An Alternative High-level Approach to Interaction with Databases

António Porto

LIACC / DCC, Faculdade de Ciências
Universidade do Porto
ap@dcc.fc.up.pt

Abstract. Most applications rely on relational databases for persistence, interacting through SQL embedded in the host programming language. Such code being error-prone and hard to maintain, many proposals have been made to raise its level, mostly in the direction of deductive and/or object-oriented databases. We have put forward an alternative approach, inspired by natural language, that packs a lot of power in very concise and readable code, while relying on standard database technology. This is achieved using the flexible term syntax and deductive capabilities of logic programming, both to compile a database scheme from a high-level description, and to interpret high-level queries and commands. In this paper we review the basic ideas of the novel approach and concentrate on the interaction language of queries and commands, formalizing its semantics on the basis of characterizing canonical database schemes. These rely on uniform notions of concept, attribute and reference, rather than the dichotomous entity-relationship model. Query and command expressions are variable-free terms, reading very naturally when appropriate nouns (rather than verbs) are chosen for all concept and attribute names. Attribute inheritance and chaining avoid many explicit joins, which are automatically derived as inner or outer joins.
Scheme-derived multi-valued global parameters can flexibly be used for implicit current values. The abstraction power is further raised by having manifold attributes, whose values may actually vary along a parametric domain, the main examples being the handling of temporal validity and multi-lingual data. Commands can also be very high-level, with simple statements possibly resulting in large transactions.

1 Introduction

About a decade ago we launched a large project to implement a full-blown academic management system for our institution. We had a vision of what a proper information system should be, that raised a corresponding need of proper software technology to be achievable, namely a concept-oriented architecture, declarative sub-languages, generic programming capabilities, and a high-level interaction-persistence link. So we ended up with the double challenge of re-engineering the academic processes through a proper information system and building proper software support for implementing the system.

We successfully met our goals using logic programming to develop the software [1], with some scientific achievements along the way. Paramount among these was a novel high-level approach for interacting with databases, offering a language for defining the scheme and another for queries and commands, inspired by natural language noun phrases. Having reported a first overview of the approach [2], in this paper we give a more detailed account of the interaction language of queries and commands, presenting a formal semantics that rigorously anchors the illustrating examples, stemming from a new characterization of a canonical database scheme.

In the quest for better, more maintainable code for database access, we are unaware of other approaches quite similar in spirit to our own. In deductive databases (see e.g. [3, 4]) database tables are viewed as predicates in logic programs, with joins expressed through variable sharing. We rely instead on a few fixed query and command predicates, whose "input" arguments are ground terms structurally derived from simple natural language principles and lifted from the atomic terminology introduced for concepts and attributes in a given scheme definition. Deductive object-oriented approaches, such as F-logic [5], are closer in spirit to our own by using the equivalent of our concepts (classes), attributes (methods) and references (types), but differ essentially on their view that data storage is object-oriented whereas we keep it relational (more flexible) while allowing object-orientation in our queries and commands. Accordingly, we use a different language for defining a scheme than for querying a corresponding database, and precompile the scheme into unit clauses used for the interpretation of queries and commands. An explicit logic programming representation of a scheme has been advocated before [6], but to encode basic set-theoretic constructions supporting the low-level relational calculus as a query language. In deductive and/or object-oriented databases logic variables are used to denote individuals inside query expressions, but we avoid them altogether (much as natural language does); in this respect we are closer to using concept languages (see e.g. [7]), also known as description logics [8]. Another major difference is our conservative stance of keeping the power of deductive rules outside of the semantics of the persistent storage. This apparent shortcoming was actually not an issue in building our large real-world application, and, importantly, it allows the use of stable, well-established logic programming and relational database technologies. Finally, a distinguishing methodological standpoint is that we attach great importance to the choice of concept and attribute names, given its impact on how natural (thus readily understandable) conceptual expressions become.

Our framework has two major components, a compiler for a scheme definition language NACS (NAtural Conceptual Scheme) and an interpreter for a query and command language NADI (NAtural Database Interaction) using the compiled scheme. In this paper we focus on the latter; a taste of the former is found in [2]. For presenting the semantics of queries we do need a scheme's abstract characterization, that we present in a preliminary section. In essence we rely on the notions of concept, attribute and reference (sets, functions and set inclusion), with attributes being lexical (yielding atomic values) or non-lexical (with a lexical tuple signature), identity or dependent (part of the primary key or not), and having a simple unambiguous semantics under a definition of canonical scheme. The next section presents the syntax and semantics for queries. Syntax exploits the flexible usage of prefix and infix operator syntax in Prolog, with e.g. `a/c$(b<x)` reading as "the a of the c whose b is less than x". Semantics is expressed in terms of tuples of values from a database. A highlight of NADI's abstraction power is the use of manifold attributes for dealing with temporal and multilingual data. Commands are overview-ed next, and we conclude with a summary of achievements and future work.

1.1 Notational conventions

Object-language expressions are in fixed-length teletype font, with standard term notation including the use of prefix and infix operators.

In the meta-language we extensively use sequences (tuples), with two structural notations: parenthesized comma-separated elements $(e_1, \ldots, e_n)$, and simple concatenation, with Roman and Greek letters ranging over respectively elements and sequences, say $e\sigma$ for the sequence with first element e and remaining (possibly empty) sub-sequence σ. The symbol ε denotes the empty sequence. The notation σ_i is used to refer to the ith element in the sequence σ, with indexes starting at 1. We use $\{\!\{\sigma\}\!\}$ for the set of elements in the sequence σ.

The subscript $_{\text{fin}}$ signals restriction to finite sets, and S^+ denotes the finite non-empty sequences of elements of S.

We refer to the domains of functions with dom. Currying is assumed for functions, with the notation f_x for $f(x)$ and f_x^y for $f(x, y)$, e.g. for a function $f : A \to B \rightharpoonup C$ (total on A, partial on B) we have $dom(f) \subseteq A \times B$ and $a \in A \Rightarrow dom(f_a) \subseteq B$.

In any formal statement we implicitly assume universal quantification of the free meta-variables in the appropriate domains (as with a above).

2 The scheme

As announced, we do not present here our own concrete language NACS for defining a database scheme. Nevertheless we must convey the structure of its outcome, i.e. the formal characterization of a scheme and a corresponding database.

After an initial brief summary of our modelling viewpoint we formalize a scheme as an abstract syntactic structure lifted from a vocabulary of names, and provide a corresponding database semantics lifted from the domains of the data types. We first give a scheme definition that is too broad, but which allows the definition of auxiliary notions (including sub-concepts) that finally help in defining a canonical scheme, on which to base the semantics of queries.

2.1 The modelling viewpoint

We consider irrelevant, and detrimental to our simplicity goal, the traditional distinction between entities and relationships made in most approaches to database modelling. We find it much more convenient to have a uniform view of *concepts* encompassing all our usual carving up of the world into types of entities, relationships, situations, events, etc. for which we have nouns in natural language such as student, course edition, enrollment, registration, etc.

These concepts are all amenable to a simple semantics of sets of individual "things" (henceforth called *individuals*) with an inner structure given by a set of local *attributes* mapping each individual in a concept to a tuple of individuals in other concepts, with the data types as the base concepts.

Attributes also correspond to nouns in natural language, usable to form attribute chains such as "the name of the lecturer of the course" that will find a direct counterpart in our query language. The *lexical* attributes are those with atomic values, and any attribute has a signature as a (possibly unary) tuple of lexical attributes. Every concept has an *identity* tuple of lexical attributes, e.g. the (student, course, year, period type, period number) of an enrollment. The classical entity-relationship distinction is the linguistically unimportant one between concepts with unary and non-unary lexical identity.

The concepts and lexical attributes map directly into tables and fields of a relational database, with the lexical identity attributes constituting the primary key and attributes with concept images yielding foreign keys.

We can contrast our modelling viewpoint with the OWL language [9] proposed for the Semantic Web [10]. We give importance to attributes with their functional nature and noun-phrase terminology, whereas in OWL they are particular properties that in general are relations, and mostly adapt verbal terminology with ensuing variant forms and the need to relate them, e.g. as inverse. For example we may have a `course_edition` concept with attributes (among others) `course` and `lecturer`, whereas in OWL we would have a `teaches` property relating lecturers and course editions and the inverse `isTaughtBy`.

2.2 The scheme structure

The basic concepts are the *data types* $\mathcal{T}$ that are usual in relational databases, including $\{\, \texttt{int}, \texttt{num}, \texttt{date}, \texttt{time} \,\} \cup \{\, \texttt{str(}n\texttt{)} \mid n \text{ is a positive numeral} \,\}$, with predefined interpretations as the sets of (respectively) integers, numbers, dates, time points and strings of at most n characters; included in $\mathcal{T}$ are also a few other size-parametric data types for large textual and binary objects, that we will not illustrate in this paper.

A conceptual scheme is an abstract syntactic structure built from the set of data types $\mathcal{T}$ and a set of names $\mathcal{N}$ disjoint from $\mathcal{T}$ that includes the special names `*` and `{}`, respectively "all" and "self".

Definition 1. *A* conceptual scheme *is a 4-tuple* $\langle C, A, T, R \rangle$ *with a* concept *set* $C \subset_{\text{fin}} \mathcal{N}$ *and mappings for* attribute signature $A : C \to \mathcal{N} \rightharpoonup \mathcal{N}^{+}$*,* lexical type $T : C \to \mathcal{N} \rightharpoonup \mathcal{T}$ *and* conceptual reference $R : C \rightharpoonup \mathcal{N} \rightharpoonup \wp_{\text{fin}}(C)$*, satisfying the following conditions:*

1. *the* lexical attributes *are* $L_c = dom(T_c) = \{\, a \mid A_c^a = a \,\} \subset_{\text{fin}} \mathcal{N}$*;*
2. $(A_c^a)_i \in L_c$*, and* $(A_c^a)_i \neq (A_c^a)_j$ *if* $i \neq j$*;*

3. $\{\texttt{\{\}}, \texttt{*}\} \subseteq dom(A_c)$, *with the* whole $A_c^{\texttt{*}}$ *being a permutation of* $L(c)$ *and the* identity $A_c^{\texttt{\{\}}}$ *a prefix thereof;*
4. $dom(R_c) \subseteq dom(A_c) \backslash \{\texttt{*}\}$;
5. $c' \in R_c^a \;\Rightarrow\; \tau_{c'}^{\texttt{\{\}}} = \tau_c^a$, *with the* tuple type $\tau : dom(A) \to \mathcal{T}^+$ *defined by* $(\tau_c^a)_i = T_c((A_c^a)_i)$;
6. $c' \in R_c^{\texttt{\{\}}} \Rightarrow A_{c'}^{\texttt{\{\}}} = A_c^{\texttt{\{\}}}$.

We can see that each concept c has a non-empty finite set of lexical attributes (those with atomic data types) that are totally ordered in the whole signature $A_c^{\texttt{*}}$, a prefix of which is the identity signature $A_c^{\texttt{\{\}}}$, meant to uniquely identify individuals of c, as will be apparent in the forthcoming definition of database semantics. For convenience a concept may also have non-lexical attributes, standing for their signature's tuple of lexical ones. As an example, consider the concept `course_edition` whose lexical attributes are those of its identity (`course`, `year`, `period_type`, `period_number`) and the (dependent) `lecturer`, along with a non-lexical `period` with signature (`period_type`, `period_number`). An attribute a of c may reference other concepts R_c^a, its lexical tuple A_c^a having the same tuple type as the identity of every referred concept. In the latter example we could have $R_{\texttt{course\_edition}}^{\texttt{lecturer}} = \{\texttt{person}\}$ with $\tau_{\texttt{course\_edition}}^{\texttt{lecturer}} = \tau_{\texttt{person}}^{\texttt{\{\}}} = \texttt{int}$.

The scheme definition language has features not reflected in this simple scheme structure, namely virtual concepts and sub-attribute inheritance, that contribute to the abstraction power of the interaction language. We can also define attributes as having null values; this can be captured simply by having in $\mathcal{T}$ extended versions $*t$ of each primitive data type t, whose domains incorporate the special null value `[]`, i.e. $[\![\, *t \,]\!] = [\![\, t \,]\!] \uplus \{\, \texttt{[]} \,\}$.

Database semantics A conceptual scheme can be given a semantics in terms of its possible *databases*, a notion lifted from the semantic domains of values $[\![\, \tau \,]\!]$ for the data types $\tau \in \mathcal{T}$, with $[\![\, \tau_1 \cdots \tau_n \,]\!] = [\![\, \tau_1 \,]\!] \times \cdots \times [\![\, \tau_n \,]\!]$ for tuple types.

Definition 2. *A* database Δ *for a scheme* $\langle C, A, T, R\rangle$ *is a mapping of* C *where:*
1. $\Delta_c \subseteq_{\text{fin}} [\![\, \tau_c^{\texttt{*}}]\!]$;
   the attribute projections $\pi_c^a : \Delta_c \to [\![\, \tau_c^a \,]\!]$ *are defined by* $(\pi_c^a(t))_i = t_j$ *if* $(A_c^a)_i = (A_c^{\texttt{*}})_j$;
2. $t, t' \in \Delta_c,\, t \neq t' \;\;\Rightarrow\;\; \pi_c^{\texttt{\{\}}}(t) \neq \pi_c^{\texttt{\{\}}}(t')$;
3. $t \in \Delta_c,\, c' \in R_c^a \;\;\Rightarrow\;\; \exists t' \in \Delta_{c'} \,.\, \pi_c^a(t) = \pi_{c'}^{\texttt{\{\}}}(t')$.

Each member of Δ_c is the syntactic characterization of an individual of the concept c as a tuple of values, of the prescribed data types, for the concept's *whole* tuple $A_c^{\texttt{*}}$ of lexical attributes. Condition 1 ensures type correctness. The role of the self attribute `{}` as *identity*, corresponding to the chosen *primary key* in database parlance, is enforced by condition 2. Finally any conceptual reference, corresponding to a database *foreign key*, is satisfied by condition 3.

Non-lexical attributes other than `*` or `{}` are not strictly needed for defining a scheme and database, as we could define R directly on lexical tuples. But they are essential abstractions for writing concise scheme definitions and also queries and commands, where they can be global parameters (see later).

Canonical scheme In practice we want to tighten the given definition of a scheme, to ensure reasonable syntactic and semantic restrictions that simplify, without sacrificing expressiveness, the handling of queries and commands. For this purpose we define first a few auxiliary notions.

Definition 3. *Given a conceptual scheme* $\langle C, A, T, R \rangle$,

1. ${}^{I}\!A_c = \{\, a \mid \{\!\{A_c^a\}\!\} \subseteq \{\!\{A_c^{\{\}}\}\!\} \,\}$ *are the* identity *attributes of* c*;*
2. ${}^{D}\!A_c = dom(A_c) \backslash ({}^{I}\!A_c \uplus \{*\})$ *are the* dependent *attributes of* c*;*
3. *the transitive closure of* $\{\, (c, c') \mid c' \in R_c^{\{\}} \,\}$ *is the* sub-concept *relation* $\sqsubset$.

As an example, consider `user` $\sqsubset$ `person` $\sqsubset$ `agent` with ${}^{I}\!A_{\texttt{user}} = {}^{I}\!A_{\texttt{person}} = {}^{I}\!A_{\texttt{agent}} = \{\texttt{\{\}}, \texttt{id\_code}\}$, ${}^{D}\!A_{\texttt{user}} = \{\texttt{user\_id}\}$, ${}^{D}\!A_{\texttt{person}} = \{\texttt{sex}\}$ and ${}^{D}\!A_{\texttt{agent}} = \{\texttt{name}, \texttt{common\_name}\}$.

The sub-concept relation, in our formalization a particular case of conceptual reference, is actually a very important notion when modelling a real-world domain. It must give rise to a proper hierarchy, with implicit inheritance of attributes reflected in the interaction language. Hierarchically related concepts share their identity attributes, being distinguishable through their dependent attributes. In the interaction language attributes should be usable across hierarchy paths in an unambiguous way. Such concerns are addressed as follows.

Definition 4. *A conceptual scheme* $\langle C, A, T, R \rangle$ *is* canonical *if*

1. $\sqsubset$ *is a strict partial order, i.e.* $c \not\sqsubset c$*;*
2. *every concept* $c \in C$ *has a unique* root $\overline{c} \in C$ *such that* $c \sqsubseteq c' \;\Rightarrow\; c' \sqsubseteq \overline{c}$, *where* $\sqsubseteq$ *is the reflexive closure of* $\sqsubset$*;*
3. $c_1, c_2 \in R_c^a \;\Rightarrow\; \overline{c_1} = \overline{c_2}$*;*
4. *hierarchically related concepts share identity but not dependent attributes, i.e.* $c_1 \neq c_2,\ \overline{c_1} = \overline{c_2} \;\Rightarrow\; {}^{I}\!A_{c_1} = {}^{I}\!A_{c_2},\ {}^{D}\!A_{c_1} \cap {}^{D}\!A_{c_2} = \emptyset$.

The first condition is not surprising. In NACS it is automatically enforced, because one can only refer to previously defined concepts when defining a new one, thereby avoiding referential loops.

The condition on unique roots may not seem essential, but it is quite natural to assume, as ideally the sub-concept relation is viewed as set inclusion for individuals of the same sort, and we naturally consider the root concept of the whole sort. The third condition makes the natural demand that an attribute value may be of only one sort.

The uniqueness of attributes among hierarchically related concepts is very important for unambiguously giving meaning to attributive expressions in the NADI interaction language. This uniqueness is quite natural in modelling, since attribute names should be nouns in natural language that reflect a (possibly partial) *functional* property of individuals of the same sort. In the previous example the `id_code` of a `user`, `person` or `agent` uniquely refers to their (shared) identity code, the `name` of a `person` is uniquely defined (through `agent`), and the `user_id` of a `person` also uniquely defined (if the `person` is a `user`). The same dependent attribute in two database tables sharing identity would open the door to the inconsistency of multiple attribute values for the same individual.

So we extend and classify the attributes applicable to a concept, as follows.

Definition 5. *For any concept c in a canonical conceptual scheme:*
1. ${}^{H}\!A_c = \bigcup_{c' \sqsubseteq \overline{c}} {}^{D}\!A_{c'}$ *are the* hierarchical *attributes;*
2. ${}^{P}\!A_c = \{\, a \in {}^{H}\!A_c \mid c \sqsubseteq c' \Rightarrow a \notin {}^{D}\!A_{c'} \,\}$ *are the* partial *attributes.*
3. ${}^{T}\!A_c = {}^{I}\!A_c \uplus ({}^{H}\!A_c \backslash {}^{P}\!A_c)$ *are the* total *attributes;*
4. ${}^{N}\!A_c = {}^{H}\!A_c \backslash {}^{D}\!A_c$ *are the* non-local *attributes;*
5. ${}^{L}\!A_c = {}^{I}\!A_c \uplus {}^{D}\!A_c$ *are the* local *attributes;*
6. ${}^{A}\!A_c = {}^{I}\!A_c \uplus {}^{H}\!A_c = {}^{T}\!A_c \uplus {}^{P}\!A_c = {}^{L}\!A_c \uplus {}^{N}\!A_c$ *are the* applicable *attributes.*

To illustrate this terminology using our running example, for the concept `person` the hierarchical attributes include `name` which is total and non-local (being locally defined above in `agent`), `sex` which is total but local, being a dependent attribute of `person` itself, and `user_id` which is partial (necessarily non-local) for being local in the sub-concept `user`.

Total/partial attributes intuitively correspond to total/partial functions on the concept domain. We can locate any hierarchical attribute in the hierarchy.

Proposition 1. *In a canonical scheme, for every concept c there is a unique* location *mapping $\lambda_c : {}^{H}\!A_c \to \{\, c' \mid c' \sqsubseteq \overline{c} \,\}$ such that $a \in {}^{H}\!A_c \Rightarrow a \in {}^{D}\!A_{\lambda_c^a}$.*

Proof. By definition 5.1, for any $a \in {}^{H}\!A_c$ there is at least one $c' \sqsubseteq \overline{c}$ such that $a \in {}^{D}\!A_{c'}$. For any other $c'' \sqsubseteq \overline{c}$ we must have $\overline{c''} = \overline{c} = \overline{c'}$, and from condition 4 of the canonicity definition 4 we conclude that ${}^{D}\!A_{c''} \cap {}^{D}\!A_{c'} = \emptyset$, so $a \notin {}^{D}\!A_{c''}$. Therefore $\lambda_c^a = c'$.

3 Queries

In our interaction language NADI a query is expressed in a term, defining a database view yielding a (possibly empty) sequence of base value tuples. Two ways are available for retrieving such answers: either one at a time, through the backtrack-able call (`Query <? Tuple`), or all at once with the determinate (`Query <?> Tuple_list`). The term representation of tuples is simply

<tuple> ::= <value> | <value>,<tuple> [1]

We first introduce by examples the main features of NADI queries, followed by a formal semantic account of their core functionality.

3.1 Illustration of the main features

All conceptual expressions in NADI are built using a variety of infix and prefix operators, taking advantage of the flexible precedence definition mechanism (a standard feature of Prolog) to minimize the use of parenthesis.

As an example, the query for "the name and sex of the students enrolled in courses of the CS department in 2008/09" can be represented by a term whose main operators, besides the usual conjunction and equality, are the left-associative '`/`' ("of the") and the stand-alone '`$`' ("for which"):

[1] We can avoid lists because <value>s are of base types and not themselves lists.

```
( name, sex ) / student /
enrollment $ ( acronym/department/course = 'CS', year = 2009 )
```

The query consists of a *projection* `(name,sex)/student` of a *concept* `enrollment` under *constraints*. The form of the expression is close to its natural language counterpart, certainly much closer than the corresponding SQL statement

```
select distinct a1.name, p1.sex
from   enrollment e1, course c1, organization o1,
       student s1, agent a1, person p1,
where  e1.course = c1.code
   and c1.department = o1.code  and o1.acronym = 'CS'
   and e1.year = 2009           and e1.student = s1.code
   and s1.person = a1.code      and s1.person = p1.code
```

where many more concepts have to be made explicit, along with aliases and join equations. Attribute chains like "the acronym of the department of the course" are explicit in our conceptual expression and effective towards its readability, whereas they are scattered and therefore hidden in the SQL syntax.

The default ordering of answers is database-dependent. We can specify ascending or descending order on given projection attributes by prefixing them with `*>` or `*<`, respectively. Prefixing an intermediate projection attribute with `?` puts its identity in the projection tuple. In our example, the alternative projection `( *<sex, *>name ) / ?student` would ask for the descending sex, ascending name and identity code of the students.

By default the SQL `distinct` qualifier is applied to the selection, yielding distinct tuples, i.e. a set of answers. This can be overridden with the `??` prefix, e.g. `??grade / enrollment $ ( course_edition=CE )` retrieves the multiset of grades for a particular course edition (for computing a histogram, say).

A very useful feature in most applications is to have parameters under global assignment, whose current values are reflected in the queries through the linguistic equivalent of using the definite article. For example, the query for the students enrolled in *the* course edition (assumed to be contextually assigned) is `student/enrollment$(@course_edition)`. Its processing performs the call `course_edition=@CE` to retrieve the current value, and then uses the constraint `course_edition=CE`. Usage is very flexible for scheme-defined parameters, e.g. `course_edition=@CE` succeeds with `CE=(123,2009,s,1)` after the assignments `year@=2009`, `course@=123` and `period@=(s,2)`, and conversely `year=@Y` succeeds with `Y=2008` after `course_edition@=(95,2008,s,1)`.

Constraints under `$` can be grouped inside nested conjunctions and disjunctions. A common individual constraint is <attribute-chain> <op> <value>, with <attribute-chain> ::= <attribute> | <attribute-chain>/<attribute-chain> and appropriate <op>s (=, \=, >, etc.) and <value>s (@<parameter> is a value). The common case of identity valuation, `{}=V`, can be written simply as `{V}`.

Sometimes we need to express constraints using sub-queries instead of explicit values, as in "this year's lecturers that were also (our) students", doable with `lecturer/course_edition$(@year,lecturer^person/student)`. The operator `^` ("is a") assumes an attribute chain on the left but a query on the right, so `lecturer` is interpreted as an attribute but `student` as a concept.

The dual of `^` is `~` ("is not a"). Both exist also as prefix (rather than infix) operators (meaning respectively "there is a" and "there isn't a") over query expressions. But if we use e.g. `A$(···^B$X···)` we generally want, inside `X`, to relate attributes of `B` to attributes of `A`. For this we use the `*` prefix to move one level up in the context of attribute interpretation. As an example we can express "the students enrolled in only one course this year" with

```
student / enrollment$( @year, ~enrollment$( @year, *student,
                                            course \= *course )
```

where `*student` can be read as "the same student" and is just shorthand for `student = *student`, equating the student of the inner and outer enrollments.

Similar contextual sub-queries can appear in the projection part of a query, e.g. we get "this year's courses and the lecturers that ever taught them" with

```
(course,lecturer/course_edition$(*course))/course_edition$(@year).
```

Queries such as this, yielding for each course a number of lecturers, suggest the usefulness of packing the answer accordingly. This is achieved by using "`;`" instead of "`,`" where the grouping is needed. So, each solution of the alternative query `(course;lecturer···)/··· <? CL` binds `CL` to a term c`:[`l_1`,···,`l_n`]` with a course c and a list of lecturers l_i of c.

We can specify group projections using prefix operators for sum (`+`), average (`+/`) or count (`#`). For example, "the number of students enrolled in each course···" is simply expressed as `(#student,course)/enrollment$···`. The grouping is implicit in the regular projection elements, in this case `course`.

Another facility is the expression, with the usual syntax, of external functions available in the database engine (e.g. arithmetic, string and date functions). An example would be a query about enrollments in the previous year, expressed simply with `enrollment$(@year-1)`.

3.2 A formal account of query answers

Although queries really produce *sequences* of answers, our semantic account will be given in terms of *sets*, avoiding both the implicit and explicit ordering—an orthogonal issue with a standard meaning—and also the variant multiset reading—a straightforward adaptation from sets.

We can assume, without loss of generality, that our queries are of the form

<query> ::= <projection>/<constrained-concept>

<constrained-concept> ::= <concept>$<constraints>

All other queries are easily converted into this format, using the self attribute `{}` and the null constraint `[]`:

$$\begin{array}{rcl} c & \mapsto & \texttt{\{\}} \; / \; c \\ p \; / \; c & \mapsto & p \; / \; c \; \texttt{\$} \; \texttt{[]} \\ p \; / \; \texttt{?}c \; \texttt{\$} \; x & \mapsto & (\; p, \; \texttt{\{\}} \;) \; / \; c \; \texttt{\$} \; x \end{array}$$

Given a query $p/c\,\$\,x$, the semantic intuition is that the constraints x denote a Boolean function acting as a filter on the individuals of c, and the projection

p (for constrained c) a function projecting the filtered individuals into values of the projection tuple type, i.e. the answers. Formally,

$$^{Q}[\![\, p/c\,\$\,x \,]\!] = {}^{P}[\![\, p \,]\!]_c(\{\, t \in \Delta_c \mid {}^{F}[\![\, x \,]\!]^{t}_{c} \,\})$$

But in general queries can be nested, and there must be a notion of conceptual *context* parametrizing the semantics. Recall the example with `*course` used in a context with two instances of `enrollment`. So the equation above is just a particular case, with $^{F}[\![\, . \,]\!]$ parametrized on a single concept c and applying to a single tuple t. In general we must make both $^{Q}[\![\, . \,]\!]$ and $^{F}[\![\, . \,]\!]$ parametric on a *sequence* (stack) σ of concepts and applicable to a corresponding sequence τ of tuples, with the empty sequence ε as the base:

$$^{Q}[\![\, p/c\,\$\,x \,]\!] = {}^{Q}[\![\, p/c\,\$\,x \,]\!]^{\varepsilon}_{\varepsilon} \tag{1}$$
$$^{Q}[\![\, p/c\,\$\,x \,]\!]^{\tau}_{\sigma} = {}^{P}[\![\, p \,]\!]_c(\{\, t \in \Delta_c \mid {}^{F}[\![\, x \,]\!]^{t\tau}_{c\sigma} \,\}) \tag{2}$$

Constraints Let us consider the core sub-language of constraint expressions:

```
<constraints> ::=  <constraint>
                 | <constraints>;<constraints>
                 | <constraints>,<constraints>
<constraint> ::=  <attribute-chain><relation-op><value-exp>
                | <existential-op><query>
                | []
<attribute-chain> ::=  <attribute>
                     | <attribute-chain>/<attribute>
<relation-op> ::=   = | \= | > | < | >= | =< | <existential-op>
<existential-op> ::=  ^ | ~
```

As with queries, we assume that alternative simplified forms in the concrete NADI syntax are mapped into this core language. For example,

$$\texttt{\{<value>\}} \quad \mapsto \quad \texttt{\{\}=<value>}$$

For null, conjunction and disjunction the semantics is straightforward:[2]

$$\begin{aligned} ^{F}[\![\, \texttt{[]} \,]\!]^{\tau}_{\sigma} &= \top \\ ^{F}[\![\, x\,,y \,]\!]^{\tau}_{\sigma} &= {}^{F}[\![\, x \,]\!]^{\tau}_{\sigma} \wedge {}^{F}[\![\, y \,]\!]^{\tau}_{\sigma} \\ ^{F}[\![\, x\,;y \,]\!]^{\tau}_{\sigma} &= {}^{F}[\![\, x \,]\!]^{\tau}_{\sigma} \vee {}^{F}[\![\, y \,]\!]^{\tau}_{\sigma} \end{aligned}$$

The semantics of existential constraints is defined from the emptiness of the denotation (answer set) of the argument query.

$$\begin{aligned} ^{F}[\![\, \hat{}\,q \,]\!]^{\tau}_{\sigma} &= \left({}^{Q}[\![\, q \,]\!]^{\tau}_{\sigma} \neq \emptyset \right) \\ ^{F}[\![\, \tilde{}\,q \,]\!]^{\tau}_{\sigma} &= \left({}^{Q}[\![\, q \,]\!]^{\tau}_{\sigma} = \emptyset \right) \end{aligned}$$

The most common constraint is the one expressing a relation between the value of a *partial generalized attribute*, expressed by an <attribute-chain>, and

[2] We use $\top$ for the Boolean "true".

a *generalized value* expressed by <value-exp>. The generalized (by chaining) attribute is partial because it may not yield a value. The expression `user_id\=X` is a syntactically valid constraint under the concept `person`, but semantically a person may not have a user id, its identity being present in the `person` but not in the `user` database table. The constraint should be understood as "the user id exists and is different from `X`". To capture the partiality of a generalized attribute we formulate its semantics ${}^{A}[\![\,.\,]\!]$ as returning a set, which may be empty or a singleton. Actually larger sets may also be returned, as explained later, giving extra meaning to the "generalized" qualification. The semantics a relational constraint becomes existential, expressed as follows from the partial generalized attribute and generalized value semantics ${}^{A}[\![\,.\,]\!]$ and ${}^{V}[\![\,.\,]\!]$, with ${}^{R}[\![\,r\,]\!]$ the relation denoted by a <relation-op> r in the union of all applicable domains:

$$ {}^{F}[\![\,a\,r\,v\,]\!]^{t\tau}_{c\sigma} = \big(\, (\, {}^{A}[\![\,a\,]\!]^{t}_{c} \times \{\, {}^{V}[\![\,v\,]\!]^{t\tau}_{c\sigma} \,\}\,) \cap {}^{R}[\![\,r\,]\!] \neq \emptyset \,\big) \tag{3} $$

Notice that a filter demands a non-empty context, to apply a partial generalized attribute to (just) its topmost (current level) tuple for a concept. Generalized values may require the whole context, being possibly (contextual) sub-queries.

Partial generalized attributes An atomic <attribute-chain> a must be one of the applicable attributes ${}^{A}\!A_c = {}^{L}\!A_c \uplus {}^{N}\!A_c$ for the contextual concept c. If local ($a \in {}^{L}\!A_c$), its value is obtained by a simple (local) projection $\pi^{a}_{c}(t)$ of the contextual tuple $t \in \Delta_c$. If non-local ($a \in {}^{N}\!A_c$), it must be valuated through the projection $\pi^{a}_{c'}(t')$ of a (non-local) tuple $t' \in \Delta_{c'}$ for a hierarchically related concept $c' = \lambda^{a}_{c} \neq c$ satisfying an identity *join* $(\,\pi^{\{\}}_{c'}(t') = \pi^{\{\}}_{c}(t)\,)$ with the (local) tuple $t \in \Delta_c$; t' always exists for a total attribute ($a \in {}^{T}\!A_c$), but possibly not for a partial one ($a \in {}^{P}\!A_c$). Partial generalized attributes expressed by chains such as `acronym/department/course`, "the acronym of the department of the course", are (recursively) partial generalized attributes of concepts referenced by an applicable attribute, requiring the corresponding join—in this example $\pi^{\{\}}_{\texttt{organization}}(t') = \pi^{\textsf{department}}_{\textsf{course}}(t)$ to allow the final projection $\pi^{\texttt{acronym}}_{\texttt{organization}}(t')$. Partial generalized attributes are indeed partial since joins may fail.

We define partial generalized attributes ${}^{A}[\![\,.\,]\!]$ indirectly, through an auxiliary syntactic mapping γ, the *generalized attribute location*, from attribute chains and their base concepts to auxiliary terms of type <location>.

<location> ::= <attribute> | `@(`<attribute>`,`<concept>`,`<location>`)`

The *indirect* `@`-structured locations give rise to joins in the semantics. In a location `@(`a`,`c'`,`l`)` for a concept c, c' is either a direct attribute reference $r \in R^{a}_{c}$ or a super-concept ($r \sqsubset c'$) or sub-concept ($c' \sqsubset r$) thereof, with l a location for c'.

Here are some examples of γ mappings:

$$
\begin{aligned}
(\texttt{name},\ \texttt{agent}) &\mapsto \texttt{name}\\
(\texttt{name}, \texttt{person}) &\mapsto \texttt{@(\{\},agent,name)}\\
(\texttt{name/student}, \texttt{enrollment}) &\mapsto \texttt{@(student,student,@(person,agent,name))}\\
(\texttt{name}, \texttt{enrollment}) &\not\mapsto
\end{aligned}
$$

The first case is explained by `name` being a dependent, and therefore a local attribute of `agent`. In the second case `name` is not a dependent attribute of `person`, but it is a non-local hierarchical one; under our assumption of a canonical scheme, it has a unique location in a hierarchically related concept, in this case `agent`, joinable through the common identity. The last case shows two effects: sub-attribute inheritance, expressable in our scheme definition language NACS, that allows the attribute `name` to be applied to a `student` meaning the `name` of its `person`; and the chained location of that attribute in the root concept `agent` of the one (`person`) referenced by the attribute `person` in `student`. The γ mapping is naturally partial as e.g. `name` is not an applicable attribute of `enrollment`.

In order to define γ, please remember that condition 3 of the canonicity definition 4 imposes a unique root on an attribute's conceptual references. Since hierarchical attributes are defined from the root (definition 5.1), we can handle attribute references through a simpler alternative to R, namely the *root reference* mapping $\overline{R} : dom(A) \rightharpoonup C$ defined by

$$\overline{R}_c^a = r \quad \text{if} \quad \{\, \overline{x} \mid x \in R_c^a \,\} = \{r\}$$

The definition of γ is split into the base case, when the attribute chain is an atomic attribute, and the recursive case:

$$\gamma_c(a) = \begin{cases} a & \text{if} \quad a \in {}^L\!A_c \\ \mathtt{@(\{\},}\lambda_c^a\mathtt{,}a\mathtt{)} & \text{if} \quad a \in {}^N\!A_c \end{cases}$$

$$\gamma_c(a_2\mathtt{/}a_1) = \begin{cases} \mathtt{@(}a_1\mathtt{,}\overline{R}_c^{a_1}\mathtt{,}\gamma_{\overline{R}_c^{a_1}}(a_2)\mathtt{)} & \text{if} \quad \gamma_c(a_1) = a_1 \\ \mathtt{@(}a'\mathtt{,}c'\mathtt{,@(}a_1\mathtt{,}\overline{R}_{c'}^{a_1}\mathtt{,}\gamma_{\overline{R}_{c'}^{a_1}}(a_2)\mathtt{))} & \text{if} \quad \gamma_c(a_1) = \mathtt{@(}a'\mathtt{,}c'\mathtt{,}a_1\mathtt{)} \end{cases}$$

In our implementation the base case of γ is precompiled when processing the NACS database scheme, for direct use by the NADI interpreter. We also apply the following equivalence to minimize joins:

$$\mathtt{@(}a\mathtt{,}\overline{R}_c^a\mathtt{,@(\{\},}c'\mathtt{,}l\mathtt{))} \quad \equiv \quad \mathtt{@(}a\mathtt{,}c'\mathtt{,}l\mathtt{)}$$

Our goal of defining partial generalized attributes is realized in terms of a partial location semantics for the γ images, applicable to the contextual tuple of a given concept.

$${}^A[\![\, a \,]\!]_c^t \;=\; {}^L[\![\, \gamma_c(a) \,]\!]_c^t$$

For the location semantics we project local attributes and recursively switch to the tuple of the non-local concept whose identity joins with the local attribute.

$${}^L[\![\, a \,]\!]_c^t = \{\, \pi_c^a(t) \,\} \qquad \text{if} \quad a \in {}^L\!A_c$$

$${}^L[\![\, \mathtt{@(}a\mathtt{,}c'\mathtt{,}l\mathtt{)} \,]\!]_c^t = \bigcup_{t' \in \Delta_{c'}} \{\, {}^L[\![\, l \,]\!]_{c'}^{t'} \mid \pi_{c'}^{\{\}}(t') = \pi_c^a(t) \,\}$$

The somewhat dense formalization in the last definition hides the simple fact that at most one t' satisfies the required join, given the uniqueness of identity. The join may fail if c' is not a reference of a in c nor a super-concept thereof, i.e. $x \in R_c^a \Rightarrow x \not\sqsubseteq c'$, thereby yielding partiality (the empty set). An example is $\gamma_{\mathtt{student}}^{\mathtt{user\_id}} = \mathtt{@(person,user,user\_id)}$ with $R_{\mathtt{student}}^{\mathtt{person}} = \{\mathtt{person}\}$.

Generalized values Our syntax of relational constraints is asymmetric for pragmatic reasons, without sacrificing expressive power. The notion of generalized value, appearing as the right argument, includes (besides plain values) fully contextual attribute chains and (sub-)queries.

```
<value-exp> ::=  <value>
               | ?<attribute-chain>
               | *<contextual-attribute-chain>
               | <query>

<contextual-attribute-chain> ::=  <attribute-chain>
                                | *<contextual-attribute-chain>
```

In a constraint we can then relate two attributes of a concept, using the `?` prefix on the right, e.g. `name \= ?common_name`. The prefix is needed to disambiguate the interpretation of an atomic name as either a value or an attribute, on the right-hand side only—thus the asymmetry of the relational constraint.

The semantics for generalized values can now be stated.

$$\begin{aligned}
{}^{V}[\![\, v \,]\!]^{\tau}_{\sigma} &= v \qquad \text{if} \quad v : \text{<value>} \\
{}^{V}[\![\, ?a \,]\!]^{t\tau}_{c\sigma} &= {}^{A}[\![\, a \,]\!]^{t}_{c} \\
{}^{V}[\![\, *a \,]\!]^{tt'\tau}_{cc'\sigma} &= \begin{cases} {}^{V}[\![\, a \,]\!]^{t'\tau}_{c'\sigma} & \text{if} \quad a = *a' \\ {}^{A}[\![\, a \,]\!]^{t'}_{c'} & \text{otherwise} \end{cases} \\
{}^{V}[\![\, q \,]\!]^{t\tau}_{c\sigma} &= {}^{Q}[\![\, q \,]\!]^{t\tau}_{c\sigma} \qquad \text{if} \quad q : \text{<query>}
\end{aligned}$$

The contextual marker `*` signals the descent of one level in the dual context of concepts and their tuples, to interpret there (and not in the current level) the sub-expression. Queries can only appear under a <existential-op>, whose relational reading is ${}^{R}[\![\, \hat{} \,]\!] = (\in)$, ${}^{R}[\![\, \tilde{} \,]\!] = (\notin)$.

Projections Let us start by considering just standard projections without group operators, ordering or external functions. Their core syntax is

```
<projection> ::=  <attribute-chain>
                | <query>
                | <projection>,<projection>
```

Sub-queries in projections are supposedly contextual, as in the example about courses and lecturers towards the end of section 3.1. The concrete syntax allows conjunctions inside attribute chains, equivalent to their distribution over the chaining operator:

$$(a,b)/c \quad \mapsto \quad a/c,\ b/c$$

We may recall, from the semantic equation 2 for queries, that answers result from applying the denotation of the projection to the *set* of database tuples for the concept filtered by the constraints. The set argument is needed for the semantics of group operators (sum, max, etc.), but for standard projections the semantics is *flat*—obtained locally (as a set of tuples) from each tuple:

$${}^{P}[\![\, p \,]\!]_{c}(T) = \bigcup \{\, {}^{\Pi}[\![\, p \,]\!]^{t}_{c} \mid t \in T \,\}$$

We have seen the partial generalized attribute semantics attached to an <attribute-chain> in a constraint. Its projection semantics is basically similar, except that where the other is partial—returning the empty set—this one returns a singleton with a *null* tuple (`[]`, . . . , `[]`) of the appropriate size. This alternative encoding of partiality with nulls corresponds, in database parlance, to *outer joins* rather than *inner joins*. The query `(name,user_id)/student/enrollment$...` yields answers with the names of all the students in question along with their user id, if recorded in `user`, otherwise a null. A contextual sub-query in a projection is also linked through an outer join. The upshot is that *every* filtered tuple yields a projected answer tuple—exactly one if the projection consists only of strict[3] attribute chains, or possibly more if sub-queries are present, e.g. in the aforementioned example in section 3.1, where for each course we may get several lecturers. So here is the definition of our tuple projection semantics:

$$
\begin{aligned}
{}^{\Pi}[\![\, p_1 , p_2 \,]\!]^t_c &= {}^{\Pi}[\![\, p_1 \,]\!]^t_c \otimes {}^{\Pi}[\![\, p_2 \,]\!]^t_c && (\, A \otimes B = \{\, \alpha\beta \mid \alpha \in A,\, \beta \in B \,\}\,) \\
{}^{\Pi}[\![\, a \,]\!]^t_c &= {}^{L}[\![\, \gamma^a_c \,]\!]^t_c \cup \{\, \texttt{[]}^a_c \mid {}^{L}[\![\, \gamma^a_c \,]\!]^t_c = \emptyset \,\} && \text{if} \quad a : \textsf{<attribute-chain>} \\
{}^{\Pi}[\![\, q \,]\!]^t_c &= {}^{Q}[\![\, q \,]\!]^t_c \cup \{\, \texttt{[]}_q \mid {}^{Q}[\![\, q \,]\!]^t_c = \emptyset \,\} && \text{if} \quad q : \textsf{<query>}
\end{aligned}
$$

The null tuple mappings are $\texttt{[]}_{(a_1,\ldots,a_n)/c\$x} = \texttt{[]}^{a_1}_c \cdots \texttt{[]}^{a_n}_c$ and $\texttt{[]}^a_c = {}^{\texttt{[]}}[\![\, \gamma^a_c \,]\!]$ for a suitable adaptation ${}^{\texttt{[]}}[\![\, . \,]\!]$ of ${}^{L}[\![\, . \,]\!]$ to chase indirect locations even through failed joins and use the lexical arity of the final attribute for the null tuple.

The projection semantics for group functions is more involved. When asking for e.g. `(course,#student)/enrollment$(@year)` (how many students enrolled in each course this year) one must consider the equivalent classes of the `enrollment` tuples filtered under their `course` projection, which are the groups, then the application of the group function (count, in this case) to each group's `student` projection, and finally the concatenation of each projected course with its student count.

3.3 Derived attributes and universal quantification

The attribute names defined in the scheme correspond to (possibly partial) functions, and we've seen the corresponding attribute semantics ${}^{A}[\![\, . \,]\!]$ yielding singletons or the null set. However, in natural language certain nouns are used exactly like attributive ones but with a relational, rather than functional, meaning. An example is "daughter"; yielding possibly more than one value, when applied to a person, it must generally be used along with quantifiers: "one daughter. . . " or "all daughters. . . ".

Our framework allows for such *derived* (pseudo-)attributes, by naming conceptual expressions with outstanding contextual references to the target concept. If e.g. `person` has (standard) attributes `father` and `mother`, we may define

```
    child    =   person$( father= *{}; mother= *{} )
 daughter    =   person$( sex=f, ( father= *{}; mother= *{} ) )
```

[3] Not involving manifold attributes, explained next.

Now ${}^{A}[\![\,\texttt{child}\,]\!]^{t}_{\texttt{person}}$ can be a set with more than one member. Since the default interpretation of constraints is existential, as expressed in equation 3, the constraint `sex/child=f` is interpreted as "with (at least) one daughter". It becomes necessary to introduce extra syntax for signalling universal quantification; we do this by enclosing the (generalized) attribute chain in braces. So, we can ask for persons having no sons with `person$({sex/child}=f)`. Supposing `person` with an attribute `age`, and `son` defined analogously to `daughter`, we may ask `person$({age/daughter}>{age/son})` for those whose daughters are older than any son.

We have to qualify equation 3 with the restriction that a and v are not braced expressions, and define the universal cases, such as e.g.

$$ {}^{F}[\![\,\texttt{\{}a\texttt{\}}\; r\; \texttt{\{}a'\texttt{\}}\,]\!]^{t\tau}_{c\sigma} = \big(\, (\, {}^{A}[\![\,a\,]\!]^{t}_{c} \times \{\, {}^{A}[\![\,a'\,]\!]^{t}_{c} \,\}\,) \subseteq {}^{R}[\![\,r\,]\!] \,\big) $$

The `not` operator is also available for constraints, with the obvious semantics of Boolean negation, so that the last example above can be equivalently stated as `person$(not( age/daughter =< ?(age/son) ))`.

3.4 Manifold attributes

Most concepts and attributes in the real world have a temporal validity, i.e. their individual existence and values change with time. If this happens with periodic regularity, as with the course editions in an academic institution, the temporal structure is explicitly ascribed to attributes in the scheme (we've used `year`, `period`). Most of the temporal variability is, however, non-periodic—a user's identifier should be changeable at any time, as well as the name (e.g. through marriage). We may want to register such changes in the database, to be able to look at the evolution of things, but also wish to retain simplicity in most queries, as in saying e.g. "the student's user id" to mean the current (real-time) value.

Another prevalent variability is that of names, acronyms, etc. with language (e.g. English, French). Again, we ideally want a database scheme accommodating this multilingual variability but avoiding explicit mention of language in queries and commands where it can be assumed in context.

Achieving these goals is a highlight of our approach. We provide special notation in NACS, and the corresponding treatment in NADI, for what we call *manifold* attributes, whose value for a given individual may vary along a certain manifold dimension. Temporal and multilingual are the most common examples of such manifolds.

Temporal attributes Take the example of a user id. Some persons may never get one, while others may have different ones over time, not necessarily all the time. Given this, the temporal dimension is considered a *fluid* manifold. In our example `user_id` is no longer a local attribute of `user`, becoming an attribute of a different sort of concept, the *fluid manifolding* `agent_with_user_id`, whose identity is the relationship between agents and maximal validity time intervals for their user id's. The temporal manifold is a virtual concept with attributes `start` and `end` (with inclusive and exclusive reading, respectively) of type `date` (in our example) or `moment`. Since `start` fully identifies each (disjoint) interval, we end up with $A^{\{\}}_{\texttt{agent\_with\_user\_id}} = (\texttt{agent}, \texttt{start})$ and ${}^{D}A_{\texttt{agent\_with\_user\_id}} = \{\texttt{end}, \texttt{user\_id}\}$.

Queries abstract away from these implementation details. In fact, asking simply for `user_id/student/`$\cdots$ retrieves by default the *current* user id's for the students in question. NADI therefore exhibits upward compatibility [11] when transforming a regular into a temporal attribute. Added expressive power comes with the possible attachment of temporal constraints to temporal attribute occurrences in queries. NACS allows us to predefine shorthand for these, such as e.g. `(<T)` for `(end=<T)` or simply `T` for `(start=<T,end>T)`. An example of usage is `user_id~d(2005,12,31)/`$\cdots$ asking for id's valid at the end of 2005. The expression `user_id~[]` uses the empty constraint to denote *all* the user id's (at all times). This can occur in a projection to list them all, or e.g. in a constraint `user_id~[]=X` to assert that the user id at some point in time is `X`; remember the existential semantics of relational constraints expressed in equation 3.

The default constraint (applied when no other is available) is `now`, standing for the dynamically evaluated current time value (of type `moment` or `date`). It can be implicitly overridden by another contextual default, assigned to the global parameter `temporal_validity`. This is useful for avoiding explicit constraints in multiple queries under the same temporal validity assumption.

What about querying the temporality associated to values? We simply prefix the projected attribute, e.g. `~user_id/`$\cdots$ asks for the time interval associated with the current user id, followed by it, and `~user_id~[]/`$\cdots$ retrieves all the $(\texttt{start}, \texttt{end}, \texttt{user\_id})$ triples. The extreme temporal values are implementation-dependent, and generally understood as "unknown".

Multilingual attributes The multilingual manifold, contrary to the temporal, is *solid* rather than fluid. This is because a multilingual attribute always returns a value, either the variant for the required language or the base value that always exists (for some language). This behaviour allows the unbroken progressive adaptation of existing services to another language, as translation data gets filled in.

A multilingual attribute, in contrast to a fluid one, remains local in its concept, to hold the base value, along with the extra dependent attribute `language`. An example might be the name of a course. We get $\{\texttt{name}, \texttt{language}\} \in {}^{D}A_{\texttt{course}}$, and course name translations get stored in the auxiliary *solid manifolding* concept `course_ml`,[4] with $A^{\{\}}_{\texttt{course\_ml}} = (\texttt{course}, \texttt{language})$ and ${}^{D}A_{\texttt{course\_ml}} = \{\texttt{name}\}$. A database view `course_ml_` is also created, as a virtual super-type of `course_ml`, to achieve the desired effect of defaulting to the base value for `name` in `course` when the translation for $(\texttt{course}, \texttt{language})$ is not in `course_ml`.

Where the `~` operator was used in fluid attribute expressions, we use `^` for solid ones. The constraints are just values for the language, the generic default being application-dependent, and implicitly overridden by a value assigned to

[4] `ml` stands for multilingual.

the global parameter `language`. So, for example, `name^en/course` might refer to the course name in English, also accessed simply by `name/course` in a context where `language=@L` yields `L=en`. Through `^name^[]/course` we get all (`language,name`) pairs, and `name^{}/course` refers to the base name, recorded in the `course` database table.

Manifold combination Manifolds can be combined. If we define the `name` of an `organization` as being both temporal and multilingual, we may then ask `name~d(2005,12,31)^en/department`··· for a department's name at the end of 2005 in English, or `~^name^{}/department`··· for the validity and base language of the (and) current name of the department.

Manifold formalization Lacking the space to fully develop the formal account of manifold attributes, we may nevertheless hint at what it takes to do so. The generalized attribute location γ has to cater for new base cases regarding fluid and solid manifold attributes. In the `user_id` example we would have e.g.

$$\gamma_{person}(\texttt{user\_id}) = \texttt{agent\_with\_user\_id\~{}user\_id}$$

This case would be handled by the location semantics with

$$^{L}[\![\, c'\tilde{}\,a \,]\!]^{t}_{c} = \{\, \pi^{a}_{c'}(t') \mid t' \in \Delta_{c'},\ \pi^{\bar{\{\}}}_{c'}(t') = \pi^{\{\}}_{c}(t) \,\}$$

where the *base* identity $\bar{\{\}}$ of a manifolding c' refers to the manifolded concept c. In the example, $A^{\bar{\{\}}}_{\texttt{agent\_with\_user\_id}} = \texttt{agent}$, a super-type of `person`. Actually, the given equation reflects just the case with no manifold constraints. In general we have to consider an extra argument in the location semantics, to carry the abstract constraint to be imposed on t' alongside the identity (join) constraint. Notice that we can refer in the same query to different variants, under appropriate constraints, of the same manifold attribute, e.g. the names before and after a given date.

4 Commands

The major commands are for the *creation*, *deletion* and *update* of concept members, expressed respectively with

```
+ <constrained-concept>
- <constrained-concept>
<constrained-concept> -+ <equalities>
```

These procedures are much more powerful than their strict SQL counterparts. Wishing to create a person, for example, we naturally need to specify the values of its total attributes (except its automatically incremented `id_code` identity) but not the actual concepts in the hierarchy where they are located, and we can add attributes for implied sub-concepts of person. For instance, the call

```
+ person$( sex=f, name=N, common_name=CN, user_id=I )
```

will split the specified attributes into their locations agent (name, common_name), person (sex) and user (user_id),[5] generate (from a database sequence) the common identity id_code=IC, and insert a new tuple for each database table, in their hierarchical order. If user_id is a temporal attribute, the very same command still works fine, inserting a tuple in agent_with_user_id with the provided user_id and the default values for start and end, which happen to be their respective extreme values (thus meaning forever). For a multilingual attribute the static default value for language would be used. The defaults can be overridden with the same syntax used in queries to attach explicit manifold constraints.

We can take much advantage of parameters. Imagine the logic programming code for a service on the Web for students to enroll in courses. Upon its invocation the parameters student, year, etc. have been assigned values, and when the student clicks on a course its code is transmitted and assigned to course, with nothing else needed but to call

```
+ enrollment$( @student, @course_edition ).
```

Equally powerful is the update command. We can simply use, for example,

```
@student -+ ( common_name=N )
```

to update the student's common name in the agent tuple whose id_code is that of the student's person attribute. This comes through the implementation of $\lambda^{\texttt{identifier}}_{\texttt{student}}$ that finds the sub-attribute inheritance of common_name from person and its location at agent $= \overline{\texttt{person}}$.

Manifold attributes require notions of update going beyond simple equalities. For example, the introduction of a new translation for the name of a course is expressed as an update for the course such as @course-+(+name^fr=FN), that actually inserts a new tuple in the course_ml translation table.

The true power of updates, however, is revealed when updating an identity attribute. This change in an individual's identity has to be propagated all over the database tables where the individual is present. In order not to violate foreign keys, this has to performed by creation and deletion, instead of immediate update, and in the correct order when chasing dependencies. The compiled scheme has all the information for this reasoning to be performed flawlessly, which in some cases would be a daunting task if done manually. If, for example, there ever was a reason to change the identity code of a course, the deceptively simple call @course-+(id_code=C) would do it, and in the end the course identity would have changed in course, course_edition, enrollment, etc. without ever having violated the database keys during the complex operation.

Deletion is similarly powerful, but presents more occasion for ambiguity and inconsistency. Deleting a concept instance implies, much as for identity updates, to chase dependencies in order to eliminate the instance from the database. But while this is sound for sub-concepts and derived concepts (those where it is just part of the identity), what about super-concepts and dependent concepts (those where it is a dependent attribute)? In both cases we may have a strong or weak reading of the deletion, respectively deleting or not the super-instance or derived

[5] Assuming that no attribute is manifold.

concept. If the strong reading might make sense for the super-concept (delete the individual, not just the fact that it belongs to a class in the hierarchy), for derived concepts it is hard to justify (say, delete enrollments with a peculiar grade that is being deleted). So we opt for the weak reading, resulting in (exception) failure if any of the hard cases happens.

Speaking of failure, it should be mentioned that we provide contextual transactions (`db_transaction:Exec`) that can be nested. Actually, any command raises an exception if not called under a transaction.

5 Summary of achievements and further work

We have shown how to interact with relational databases using a vastly more effective language than SQL, following natural language principles of noun phrase composition with attribute chains and contextual definite references. The natural and concise character of our queries and commands is a match for other approaches based on deductive and/or object-oriented paradigms, with higher impact on the readability of code and ease of its change, and the bonus of requiring only standard and mature relational database technology.

In this paper we exhibited precise formal semantics for the core features of our query language, based on a formal model of canonical schemes and corresponding databases, with our attributive syntax translating into implicit inner and outer joins. Useful features include definite references ("the") to contextual entities, local or global. The abstraction power is much enhanced with a generic treatment of manifold phenomena such as temporal and multilingual attributes, incorporated in the scheme definition and database interaction languages, resulting in very powerful effects with little effort. Commands can be very high-level, with concise statements possibly resulting in large transactions.

Complying with the formal semantics, the NADI interaction language has been implemented with high reliance on the distinguishing features of logic programming, namely structural unification and implicit backtracking, to reason over a compiled version of a database scheme. This compilation is itself a deductive task carried out by a logic program, over a scheme description in the NACS language (described elsewhere [2]) that exploits inheritance and name composition to achieve also a high level of conciseness and readability. The architecture is available to developers of Prolog applications through the scheme compiler, the resulting on-line scheme documentation, and a few interface predicates for queries and commands. A team of around ten people has used it for years to build a very large real-world academic management system [1].

There are several directions for improving and building on this work. One is to embed the NADI interaction language in other programming languages through libraries capable of calling Prolog. Performance-wise we can achieve static optimizations of the code by partial evaluation, which should be quite manageable since we are using a purely compositional alternative to Prolog. It is important also to cope with multiple databases and schemes in the same application. A different and much greater challenge, of vast importance, is to tackle the problem of scheme change. This is a fact of life for most real-world applications, and generally a nightmare for the software development teams. One will have to jump from a purely static view of a scheme to a much higher-level plane where to express the *process* of scheme change rather than purely its *result*, and use this to deduce the impacts, and proceed with the necessary changes, on both the current relational database structure and the database interaction code.

At another level of effort it would be interesting to promote these languages as complementary to the current dogmatic choice of languages for the semantic Web, as we believe there is a misguided misconception of what is "content", and an approach favouring a question-answering paradigm of how to acquire useful information, tapping on the immense potential of existing relational databases, is no less adequate than the idea of extracting "knowledge" as structured data to be reasoned upon. One can easily conceive of a site publishing the scheme of (part of) its relational database, in NACS or its compiled form, and allowing clients to submit corresponding NADI queries to Web services providing answers.

References

1. Porto, A.: An integrated information system powered by Prolog. In Dahl, V., Wadler, P., eds.: Practical Aspects of Declarative Languages, 5th International Symposium, Proceedings. Volume 2562 of Lecture Notes in Computer Science., Springer (2003) 92–109
2. Porto, A.: High-level interaction with relational databases in logic programming. In Gill, A., Swift, T., eds.: Practical Aspects of Declarative Languages; 11th International Symposium, PADL 2009; Savannah, GA, USA, January 2009; Proceedings. Volume 5418 of Lecture Notes in Computer Science., Springer (2009) 152–167
3. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer (1990)
4. Liu, M.: Deductive database languages: problems and solutions. ACM Comput. Surv. **31**(1) (1999) 27–62
5. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. Journal of the ACM **42**(1) (July 1995) 741–843
6. Niemi, T., Järvelin, K.: Prolog-based meta-rules for relational database representation and manipulation. IEEE Transactions on Software Engineering **17**(8) (1991) 762–788
7. Lenzerini, M., Schaerf, A.: Concept languages as query languages. In: Proceedings of the 9th National Conference on Artificial Intelligence, AAAI Press / The MIT Press (1991) 471–476
8. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook - Theory, Implementation and Applications. Cambridge University Press (2003)
9. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. (2004) http://www.w3.org/TR/owl-guide/.
10. Antoniou, G., van Harmelen, F.: A Semantic Web Primer. Second edn. The MIT Press (2008)
11. Bair, J., Bölen, M.H., Jensen, C.S., Snodgrass, R.T.: Notions of upward compatibility of temporal query languages. Wirtschaftsinformatik **39**(1) (1997) 25–34