On Structuring Contextual Logic Programs

Salvador Abreu[1] Vitor Nogueira[1] Daniel Diaz[2]

[1] Universidade de Évora and CENTRIA, Portugal
{spa,vbn}@di.uevora.pt
[2] Université de Paris-I, France
Daniel.Diaz@univ-paris1.fr

Abstract Standardization for Prolog came during the 1990's, initially and deliberately leaving out one aspect which is essential for real world application development: the modularity mechanism. This situation has in the meantime been remedied in the current ISO proposal for modules in Prolog.
In this article we build on our previous work on Contextual Logic Programming (CxLP) and introduce mechanisms which provide much needed functionality: on one hand, a stricter specification for acceptable program structure when using contexts and, on the other, a mechanism which more effectively promotes OO-style code reuse and concealment, while retaining a lightweight syntax.

1 Introduction

Contextual Logic Programming was proposed by Monteiro and Porto [5] as a means of bringing modularity to the Prolog language. More recently, it was reintroduced as a practical extension to the Prolog language by Abreu and Diaz [1], aiming to provide a program structuring mechanism as well as fulfill some of Prolog's shortcomings when used for programming in-the-large, namely by enabling an object-oriented programming style without relinquishing the expressiveness of Logic Programs.

For their dynamically configurable structure, contexts clearly subsume the base mechanisms of most module systems, in particular the ISO/IEC Prolog Modules standard proposal [4]. This strength hides a weakness: the potentially dynamic nature of a context leads to severe difficulties in predicting its value at compile time, particularly when taking a separate compilation approach, a situation that leads to the postponement until runtime of several checks which could otherwise be performed at compile time. In section 2 we propose a mechanism which can help keep context structure in control. At the other end of the spectrum, when dealing with contexts made up of a large number of distinct units (components), when using units as (functional) building blocks for a context such as when using contexts to model ontologies, having a flat namespace for units can quickly become a hurdle. Section 3 outlines a mechanism which can overcome this limitation and still retain the notational simplicity of CxLP.

2 Context Structure

One issue with Contextual Logic Programming is the freedom it provides the programmer with, which is sometimes perceived to be excessive: the lack of compile-time knowledge about the actual runtime structure of the context has consequences over the possibility of certain types of program analysis, as it is impossible to tell which predicates will be available. Indeed, freely configurable contexts, combined with Prolog's typelessness and meta-programming practices, can create a static analyzer's nightmare. From a language design viewpoint, we wish to avoid predicate and interface declarations, in order to keep CxLP syntactically compact.

2.1 Defining the Structure of the Context

The *context search* mechanism [1] is a basic Contextual Logic Programming mechanism. It also happens to be the origin of the issues that plague Contextual Logic Programming from a static program analysis point of view: it is impossible, in the general case, to know beforehand what units will actually constitute the runtime context; therefore the available set of predicates is also unknown.

The definition of a predicate is given by the clauses in the topmost unit that defines it (i.e. the "override" semantics). This is unknown at compile time because the compilation "unit" is the CxLP *unit*, not the *context*. The actual runtime context in which the present unit will occur may include an arbitrary set of units with the correspondingly arbitrary set of predicate definitions.

A workaround for this problem was proposed in [1] by means of the `:>>` operator which allows for a particular unit to specify a predicate (`context/1`) which will dictate what actual context it will be used, at runtime.

In pursuing simplicity and compactness, we relocate this notion of structure into the unit directive itself. Instead of resorting to the `context/1` hack, the directive simply specifies a prefix[1] of the context in which the unit can appear.

As an illustration, consider the units `person(NAME, ADDRESS, BIRTHDATE)` and `student(ID, PROG)` to represent the concepts that a *student* "is-a" *person*, meaning it should inherit everything a person has whilst extending it with new attributes (in this case, an ID number and a study program code.) Using the extended directive to express such concepts we'll have the following unit declarations:

```
:- unit person(NAME, ADDRESS, BIRTHDATE).
:- unit student(ID, PROG).person(NAME, ADDRESS, BIRTHDATE).
```

Notice that this notation is backward compatible, i.e. the declaration for unit `person/3` looks the same as previously.

We explicitly state the prefixes of the contexts in which the goals of these units can occur: any goal of unit `student/2` is evaluated knowing that the current context starts with `[student(_, _), person(_, _, _), ...]`.

[1] The prefix starts with the current unit, looking at the context as a stack, headed by its topmost element.

Finally, this declaration allow us to have direct access to the unit parameters of unit `person/3` from inside unit `student/2`. The justification for this programming technique is simple, from an OOP point of view: proper inheritance may require sharing of the "superclass" instance variables.

3 Scoped Units

Most modern programming languages provide layered modularity mechanisms in which visibility is controlled. Take for example Java packages or XML namespaces. In the case of CxLP, units are a program-structuring component which is combined to form contexts. The simple CxLP specification does not effectively account for accidental naming collisions, such as would happen with unit names in situations where there are many of them.

To address this situation, we introduce a new mechanism: *local units*, for which the unit name/set of clauses binding is only valid from within an other, specific unit, called the "interface unit." Moreover, the context search procedure described in [1] has been modified to ignore local units, unless they are topmost in the context: this effectively makes predicates defined in local units invisible to all but explicit accesses, which can only be originated in the interface unit.

An example of a program which uses local units:

```
:- unit lists.
rev(L1, L2) :- utils :> rev_aux(L1, [], L2).
...

:- unit lists/utils.
rev_aux([], L, L).
...
```

The second unit declaration (`lists/utils`) is actually defining a unit called `utils/0` which is only visible directly from within unit `lists/0`. Therefore there may be more than one local unit with this same name, there being no conflict as it will be local to the interface unit. Interface units still share a single, flat name space.

4 Concluding Remarks

One issue which has to be satisfactorily resolved in order for a system based on GNU Prolog/CX to scale well is the structuring of the unit namespace. We are currently working on possible approaches to dealing with this problem, some of which look very promising and we expect to be able to report on that shortly.

We are currently evolving the techniques presented in this article to include some of them in the ISCO [2] compiler. It is in our plans to do extensive benchmarking of various code generation strategies, some of which will include static contexts.

As the scope of ISCO applications keeps growing, these techniques should prove more important as they should enable the development of tools which can establish properties of programs developed with GNU Prolog/CX [1]. This will mean the development of global analysis tools along the lines of those available in Ciao Prolog [3], e.g. by making use of abstract interpretation. These tools will serve optimization, debugging and documentation purposes.

Now that we have a working prototype implementation, we plan to compare it with other approaches to the modularity problem. This will be the subject of a future article.

References

1. Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.
2. Salvador Abreu and Vitor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Osamu Takata, Masanobu Umeda, Isao Nagasawa, Naoyuki Tamura, Armin Wolf, and Gunnar Schrader, editors, *Declarative Programming for Knowledge Management, 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005, Fukuoka, Japan, October 22-24, 2005. Revised Selected Papers.*, volume 4369 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2006.
3. Daniel Cabeza and Manuel Hermenegildo. The Ciao Module System: A New Module System for Prolog. In Ins Dutra, Vtor Santos Costa, Gopal Gupta, Enrico Pontelli, Manuel Carro, and Peter Kacsuk, editors, *Electronic Notes in Theoretical Computer Science*, volume 30. Elsevier Science Publishers, 2000.
4. ISO/IEC JTC1/SC22/WG17. Information technology – Programming languages – Prolog – Part 2: Modules. Technical Report DIS 13211, ISO, 2000.
5. L. Monteiro and A Porto. Contextual logic programming. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 284–299, Lisbon, 1989. The MIT Press.