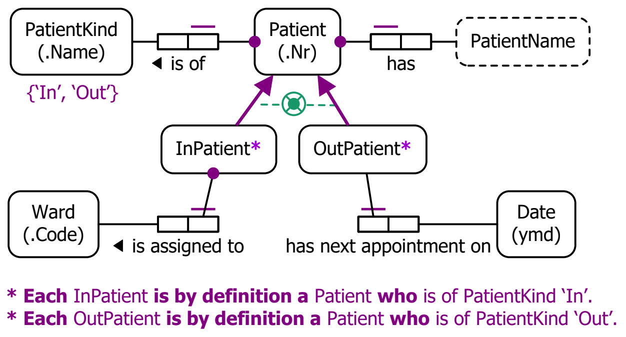


Figure 10 Modeling Table 2 in ORM notation, using derived subtypes.

The combination of the sutbtype definitions and the mandatory role, uniqueness and value constraints on the "Patient is of PatientKind" fact type implies the exclusive-or constraint over the subtyping. This exclusive-or constraint is now colored green, to indicate that it is derivable.

Neither Barker ER nor UML provide graphical support for subtype derivations, and are ignored for this example. The schema for this example may be coded in LogiQL as follows. Compared with the previous schema code, the exclusion constraint code is omitted (as it is implied), and some new code (colored blue) is added.

```
Patient(p), hasPatientNr(p:pNr)  ->  int(pNr).
Ward(w), hasWardCode(w:wc)  ->  string(wc).
PatientKind(pk), hasPatientKindName(pk:pkn)  ->  string(pkn).
InPatient(p)  ->  Patient(p).          // Each inpatient is a patient
lang:entity(`InPatient).               // InPatient is an entity type
OutPatient(p)  ->  Patient(p).         // Each outpatient is a patient
lang:entity(`OutPatient).              // OutPatient is an entity type

patientNameOf[p] = pn  ->  Patient(p), string(pn).
patientKindOf[p] = pk  ->  Patient(p), PatientKind(pk).
wardAssignedTo[p] = w  ->  Patient(p), Ward(w).
nextAppointmentDateOf[p] = d  ->  Patient(p), datetime(d).

Patient(p)  ->  patientNameOf[p] = _.          // Each patient has a name
Patient(p)  ->  patientKindOf[p] = _.          // Each patient has a patient kind
hasPatientKindName(_:pkn)  ->  pkn = "In" ; pkn = "Out".
InPatient(p)  ->  wardAssignedTo[p] = _.       // Each inpatient is assigned a ward

// subtype derivation rules
InPatient(p)  <-  patientKindOf[p] = pk, hasPatientKindName(pk, "In").
OutPatient(p)  <-  patientKindOf[p] = pk, hasPatientKindName(pk, "Out").
```

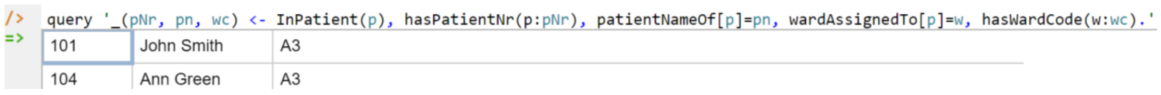
The sample data may be coded as shown below. Note that the explicit assertions of inpatient and outpatient status used in the previous section are omitted since these facts are now derivable.

```
+Patient(p1), +Patient(p2), +Patient(p3), +Patient(p4), +Ward(w1), +hasWardCode(w1:"A3"),
+PatientKind(pki), +hasPatientKindName(pki:"In"),
+PatientKind(pko), +hasPatientKindName(pko:"Out"),
  +hasPatientNr(p1:101), +hasPatientNr(p2:102), +hasPatientNr(p3:103), +hasPatientNr(p4:104),
  +patientKindOf[p1] = pki, +patientKindOf[p2] = pko,
  +patientKindOf[p3] = pko, +patientKindOf[p4] = pki,
  +patientNameOf[p1] = "John Smith", +patientNameOf[p2] = "Sue Jones",
  +patientNameOf[p3] = "John Smith", +patientNameOf[p4] = "Ann Green",
  +wardAssignedTo[p1] = w1, +wardAssignedTo[p4] = w1,
  +nextAppointmentDateOf[p3] = string:datetime:convert["2017-07-22 00:00:00 PST"].
```

The schema and data may be entered in the usual way. As a sample query on this model, the following query returns the patient number, patient name, and ward for each inpatient.

```
_(pNr, pn, wc) <- InPatient(p), hasPatientNr(p:pNr), patientNameOf[p]=pn,
                  wardAssignedTo[p]=w, hasWardCode(w:wc).
```

Figure 11 shows a screenshot with the query result.



```
/> query '_(pNr, pn, wc) <- InPatient(p), hasPatientNr(p:pNr), patientNameOf[p]=pn, wardAssignedTo[p]=w, hasWardCode(w:wc).'
=> 101    John Smith    A3
   104    Ann Green     A3
```

Figure 11 A query to list the patient number, patient name, and ward for each inpatient.

Conclusion

The current article discussed how to declare basic subtyping in LogiQL. Future articles in this series will examine how LogiQL can be used to specify business constraints and rules of a more advanced nature. The core reference manual for LogiQL is accessible at https://developer.logicblox.com/content/docs4/core-reference/. An introductory tutorial for LogiQL and the REPL tool is available at https://developer.logicblox.com/content/docs4/tutorial/repl/section/split.html. Further coverage of LogiQL may be found in [16].

Acknowledgement: I'm grateful to Steve Beck for his help with the use of datetime functions in LogiQL.

References

1. Abiteboul, S., Hull, R. & Vianu, V. 1995, *Foundations of Databases*, Addison-Wesley, Reading, MA.
2. Aref, M., Cate, B., Green T., Kimefeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. & Washburn, G. 2015, 'Design and Implementation of the LogicBlox System', *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, ACM, New York. http://dx.doi.org/10.1145/2723372.2742796.
3. Barker, R. 1990, *CASE*Method: Entity Relationship Modelling*, Addison-Wesley, Wokingham.
4. Curland, M. & Halpin, T. 2010, 'The NORMA Software Tool for ORM 2', P. Soffer & E. Proper (Eds.): *CAiSE Forum 2010*, LNBIP 72, pp. 190-204, Springer-Verlag Berlin Heidelberg 2010.
5. Halpin, T. 2014, 'Logical Data Modeling: Part 1', *Business Rules Journal*, Vol. 15, No. 5 (May, 2014), URL: http://www.BRCommunity.com/a2014/b760.html.
6. Halpin, T. 2014, 'Logical Data Modeling: Part 2', *Business Rules Journal*, Vol. 15, No. 10 (Oct., 2014), URL: http://www.BRCommunity.com/a2014/b780.html.
7. Halpin, T. 2015, 'Logical Data Modeling: Part 3', *Business Rules Journal*, Vol. 16, No. 1 (Jan., 2015), URL: http://www.BRCommunity.com/a2015/b795.html.
8. Halpin, T. 2015, 'Logical Data Modeling: Part 4', *Business Rules Journal*, Vol. 16, No. 7 (July, 2015), URL: http://www.BRCommunity.com/a2015/b820.html.
9. Halpin, T. 2015, 'Logical Data Modeling: Part 5', *Business Rules Journal*, Vol. 16, No. 10 (Oct., 2015), URL: http://www.BRCommunity.com/a2015/b832.html.
10. Halpin, T. 2016, 'Logical Data Modeling: Part 6', *Business Rules Journal*, Vol. 17, No. 3 Mar., 2016), URL: http://www.BRCommunity.com/a2016/b852.html.
11. Halpin, T. 2016, 'Logical Data Modeling: Part 7', *Business Rules Journal*, Vol. 17, No. 7 (July, 2016), URL: http://www.brcommunity.com/a2016/b866.html.
12. Halpin, T. 2016, 'Logical Data Modeling: Part 8', *Business Rules Journal*, Vol. 17, No. 11 (Nov, 2016), URL: http://www.brcommunity.com/a2016/b883.html.
13. Halpin, T. 2015, *Object-Role Modeling Fundamentals*, Technics Publications, New Jersey.
14. Halpin, T. 2016, *Object-Role Modeling Workbook*, Technics Publications, New Jersey.

15. Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases, 2nd edition*, Morgan Kaufmann, San Francisco.
16. Halpin, T. & Rugaber, S. 2014, *LogiQL: A Query language for Smart Databases*, CRC Press, Boca Raton. http://www.crcpress.com/product/isbn/9781482244939#.
17. Halpin, T. & Wijbenga, J. 2010, 'FORML 2', *Enterprise, Business-Process and Information Systems Modeling*, eds. I. Bider et al., LNBIP 50, Springer-Verlag, Berlin Heidelberg, pp. 247–260.
18. 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.
19. 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/.
20. OMG, 2012, *OMG Object Constraint Language (OCL), version 2.3.1*. Retrieved from http://www.omg.org/spec/OCL/2.3.1/.