Using a Logic Programming Language with Persistence and Contexts

Salvador Abreu and Vitor Nogueira

Universidade de Évora and CENTRIA FCT/UNL, Portugal
{spa,vbn}@di.uevora.pt

Abstract. This article merges two approaches: one dealing with persistence for logic programs, as provided by a relational database back-end and another which addresses the issues of logic program structuring, by way of the parametric context. We claim that these two can be effectively combined to obtain a language which offers significant gains in expressiveness over previous work. This claim is experimentally backed by the applications that have been developed using these tools.

1 Introduction

Contexts: The idea of Contextual Logic Programming (CxLP) was introduced in the late 1980s by Monteiro and Porto [7] and is related to similar efforts such as that by Miller described in [6]. The purpose of CxLP was initially to deal with Prolog's traditionally flat predicate namespace, which seriously hindered its usability in larger scale projects. The impact of these extensions has mostly failed to make it back into the mainstream language, as the most widely distributed implementations only provide a simple, SICStus-like module mechanism, if any.

A more recent proposal [3] rehabilitates the ideas of Contextual Logic Programming by viewing contexts not only as shorthands for a modular theory but also as the means of providing dynamic attributes which affect that theory: we are referring to unit arguments, as described in Abreu and Diaz's work. It is particularly relevant for our purposes to stress the *context-as-an-implicit-computation* aspect of CxLP, which views a context as a first-class Prolog entity – a term, behaving similarly to an object in OOP languages.

Persistence: having persistence in a Logic Programming language is a required feature if one is to use it to construct actual information systems; this could conceivably be provided by Prolog's internal database but is best accounted for by software designed to handle large quantities of factual information efficiently, as is the case in relational database management systems. The semantic proximity between relational database query

languages and logic programming languages have made the former privileged candidates to provide Prolog with persistence, and this has long been recognized.

ISCO [2] is a proposal for Prolog persistence which includes support for multiple heterogeneous databases and which access to technology beyond relational databases, such as LDAP directory services or DNS. ISCO has been successfully used in a variety of real-world situations, ranging from the development of a university information system to text retrievalor business intelligence analysis tools.

ISCO's approach for interfacing to DBMSs involves providing Prolog declarations for the database relations, which are equivalent to defining a corresponding predicate, which is then used as if it were originally defined as a set of Prolog facts. While this approach is convenient, its main weakness resides in its present inability to relate distinct database goals, effectively performing joins at the Prolog level. While this may be perceived as a performance-impairing feature, in practice it's not the show-stopper it would seem to be because the instantiations made by the early database goals turn out as restrictions on subsequent goals, thereby avoiding the filter-over-cartesian-product syndrome.

Contexts and persistence: considering that it is useful to retain the regular Prolog notation for persistent relations which is an ISCO characteristic, we would like to explore the ways in which contexts can be taken advantage of, when layered on top of the persistence mechanisms provided by ISCO. In particular we shall be interested in the aspects of common database operations which would benefit from the increase in expressiveness that results from combining Prolog's declarativeness and the program-structuring mechanisms of Contextual Logic Programming.

We shall illustrate the usefulness of this approach to Contextual Logic Programming by providing examples taken from a large scale application, written in GNU Prolog/CX, our implementation of a Contextual Constraint Logic Programming language. More synthetic situations are presented where the constructs associated with a renewed specification of Contextual Logic Programming are brought forward to solve a variety of programming problems, namely those which address compatibility issues with other Prolog program-structuring approches, such as existing module systems. We also claim that the proposed language and implementation mechanisms can form the basis of a reasonably efficient development and production system.

The remainder of this article is structured as follows: section 2 recapitulates on ISCO, while section 3 presents our revised specification of Contextual Logic Programming, stressing the relevance of unit arguments. In section 4 we show some uses for a unified language, which uses both persistence and contexts. Finally, section 5 draws some conclusions and attempts to point at unsolved issues, for further research.

2 Persistence with ISCO

Is is not the purpose of this article to introduce the ISCO language, but a short review of its main features is useful for the discussion ahead.

ISCO has been described in [2] and comes as an evolution of the Logic description language presented in [1]. It is a mediator language in that an ISCO program may transparently access data from several distinct sources in a uniform way: all behave as regular Prolog predicates. Some relevant advantages ISCO holds over competing approaches are its ability to concurrently interface to several legacy systems, its high performance by virtue of being derived from GNU-Prolog and its simplicity.

Predicate declarations: predicates that are to be associated with an external representation must be declared. This is necessary because DBMSs need to have table fields *named* and *typed* and none of this information is derivable from a regular Prolog predicate.

Example 1 (ISCO class teacher).

```
class teacher.
    name:       text.
    department: text.
    degree:     text.
```

A class `teacher` can be declared in ISCO as in example 1. This defines predicate `teacher/3`, which behaves as a database predicate but relies on an external system (e.g. an RDBMS) to provide the actual facts.

Class declarations in ISCO may reflect inheritance, although that isn't shown in the previous example. Several features of SQL have been mapped into ISCO: keys and indexes, foreign keys and sequences to name a few.

Operations: classes in ISCO stand for *persistent predicate declarations*. The way these may be used is similar to what is done in Prolog for "database" predicates:

– non-deterministic sequential access to all clauses,
– insertion of new facts (ala `assertz/1`),
– removal of specific facts (ala `retract/1`)

The use of relational database back-ends spurred the adoption of an "update-like" operation in ISCO, which has no traditional Prolog counterpart.

These operations may specify constraints on their arguments to limit the tuples they apply to. These may be actual CLP(FD) constraints or more specific syntactic constructs, designed to provide a minimal yet useful set of features to tap into the potential efficiency provided by the RDBMS: for example, there are notations to specify solution ordering or substring matching.

Outlook: ISCO-based logic programs access the RDBMS-resident relations one-at-a-time, i.e. each predicate maps directly to queries onto the corresponding table. This results in joins being performed at the Prolog level, which may impair efficiency in some cases. This cannot be easily dealt with, as calls to class predicates may be mixed with regular Prolog goals, so that it's hard to determine beforehand what compound SQL query could be generated to effect a proper database-side join. Using Draxler's Prolog-to-SQL compiler[4] could improve this situation somewhat, but it has yet to be done.

3 Overview of Contextual (Constraint) Logic Programming

Contextual Logic Programming (CxLP) [7] is a simple yet powerful language that extends logic programming with mechanisms for modularity. In CxLP a finite set of Horn clauses with a given name is designated by *unit*. In [3] we presented a new specification for CxLP, which emphasizes the OOP aspects by means of a stateful model, allowed by the introduction of unit arguments. Using the syntax of GNU Prolog/CX, consider a unit named `teacher` to represent some basic facts about the teaching at a University:

Example 2 (CxLP unit teacher).

```
:-unit(teacher).

name(N)       :- teacher(N, _, _).
department(D) :- teacher(_, D, _).
```

```
degree(D)     :- teacher(_, _, D).

teacher(john, cs, phd).
teacher(bill, cs, msc).
```

The only difference between the code of example 2 and a regular logic program is the first line that declares the unit name. Consider also another unit to represent information about courses, in example 3:

Example 3 (CxLP unit course).

```
:-unit(course).

teacher(N) :- course(N, _).
course(C)  :- course(_, C).

course(john, ai).
course(bill, lp).
```

A set of units is designated as a *contextual logic program*. With the units above we can build a program `P = {teacher, course}`. If we consider that `teacher` and `course` designate sets of clauses, then the resulting program is given by the union of these sets.

For a given CxLP program, we can impose an order on its units, leading to the notion of *context*. Contexts are implemented as lists of unit designators and each computation has a notion of its *current context*. The program denoted by a particular context is the union of the predicates that are defined in each unit. We resort to the *override semantics* to deal with multiple occurrences of a given predicate: only the topmost definition is visible.

To construct contexts, we have the *context extension* operation given by the operator `:>`. The goal `U :> G` extends the *current context* with unit `U` and resolves goal `G` in the new context. For instance, to find out about the academic qualification of the person who taught the Logic Programming course (lp), we could ask:

```
teacher :> course :> (course(N, lp), teacher(N, _, DEG))
```

In this goal, we start by extending the initially empty (`[]`) context with unit `teacher`, obtaining context `[teacher]`. This context is again extended with unit `course`, yielding the context `[course, teacher]`, and it is in the latter context that goal `course(N, lp), teacher(N, _, DEG)` is derived.

3.1 Units with arguments

In [3] we add *units arguments* as a significant part of the CxLP programming model: a unit argument can be interpreted as a "unit global" variable, i.e. one which is shared by all clauses defined in the unit. Unit arguments help avoid the annoying proliferation of predicate arguments, which occur whenever a global structure needs to be passed around. For instance, the `teacher` unit could be rewritten as in example 4:

Example 4 (CxLP unit teacher with arguments).

```
:- unit(teacher(NAME, DEPARTMENT, QUALIFICATIONS)).

name(NAME).
department(DEPARTMENT).
qualifications(QUALIFICATIONS).

teacher(john, cs, phd).
teacher(bill, cs,  msc).

item :- teacher(NAME, DEPARTMENT, QUALIFICATIONS).
```

In this modified version, we have three unit argument `NAME`, `DEPARTMENT` and `QUALIFICATIONS`, along with three unary predicates to *access* these arguments. There is a new predicate `item/0` that instantiates all unit arguments using facts from the database.

To answer the same question as before, and considering a similar change was done to unit `course` we could say:

```
course(N, lp) :> item, teacher(N, _, DEG) :> item.
```

The way in which this query works is substantially different, though, because we are now instantiating variables which occur in the units, not in the goals. In fact, the goals are now bare.

3.2 Contextual Constraint Logic Programming

It was a natural thing to try to combine contextual and constraint logic programming, yielding what we call *Contextual Constraint Logic Programming* or CCxLP. This paradigm subsumes CLP [5] and enriches CxLP, by allowing unit arguments and contexts to be constrained variables.

Considering again the GNU Prolog/CX implementation, where the CLP scheme is particularized to Finite Domains (CLP(FD)) and reified

Booleans (CLP(B)), we can define the unit of example 5 to represent the 24–Hour timekeeping system:

Example 5 (CxLP unit time).

```
:- unit(time(HOUR, MINUTE, SECOND)).

args :- fd_domain(HOUR, 0, 23),
        fd_domain(MINUTE, 0, 59),
        fd_domain(SECOND, 0, 59).

hour(HOUR).
minute(MINUTE).
second(SECOND).
```

Using unit `time` to express the 9-to-5 working hours, we can simply do:

```
time(T) :> (args, hour(H), H #>= 9, H # =< 17).
```

which will result in variable `T` having variable subterms which are constrained variables which correspond to the time period in question. This sort of situation is explored in more depth in [8].

4 Contexts in ISCO

The University of Evora's Integrated Information System, SIIUE, is implemented in GNU Prolog/CX [3] and ISCO [2]. It relies on a relational database management system – PostgreSQL in the occurrence – for which an efficient, although very low-level, interface has been provided. We selected this approach rather than, for instance, resorting to the Draxler Prolog-to-SQL query generator [4] because the database is totally generated from within Prolog, both schema and data.

One "selling point" on ISCO is that modeling and implementation of information systems and related applications should be performed directly at the Logic Programming level, so we also want to hide the fact that there is an underlying relational database at all: it's only being used as a persistence provider.

Following an "extreme-contextual" approach, it was decided by design that every predicate which represents a problem-domain relation would also have an associated unit, of the same name and arity according to the number of arguments in the relation. This would hold whether the predicate is implemented as a set of facts backed by the persistency mechanism, or as a regular predicacte with non-fact clauses, as would be the case for a computed relation.

Taking up the definition of example 1 (page 3), class `teacher` should now compile to code like that of example 6:

Example 6 (CxLP & ISCO unit teacher with arguments).

```
:- unit(teacher(NAME, DEPARTMENT, QUALIFICATIONS)).

name(NAME).
department(DEPARTMENT).
qualifications(QUALIFICATIONS).

item :- teacher(NAME, DEPARTMENT, QUALIFICATIONS).

teacher(A, B, C) :- <<SQL ACCESS CODE>>
```

The point is that all the code in this unit is generated by the ISCO compiler, which will include yet a few more predicates, in particular those required to deal with insertion, removal and tuple updates.

Following the CxLP approach, constrained queries to persistent relations are specified by constructing a *context* that constitutes a complete specification of the query: this context can be viewed as an implicict computation, in which the goal `item` will activate the intended query.

Example 7 (CxLP & ISCO constraing unit).

```
:- unit(where(CONDITION)).
item :- CONDITION, :^ item.
```

Example 7 shows how a context component – a unit – may be used to impact the meaning of the `item` goal: it retains the semantics provided by the remainder of the context, after being put in a conjunction with an arbitrary goal, specified as a unit argument to unit `where/1`. The `:^` operator is analogous to the `super` keyword in OO languages. An example of using this approach:

```
teacher(N,D,Q) :> where(member(Q, [bsc, msc]))
               :> :< TAs,
TAs :< item
```

The first goal binds variable `TAs` to a context which will generate all teachers which don't hold a PhD. degree. This is achieved with the solutions for the `item` predicate.

The `:<` unary operator unifies its argument with the *current context* while the similarly named binary operator effects a *context switch*.

In spite of its simplicity, this example is representative of the way programs are coded in ISCO and GNU Prolog/CX, with contexts representing generators being built and saved in a variable in which the `item` goal is subsequently evaluated: this can be thought of as the creation and storage of an object, followed by sending of messages to that object.

An Application: a simple yet illustrative example is that of maintaining a browser session. A session is a sequence of related web pages, pertaining to a specific interaction, as identified by user input. Sessions are represented by server-side information which persists across individual pages and is referenced by a client-side (CGI) variable.

A system such as Universidade de Évora's SIIUE [1] requires sessions. In order to provide session support using ISCO and GNU Prolog/CX, we resorted to a thin PHP layer, which collects all variables, be they provided in the URL, in a POST, as cookies or as a server-side variable. These are then supplied to the GNU Prolog/CX program as a unit, cgi/1, which provides predicates to access and enumerate variables.

The sequence of web pages is represented and managed using a sequence of contexts, each representing what may be done at a particular stage. The way a particular link which advances the session is represented is quite simple, as it only requires that:

- the current context be available, as a session variable,
- the link specify the next context, possibly as an extension to the current one,
- every such context should behave uniformly, allowing for the same predicate to render the HTML code.

5 Conclusions and Directions for Future Work

ISCO and GNU Prolog/CX have proven to be a good match, as they both provide useful extensions to Prolog, in particular if what we have in mind is the construction of complex information systems. This is certainly the case for SIIUE, which – at the time of this writing – total about 38000 lines of GNU Prolog/CX code in over 400 units, the persistent database currntly sports over 8 million tuples in 200 relations, a couple of which having over 1 million tuples. SIIUE is daily operated on by several thousand users.

This system has already been applied in a variety of situations, beyond the one that spurred its development, SIIUE. Theq range of application domains is incessantly growing. There are very promising developments

in the fields of business intelligence and information retrieval, for which prototype systems have already been implemented.

One aspect which remains challenging is the development of a particular application with ISCO and GNU Prolog/CX, from scratch. We are presently looking into ways of integrating Logic Programming-based tools, such as ISCO and CxLP, into Content-Management Systems such as Plone or Mambo. It is our belief that these CMS will benefit from having plug-ins with the expressiveness of Prolog.

At present, ISCO joins are always performed on the Prolog side, thereby throwing away good opportunities for query optimization on the DBMS server side. We are presently working on this issue, and will possibly resort to Christoph Draxler's [4] Prolog-to-SQL compiler, although the task is made more difficult because ISCO allows for CLP(FD,$\mathcal{B}$) constraints to apply to variables.

Another approach which we are presently exploring involves associating more than just GNU Prolog's native FD constraints to query variables, by making use of an attributed-variable extension to GNU Prolog.

References

1. Abreu, S., *A Logic-based Information System*, in: E. Pontelli and V. Santos-Costa, editors, *2*$^{\text{nd}}$ *International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, Lecture Notes in Computer Science **1753** (2000), pp. 141–153.
2. Abreu, S., *Isco: A practical language for heterogeneous information system construction*, in: *Proceedings of INAP'01* (2001).
3. Abreu, S. and D. Diaz, *Objective: in Minimum Context*, in: C. Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, Lecture Notes in Computer Science **2916** (2003), pp. 128–147, iSBN 3-540-20642-6.
4. Draxler, C., *A Powerful Prolog to SQL Compiler*, Technical Report 92–61, Centre for Information and Language Processing, LudwigsMaximillians-Universität München (1992).
5. Jaffar, J. and M. Maher, *Constraint Logic Programming: a Survey*, The Journal of Logic Programming **19/20** (1994).
6. Miller, D., *A logical analysis of modules in logic programming*, The Journal of Logic Programming **6** (1989), pp. 79–108.
7. Monteiro, L. and A. Porto, *Contextual logic programming*, in: G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming* (1989), pp. 284–299.
8. Nogueira, V., S. Abreu and G. David, *Towards Temporal Reasoning in Constraint Contextual Logic Programming*, in: P. H. et al., editor, *Proceedings of the 3rd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'04*, TU Berlin, 2004, pp. 119–131.