Objective: in Minimum Context

Salvador Abreu[1] and Daniel Diaz[2]

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

Abstract. The current proposals for the inclusion of modules in the ISO Prolog standard are not very consensual. Since a program-structuring feature is required for a production programming language, several alternatives have been explored over the years.
In this article we recall and expand on the concepts of *Contextual Logic Programming*, a powerful and simple mechanism which addresses the general issue of modularity in Logic Programs. We claim that *unit arguments* are an essential addition to this programming model, illustrate the claim with examples and draw parallels with Object-Oriented programming.
We argue that Contextual Logic Programming is an interesting and effective tool for the development of large-scale programs built upon the Contextual Logic Programming paradigm and argue that contexts with arguments actually provide a powerful, expressive and very convenient means of structuring large applications upon a Prolog basis. We substantiate our claims with examples taken mostly from a "real world" application, Universidade de Évora's Academic Information System, which is currently being developed using the prototype implementation described in this article.
We sketch the most relevant aspects of a new implementation of Contextual Logic Programming, GNU Prolog/CX, focusing on the impact on performance of the features which were added to a regular Prolog system, highlighting the low overhead which is incurred in case these extensions are not used.

Categories: **D.2.2**–Modules and interfaces, **D.1.6**–Logic Programming, **D.1.5**–Object-oriented Programming.
General Terms: Contextual Logic Programming.

1 Introduction and Motivation

The issue of modularity in Logic Programming has long been recognized as an important and significant challenge, having spurred a large number of different proposals, for example Cabeza and Hermenegildo's proposal for Ciao Prolog[**?**] or, albeit with a cast towards the interaction with meta-programming issues, Hill and Lloyd's Gödel [8]. An extensive and still largely applicable overview of the issue was made by Bugliesi et al. in [5,4]. One problem mentioned in this decade-old work which hasn't yet been satisfactorily resolved is the adoption of a consensual model for a program-structuring mechanism applicable to Prolog, which simultaneously satisfies the requirements of large-scale application development while retaining Prolog's succinctness and expressivity: the proposals which impose a module system based on the hitherto-developed approaches stemming from particular implementations invariably bump into details which fail to be consensual such as: whether it should be atom or predicate-based, how the declarations fit into the general scheme of a Prolog program, what knowledge is required on the part of the programmer to effectively use the module system, just to name a few.

This issue is very present in the vendor and developer community, which is still in search of an adequate response to the modularity problem: although most implementations of Prolog provide a module system, these are sometimes subtly dissimilar, a situation which hinders the use of Prolog as a portable application implementation language, for lack of a standard covering the issue.

Some proposals which relate OO and Logic Programming depart significantly from the simple addition of predicate scoping control mechanisms, such is the case for instance in McCabe's Logic

and Objects [10] or Moura's LogTalk language [14,13]. One aspect that can be pointed out about these languages, particularly the latter two, is that they require a significant amount of annotations, w.r.t. classical Prolog programs, in order to take advantage of the features they provide.

An approach which takes on the problem of modularity in a simple, close to the spirit of Prolog and yet very powerful way is Contextual Logic Programming(CxLP) [12], for which an early WAM-based implementation design was presented in [9]. Later work by Omicini et al. [15] proposed a more sophisticated approach, called CSM, based on the SICStus Prolog (version 2.1) module system, which tried hard to address the run-time efficiency aspects in a satisfactory way, although with an impact on the language itself. Not being bundled with a "vendor implementation", CxLP never attained a very significant user base.

In this article we present a new implementation of Contextual Logic Programming, for which a prototype was built: GNU Prolog/CX, based on GNU Prolog [7]. A central contribution of this work is its support for the extensive use of unit arguments.

Take a simple example: consider the issue of keeping a dictionary, represented as an incomplete Prolog term, common to a lot of predicates. A regular Prolog program would require all calls to the dictionary predicates to include the dictionary term as one of the arguments. In Contextual Logic Programming a similar effect can be achieved by designating that data structure to be an argument to a unit present in the context, for example:

```
:- unit(dict(ST)).

dict(ST).

lookup(KEY, VALUE) :- ST=[KEY=VALUE|_].
lookup(KEY, VALUE) :- ST=[_|STx], dict(STx) :> lookup(KEY, VALUE).
```

Unit `dict(ST)` can subsequently be included in a context, allowing for dictionary operations to simply *omit* the dictionary argument, thereby overcoming one of Prolog's most irritating features, which can arguably be considered an obstacle to the language becoming more popular: the proliferation of predicate arguments in anything other than "toy problems," as well as the lack of a language feature tying various clauses or predicates together.

GNU Prolog/CX is presently being used at Universidade de Évora to construct its second-generation integrated information system, a project which was spurred by the University's decision to simultaneously reformulate and adapt to "the Bologna principles" its 42 undergraduate study programmes, all at once, for the academic year 2003/04. This system's regular user base consists of about 8000 students, 600 faculty and 300 staff members. It is undergoing intensive development by a team of three full-time programmers and, at the time of this writing, is actually *ahead of schedule* and is already proving invaluable in assisting with the timeliness and coherence of the academic restructuring process.

The remainder of this article is structured as follows: section 2 presents our revised specification of Contextual Logic Programming. In section 3 the issue of likening Contexts and Objects is explored in some detail, while section 4 discusses some implementation issues, the options we made and their performance impact. Finally, section 5 draws some conclusions and attempts to point at unsolved issues, for further research.

2 Unit Arguments in Contextual Logic Programming

The fundamental contribution w.r.t. the previous proposals dealing with Contextual Logic Programming by Miller [11], Natali, Omicini, Porto and others [5,6,12] is the extensive use of *unit arguments* or *parameters* as a central feature of contexts. This construct implies looking at the unit specification as a compound term, i.e. one with a main functor and subterms. The subterms are restricted to be named variables, in the unit declaration directive.

Although present in some of the early formalizations of the concept of "units as sets of clauses", parameters haven't been systematically explored in the setting of Contextual Logic Programming;

we purport to do so in this article. A unit argument can be thought of as a sort of "unit global" variable, i.e. one which is shared by all clauses defined in the unit, thereby aiming to solve one of Prolog's traditionally most annoying features: the inevitable proliferation of predicate arguments, whenever a global structure is to be passed around. This sort of feature is already present in other approaches to the integration of the LP and OO paradigms, such as McCabe's Logic and Objects[10].

Consider a clause $C = H \leftarrow B$, where H is the clause head and B the body, being defined as:

$$\lambda\overrightarrow{v}.\exists\overrightarrow{w}.H \leftarrow B \tag{1}$$

Where $\overrightarrow{v}$ is the set of variables which occur in the clause head H. $\overrightarrow{w}$ is the set all remaining variables which occur in the clause. We will subsequently omit the explicit existential quantification over $\overrightarrow{w}$ when presenting clauses: this is just notational convenience as it should remain, although implicitly.

Clause C is part of predicate p where p is the main functor of H. Predicate p is made up of the (ordered) set of clauses which have p as their head's main functor.

2.1 Unit and Unit Arguments

Arguments (parameters) for a unit u can be thought of as a unit-global substitution σ, which implicitly applies to all clauses in u.

Similarly, a unit u with parameter variables $\overrightarrow{p}$ can be described as a set of clauses, for which all variables in $\overrightarrow{p}$ are implicitly existentially quantified in all clauses in the unit. Taking this into consideration, equation (1) may now be presented as:

$$u \equiv \lambda\overrightarrow{p}.\{\lambda\overrightarrow{v_i}.H_i \leftarrow B_i\} \tag{2}$$

In other words, all clauses are augmented with variables shared with the unit identifier. During execution, these will be implicitly bound to the subterms of the head of the current context corresponding to these variables.

2.2 Context traversal

When reducing a goal whose main functor is `pred/arity`, a Contextual Logic Programming engine will traverse the *current context* looking for the first unit which includes a definition for this predicate. This process is detailed below, in section 2.4, equation (5).

The predicate under consideration will then have its body execute in a new context, that is the suffix of the original current context which starts with the unit which contains the first definition for the predicate being invoked. This amounts to the "eager" resolution method initially mentioned in the work of Lamma et al. [9]: the new context becomes the *current context* for the selected clause and the initial current context is now designated by the *calling context*.

2.3 Context Operators

Consider that: U is a *unit descriptor*, i.e. a term whose main functor designates a unit name; C is a *context*, i.e. a list of unit descriptors; G is a Prolog goal.

There are two sets of context operators. The first three, which we'll refer to as the *basic set* consists of those which are fundamental to the specification and low-level implementation of GNU Prolog/CX. They are:

- $\underline{C \texttt{ :< } G}$, the *context switch* operation, which attempts to reduce goal G in context C, i.e. totally bypassing the current context.
- $\underline{\texttt{:> } C}$, the *current context enquiry* operation, which unifies C with the current context.
- $\underline{\texttt{:< } C}$, the *calling context enquiry* operation, which unifies C with the calling context.

A further set of operations involving contexts are useful in a number of situations, some of which will be addressed later in this article. These operations may be defined using the fundamental set, as follows:

- U `:>` G, the *context extension* operation, which extends the current context with unit U before attempting to reduce goal G, for which see equation (9). This operator is defined as if by the Prolog clause:

```
U :> G :- :> C, [U|C] :< G.
```

- `:^` G, the *supercontext* operation, where the suffix of the current context obtained by dropping the topmost unit is used to evaluate G. This behaves as if defined by the Prolog clause:

```
:^ G :- :> [_|C], C :< G.
```

- `:#` G, the *lazy call* operation, essentially the same as introduced in Lamma et al. [9] and so called because the current context doesn't change w.r.t. the calling goal, which behaves as if defined by the Prolog clause:

```
:# G :- :< C, C :< G.
```

- U `::` G, the *guided context traversal* operation, which behaves as if defined by the Prolog clauses:

```
U :: G :- :> C, GC=[U|_], suffixchk(GC, C), GC :< G.
```

  Where `suffixchk/2` is a deterministic predicate where `suffixchk(SUFFIX, LIST)` succeeds if `SUFFIX` is a suffix of `LIST`.

One operation that is left out by design is that provided by the *eager call* operator, as per Lamma et al. [9], as it is implicitly used in all goals.

2.4 Operational Semantics

A GNU Prolog/CX computation is denoted by the evaluation of a *goal* in a *context*. Before stating how this process is carried out, a few definitions for a Contextual Logic Program are in order. Let:

- u denote a *unit*, i.e. a term which designates a set of predicates. A unit is said to be:
  - *simple* in which case the term is an atom.
  - *qualified* when the term is compound, i.e. has subterms.
- $\overline{u}$ denote the set of predicate symbols that are defined in unit u.
- $\gamma \equiv u_1.u_2 \cdots u_n$ is called a *context*, and is constructed as a list of units. We shall use the letters γ and δ to denote contexts. In the conditions for some rule, we may specify a context as $\gamma = u.\gamma'$.
- γ/δ is the pair formed by the *current context* and the *calling context*.
- A goal $\gamma/\delta \vdash G$ is read G *in context* γ *with calling context* δ.
- The *empty context*, denoted `[]` represents the set of predicates accessible from a regular Prolog program. Such predicates will be assumed to be defined in the special unit `[]` and will have precedence over predicates of the same name defined in other units.[1] These may resort to ISO-Prolog style module qualifiers.

Equipped with these definitions, we may now enumerate the Contextual Logic Programming rules which specify computations and apply to GNU Prolog/CX.

[1] This is a small concession to efficiency, in that regular Prolog programs should incur little or no overhead because of the CxLP engine.

Prolog Goal Expansion

When a particular goal has matching clauses in the special `[]` unit, the *Prolog goal expansion* rule applies:

$$\frac{\gamma/\delta \vdash (G_1, G_2 \cdots G_n)\theta}{\gamma/\delta \vdash G} \begin{cases} G \in \overline{[]} \\ H \leftarrow G_1, G_2 \cdots G_n \in [] \\ \theta = \text{mgu}(G, H) \end{cases} \tag{3}$$

This rule is similar to goal expansion rule (4) but for one aspect: a candidate clause for the goal was found in the *empty context*, i.e. in the set of regular Prolog clauses. The effect of applying this rule is that the goal gets replaced by the body of the clause, but without the current context being affected at all.

Whenever applicable, this rule *takes precedence over rule (4)*.

The inclusion of a rule such as this one allows for GNU Prolog/CX to easily build upon a regular Prolog implementation, preserving the semantics of all library predicates and, in particular, of meta-predicates such as `setof/3`.

Goal Expansion

When a goal has a definition in the topmost unit in the context, it will be replaced by the body of the matching clause, after unification:

$$\frac{\gamma/\delta \vdash (G_1, G_2 \cdots G_n)\theta}{\gamma/\delta \vdash G} \begin{cases} \gamma = [u, ...] \\ G \in \overline{u} \\ H \leftarrow G_1, G_2 \cdots G_n \in u \\ \theta = \text{mgu}(G, H) \end{cases} \tag{4}$$

This rule implies that a goal G invoked in a context γ will be replaced by the body of a clause for that predicate, only if the unit u on top of the current context defines the predicate of G. The context for each of the subgoals in the body of the clause is identical to that of the calling goal, after head unification has been performed.

Note that whenever this rule applies, the calling context δ is preserved and passed on unchanged to the clause body.

Context Traversal

When neither the Prolog Goal Expansion (3) nor the Goal Expansion rule (4) apply, the context will be traversed by re-evaluating the goal in the supercontext, i.e. by dropping the top unit.

$$\frac{\gamma'/\gamma \vdash G}{\gamma/\delta \vdash G} \begin{cases} \gamma = [u_1, u_2, ...u_n, ...] \\ \gamma' = [u_n, u_{n+1}, ...] \\ \forall i \in \{1 \cdots n-1\}, G \notin \overline{u_i} \\ G \in \overline{u_n} \end{cases} \tag{5}$$

This process locates the first unit u_n which defines the goal's predicate. It then specifies that goal G is to be evaluated using, as *current context*, γ' which is the longest suffix of γ which starts with unit u_n and, as *calling context*, the original current context γ. The previous calling context, δ is ignored.

Context Enquiry

In order to make the context switch operation (8) useful, there needs to be an operation which fetches the current context:

$$\frac{\theta}{\gamma/\delta \vdash \texttt{:>}\, X} \begin{cases} \theta = \text{mgu}(X, \gamma) \end{cases} \tag{6}$$

This rule recovers the current context γ as a Prolog term and unifies it with term X, so that it may be used elsewhere in the program.

Calling Context Enquiry

This rule recovers the calling context as a Prolog term, to be used elsewhere in the program.

$$\frac{\theta}{\gamma/\delta \vdash :< X} \begin{cases} \theta = \mathrm{mgu}(X, \delta) \\ \end{cases} \tag{7}$$

This is similar to the context enquiry rule (6), except that X is unified with the current clause's calling context δ instead of the current context.

Context Switch

The purpose of this rule is to allow execution of a goal in an arbitrary context, independently of the current context.

$$\frac{\gamma'/\gamma' \vdash G}{\gamma/\delta \vdash \gamma' :< G} \tag{8}$$

This rule causes goal G to be executed in context γ' (both current and calling.)

Moreover, we shall present rules for the *Context Extension*, *Super Call*, the *Lazy Call* and *Guided Context Traversal* operations, even though these may be defined using the previously introduced set. We feel that these are sufficiently important in actual use that they warrant a separate discussion.

Context Extension

This rule simply prepends a single qualified unit to the current context before attempting to execute a goal.

$$\frac{\gamma'/\gamma' \vdash G}{\gamma/\delta \vdash u :> G} \begin{cases} \gamma' = [u|\gamma] \\ \end{cases} \tag{9}$$

Goal G will be evaluated in the new context γ', in the roles both of calling context and current context.

Super Call

The context traversal rule (5) stops as soon as the goal expansion rule (4) applies, thereby preventing units lower in the context to be candidates for the goal expansion. Sometimes it may be interesting to programatically allow a goal to be evaluated further down in the context, even though there is already a matching unit for the concerned predicate.

$$\frac{\gamma'/\gamma' \vdash G}{\gamma/\delta \vdash :\hat{} G} \begin{cases} \gamma = [u_1, u_2 \cdots u_n] \\ \gamma' = [u_2 \cdots u_n] \end{cases} \tag{10}$$

This rule evaluates G in the supercontext γ' (i.e. the context obtained from the current context γ by dropping the topmost unit). The calling context for G is set identically.

Lazy Call

The *lazy call* operator is intended to behave like a regular goal expansion / context traversal (rules (4) and (5)), except that it starts with the *calling context* instead of the current context.

$$\frac{\delta/\delta \vdash G}{\gamma/\delta \vdash :\# G} \tag{11}$$

This rule evaluates G using the current clause's calling context δ both as the current and the calling contexts.

Guided Context Traversal

Sometimes it is interesting to search the context for a *specific* unit, and trigger goal expansion in the suffix of the context that starts at that point. Such is the purpose of the rule:

$$\frac{\gamma'/\gamma' \vdash G\theta}{\gamma/\delta \vdash u :: G} \begin{cases} \gamma = [u_1, ...u_n, ...] \\ \gamma' = [u_n, u_{n+1}, ...] \\ \forall i \in \{1 \cdots n-1\}, \nexists\, \sigma = \text{mgu}(u, u_i) \\ \theta = \text{mgu}(u, u_n) \end{cases} \tag{12}$$

The guided context traversal process locates the first unit u_n in the current context γ which unifies with u. It then specifies that goal G is to be evaluated using, both as *current context* and as *calling context*, γ' which is the longest suffix of γ which starts with unit u_n, after u has unified with u_n. The previous current and calling contexts are ignored. If u doesn't unify with any unit in the context, this rule *fails*.

A slightly abusive (from a formal standpoint) but convenient reading of this rule will treat parameter u specially, so that it either:

- Unifies with u_n, in which case this rule behaves exactly as stated.
- Is the *main functor* of u_n: in this situation, u is not required to unify with u_n, only to match its functor.

This rule will be especially useful for succinctly specifying particular subcontexts, as will be seen in section 3.2. Note that this rule *does not* search for which unit defines a predicate for G: this task is still left to the regular context traversal rule (5), which ought to be applicable to the resulting state.

3 Using Contexts as Objects

The integration of the Object-Oriented and Logic Programming paradigms has long been an active research area since the late 1980's; take for example McCabe's work [10]. The similarities between Contextual Logic Programming and Object-Oriented Programming have been focused several times in the literature; see for instance the work by Monteiro and Porto [12] or Bugliesi [3].

Other than the implementation-centered reports, previous work on Contextual Logic Programming focuses largely on issues such as the policy for context traversal, what the context becomes once a unit satisfying the calling goal is found, what to do when multiple units provide clauses for the same predicate, how to automatically tie several units together or how to provide encapsulation and concealment mechanisms.

To the best of our knowledge, no published work builds on the notion of context arguments and their widespread use, although Miller's initial work [11] mentions the possibility of using module variables. This feature was present as a "hack" in the first C-Prolog based implementation of Contextual Logic Programming but was a little let down, possibly for lack of an adequate formalization and the nonexistence of convincing examples. We propose to alter this situation, by addressing these issues directly.

Instead of viewing a context as an opaque execution attribute, as happens in CSM [15] for instance, we choose to regard it as a first-class entity, i.e. as a Prolog term. Not only is the context accessible from the program, but *it is intended* that it be explicitly manipulated in the course of a program's regular computation. The performance impact of this option will be succinctly analyzed in section 4: at this point we shall concentrate on the possibilities it allows from an expressiveness point of view.

3.1 Terminology

Table 1 establishes some parallels between Contextual Logic Programming (CxLP) and regular OO terminology. The most notable difference between the CxLP and OO paradigms has to do with the concept of *inheritance*: instead of being statically defined, it is completely dynamic for each context (i.e. an "object") defines its own structure and, implicitly, its behaviour wrt. messages.

A "context skeleton" is a list whose elements are unit designators, i.e. terms of the form `FUNCTOR/ARITY` where `FUNCTOR` is an atom and `ARITY` is an integer. It can be thought of as designating the structure of a specific context.

OO	CxLP
Class	Context skeleton
Object Instance	Context
Instance Variable	Unit argument
Class member	Unit
Method	Predicate
Message	Goal

Table 1. CxLP vs. OO paradigms

3.2 Access Methods

One fundamental feature of OO languages is that object elements are only manipulated via accessor methods, take for example the approach taken in Smalltalk, where instance variables are not visible outside the body of instance methods. Similarly and as in our approach the role of instance variable is carried out by unit parameters – which are variables of the underlying Logic Programming system, i.e. they can be either regular or constrained Logic Variables – a programming dialect has been devised which fulfills the same purpose. Consider for example the units:

```
:- unit(person(ID, NAME, BIRTH_DATE)).   :- unit(room(ID, NAME, CAPACITY)).

person(ID, NAME, BIRTH_DATE).            room(ID, NAME, CAPACITY).

id(ID).                                  id(ID).
name(NAME).                              name(NAME).
birth_date(BIRTH_DATE).                  capacity(CAPACITY).
```

Unit `person/3` implicitly defines "access" predicates[2] `id/1`, `name/1` and `birth_date/1`. In practice this allows for such a goal to be evaluated in a context which includes `person(...)`, with the effect that its argument will be unified with the corresponding parameter in the context. The whole of the unit and its actual parameters may be accessed via the `person/3` predicate, which unifies its arguments with the unit parameters, in an actual context. Unit `room/3` is structured similarly.

Should the context include another unit which specifies arguments (and therefore access predicates) with the *same name*, as in the context `C = [person(...), room(...), ...]`, it would seem complicated to get to unit `room/3`'s definition of `id/1`, as one would have to contract the context in order to bypass the definition provided by unit `person/3`. Such is the purpose for which the *Guided Context Traversal* rule (12) was introduced: we can simply evaluate the goal `C :< room :: id(RID)` in order to unify `RID` with the first argument of the first occurrence of `room/3` in context `C`, i.e. the `room`'s `id`.

3.3 Semi-static Inheritance

Sometimes it is important to ensure that one specific unit (say, `u1`) *specializes* the behaviour specified by another (we'll call it `u2`). This goal can be achieved "manually" by ensuring that contexts being built adhere to the desired structure, i.e. `[u1, u2, ...]`. Besides being awkward to enforce, this assumption will easily fail if we extend some context just with `u1` before launching a goal.

Previous work by, namely that by Natali and Omicini [15] addressed the issue by including new directives in the unit declaration, which would enforce an explicit inheritance relation, by means of an `inherits` or `is_a` operator or the `requires` directive.

[2] These "access predicates" can be automatically generated by a preprocessor: take for instance the ISCO compiler, which can produce units for each defined class using this approach.

It is our feeling that these approaches are at odds with the nature of the Prolog language, in that they foster abundant annotations which are, essentially, alien to this programming language. To address the issue in a minimalistic way, we propose the following coding conventions, which do not rely on any mechanism other than those which have already been presented.

Consider that, for proper operation, unit `registration` requires units `student` and `course` to sit immediately below it in the context, `registration` can be coded as follows:

```
:- unit(registration(A, B)).

context([U, student(B), course(A) | CX]) :- :> [U | CX].
```

Subsequently, code wanting to use unit `registration` should ensure that the context is properly constructed: this is the purpose of the predicate `context/1` in unit `registration`. Notice that it uses the `:>` operator to extract the current context but also the *unit arguments*, in order to unify its argument with a new term which is the desired context.

Predicate `context/1` should then be used as shown in this goal:

```
registration(a, X) :> context(CX), CX :< GOAL.
```

The point is that, with this approach, a unit may specify what other units should be in the context for it to operate properly. It does so with minimal term reconstruction as the final context which the goal is applied to can share much with that provided to the `context/1` goal. For this to become a standard dialect of GNU Prolog/CX it is sufficient to define the following operator:

```
U :>> GOAL :- U :> context(CX), CX :< GOAL.
```

Thus, `U :>> G` effectively becomes a variation on the standard context-extension goal `U :> G`, one which enforces the semi-static inheritance specified by the unit we're extending the context with.

Nothing prevents `context/1` from being nondeterministic, thereby allowing for flexible multiple inheritance behaviors to be implemented.

3.4 Creating and Using Contexts

Consider an example loosely inspired by LDAP: objects ("records") are structured as belonging to classes in a casuistic basis, a single "key value" may be searched for, returning several instances which may be classified as belonging to distinct classes. Consider that we have an *account*, which inherits from `person` but also from other "classes": the *context* will define the structure of the complete instance and can be constructed like this:

```
person(1, 'Dan', 1970-01-01) :>
       login(diaz, foo123) :>
       email('diaz@paris.fr') :> :> C.
```

Which can be read as "unify `C` with the current context, after performing these extensions". After the completion of this goal, variable `C` will be bound to a context that is able to provide successful answer substitutions to whatever predicates are defined in the `person/3`, `login/2` and `email/1` units: this includes all access predicates, as previously introduced, as well as any other predicates defined in these units.

The term to which `C` is then bound can later be supplied as the context argument to a context switch operation in which the goal may be one of the access predicates. Thus, we can have the goal:

```
C :< (login(ID, _), person::name(WHO)).
```

which will succeed, binding variable `ID` to `diaz` and variable `WHO` to `'Dan'`.

If this coding style is used in a language which provides a mechanism for persistent storage such as ISCO [2] – which presently relies on Contextual Logic Programming both for its specification

and its implementation – the binding for `C` may be stored, retrieved and later re-used. We are presently using this approach as a means of implementing *sessions* in SIIUE's user interface, a very early version of which is described in [1].

3.5 "Virtual Methods"

A useful mechanism provided by OO languages is that of virtual methods: this allows for common behaviour to be coded once and for all in a "superclass," while allowing for subclass-specific methods to influence the outcome of the common (i.e. defined in the superclass) code.

GNU Prolog/CX retains the *lazy call* mechanism to directly address this issue, as was the case in Natali et. al's work, see for example [15]. An alternative form is achieved through a careful organization of contexts in which the pertinent calling context is included as an argument to a particular unit. Note that this approach creates circular structures, but this should not constitute a problem.

3.6 Contexts as Implicit Computations

In a setting related to the ISTO[3] language, consider a unit `person(ID, NAME, BIRTH_DATE)` which defines the following predicates:

- `item/0` which returns, through backtracking, all instances of the `person/3` database relation by instantiating unit arguments,
- `delete/0` which nondeterministically removes instances of the `person/3` database relation, as restricted by the unit arguments,
- `insert/0` which inserts new instances into the `person/3` database relation, taking the values from the unit argument.

Accessing an "object" specified by a context is always done via one of these predicates, which are to be evaluated in a context which specifies the relation (in this case `person/3`). Assume that there are also predicates with the same name and one argument, which represents the relevant unit with bound arguments, i.e. `item/1`, `delete/1` and `insert/1`. An actual implementation of these predicates could rely on the standard Prolog built-ins `clause/1`, `retract/1` and `assertz/1` or access an external database, as is done in the ISCO compiler [2].

Consider also that we have the following general-purpose units:

```
:- unit(delete).                :- unit(insert).
item :- delete.                 item :- insert.
item(X) :- delete(X).           item(X) :- insert(X).
```

These units can be prepended to the context mentioned above to signify that accessing the `item/0` (or `item/1`) relation will map to one of the other functions: the operation which is to be applied (e.g. query, insert, delete) will then be an integral part of the context, for example:

```
person(ID, NAME, BIRTH) :> (
        read(ID),
        item -> (
            read(NNAME), read(NBIRTH),
            delete :> item,
            person(ID, NNAME, NBIRTH) :> insert :> item ))
```

[3] ISTO is a development of ISCO [2], a Prolog-based mediator language which can transparently access several kinds of information sources, namely relational databases. ISTO relies on GNU Prolog/CX as its compiler's target language.

This code reads a person's `ID` and, if it already exists, proceeds by reading and replacing the name and birth date in the corresponding tuple.

The purpose of using this stylized approach is to present the context as a complete specification for an intended ("latent") computation, leaving it up to the programmer to initiate the process by invoking the `item/0` goal.

3.7 Behaviour Extension

Consider the situation where a predicate `p/a` defined by a context `C` is to be partly redefined in a unit which requires additional goals to be satisfied. For example, suppose that modifications to relation `person/3` are to be logged as tuples for a new dynamic relation `log(TIME, ACTION)` which is to be built when new tuples for the `person/3` relation are inserted or existing ones removed. Since all actions (query, insert, delete) are performed in an identical way, it is sufficient to prefix the context with a unit which redefines the "action" goal (`item/1`) for the `insert` and `delete` units in the intended way. A possible implementation for `log` is:

```
:- unit(log).
item(X) :- time(T),
           LC = [ insert, log(T, L) ],
           ( delete :: item(X) -> L=delete(X), LC :< item
           ; insert :: item(X) -> L=insert(X), LC :< item
           ; :^ item(X) ).
```

The guided traversal goals will only succeed in case there's a `delete` or `insert` unit in the context. Note the explicit construction of the `LC` context. For example, in order to remove a tuple `X`, for which the `ID` is less than 10, from the `person/3` relation, we can write:

```
ID #< 10, person(ID, _, _) :> delete :> log :> item(_).
```

The inclusion of the `log` unit in the context has the effect that the removal of each tuple `X` is accompanied by the insertion of the term `log(T, X)` into the `log/2` relation.

As another example, should we purport to act upon *all* tuples (e.g. as in set programming), it is sufficient to define a new unit `all/0` as follows:

```
:- unit(all).
item(X) :- findall(I, :^ item(I), X).
```

Which could then be used to, say, clean up someone's financial past:

```
ID #< 10, debt(ID, _, _) :> delete :> log :> all :> item(_).
```

4 Overview of the Prototype Implementation

In order to experiment programming with contexts we have developed a first prototype inside GNU Prolog [7]. Our main goal was to have a light implementation modifying the current system as little as possible. Due to space restrictions we only give here an overview of this implementation and focus on the most important points only.

4.1 Managing Contexts

A context is simply stored as a Prolog list. Each element being either an atom (unit without arguments) or a compound term whose principal functor is the unit name and arguments are unit arguments. The drawback of this simple approach could be a little overhead when a predicate is called since the current context need to traversed (we can imagine a better data structure than

a Prolog list, as done by Omicini et al. [15]). On the other hand, there are several advantages to this choice: the interface with the user code (needed by the context extension and the inquiry operations) is very simple. Also the retrieval of unit arguments is very easy. Indeed, this comes down to geting the first element of the the current context and, from this term, the associated argument. Both operations can be achieved using the built-in predicate `arg/3`. In fact we have enhanced the WAM with a new instruction `cxt_arg_load` which acts exactly as `arg/3` but faster since it does not include any error checking.

The WAM must also be extended with 2 new registers to maintain the calling context (`CK`) and the current context (`K`). Initially, `K` and `CK` contain the empty context (i.e. the atom `[]`). We decided to map these registers to 2 WAM temporaries: `x(254)` for `CK` and `x(255)` for `K`. This simplifies the compiler, reduces the number of new WAM instructions (no new instruction is needed to read/write these registers from/to WAM variables) and makes it possible to benefit from the optimizations performed by the register allocator.

Finally, note that both `K` and `CK` should be saved in choice-points and in environments. However, to avoid to penalize code which does not use contexts we do not reserve a cell in environments automatically. Instead, if a context call is needed, the value of `K` is saved in a permanent variable (`y(...)` variables in the WAM). For choice-points we also avoid to automatic allocation of an additional cell. Instead, when `K` or `CK` must be modified, its current *value* is trailed (to be restored when backtracking occurs). To avoid useless trailings (only one trailing is needed per choice-point) a time-stamp is attached to `K` and `CK` and is compared to the time-stamp of the current choice-point to detect whether the involved register has already been trailed. GNU Prolog already provides all necessary mechanisms (time-stamp on choice-points and a value-trail) since they are needed for its finite domain constraint solver.

4.2 The Context Call

The main change concerns a call to a given predicate. While such a call can be resolved statically in a classical Prolog execution, a context call needs to explore the context to resolve the call. Obviously, it is possible to consider all calls as context calls. However we introduced a little optimization: calls to built-in predicates and calls to predicates defined in the same unit are translated into classical WAM calls (which give rise to native jumps). We thus have introduced 2 new instructions: `cxt_call(P/N,V)` and `cxt_execute(P/N,V)` where `P/N` is the predicate to call and `V` is a variable specifying the context to use. Both instructions first look for a definition for `P/N` in the global predicate table (containing all built-in predicates and predicates not defined inside a unit). If no global definition is found, the current context is scanned until a definition is found. This process is summarized in figure 1.

```
trail value of K and CK if needed (testing their time-stamp)
K ← V
CK ← K
if there is a definition for P/N in the global table then
    branch to this definition
while K is of the form [E|K'] do      // i.e. K is not empty
    let T be the predicate table associated to <functor/arity> of E
    if there is a definition for P/N in T then
        branch to this definition
    K ← K'
end
error: P/N is an undefined predicate
```

Fig. 1. Context search procedure

4.3 Example of Code Produced

Consider the following unit:

```
:- unit(u1(A,B)).

q :- r, a :> s, t.
p(X) :- X :> q(A,B).
v :- :< C, [a|C] :< w.
```

Here is the compilation of the first clause. Note how the permanent variable `y(0)` is used to save `K`, and used to the context call of `r/0` and of `t/0`. It is also used to extend the context with the unit `a` for the call to `s/0`.

```
predicate(q/0,3,static,private,user,[
    allocate(1),                % environment with 1 cell to save K
    get_variable(y(0),255),     % y(1) = K (current context)
    cxt_call(r/0,y(0)),         % call r/0 with context K
    put_list(0),                % x(0) = [
    unify_atom(a),              %         a|
    unify_local_value(y(0)),    %           K]
    cxt_call(s/0,x(0)),         % call s/0 with context [a|K]
    put_value(y(0),0),          % x(0) = K
    deallocate,                 % remove environment
    cxt_execute(t/0,x(0))]).    % call t/0 with context K
```

The second clause shows how unit arguments are handled. Recall that `cxt_arg_load` is equivalent to `arg/3`. Note that this instruction is used to get the first element of the list (i.e. the current unit) and, from this term, all necessary unit arguments. Here is the WAM code produced:

```
predicate(p/1,5,static,private,user,[
    get_variable(x(3),0),       % x(3) = X
    cxt_arg_load(1,x(255),x(1)),% x(1) = K (current context)
    cxt_arg_load(1,x(1),x(0)),  % x(0) = A (first unit argument)
    cxt_arg_load(2,x(1),x(1)),  % x(1) = B (second unit argument)
    put_list(2),                % x(2) = [
    unify_local_value(x(3)),    %         X|
    unify_local_value(x(255)),  %           K]
    cxt_execute(q/2,x(2))]).    % call q/2 with context [X|K]
```

Finally, the last clause recovers the calling context and uses it to create a new context to call `w/0`. This gives rise to the following code:

```
predicate(v/0,7,static,private,user,[
    put_list(0),                % x(0) = [
    unify_atom(a),              %         a|
    unify_value(x(254)),        %           CK]
    cxt_execute(w/0,x(0))]).    % call w/0 with context [a|CK]
```

4.4 Evaluation of the Overhead

It this section we try to evaluate the overhead of our context implementation (i.e. when no contextual programming facility is used).

For this purpose we have compared the prototype with GNU Prolog on a set of benchmarks. However, since all benchmarks are on a single source file no context calls would be generated, due to the optimizations explained above and thus no penalty could be constated. We have then turned all optimizations off to force the compiler to generate context calls for *all* calls. This means that even recursive predicates (common in benchmarks) and built-in predicates give rise to contextual (i.e. indirect) calls. Table 2 exhibits those results on a set of classical benchmarks. We have also included the GNU Prolog `pl2wam` sub-compiler as it is a more representative example (applied to `pl2wam` itself). For each benchmark (including `pl2wam`), the execution time is the total time in seconds needed for 10 iterations. Times are measured on an Pentium4 1.7 Ghz with 512 MBytes of memory running RedHat Linux 7.2. The average slowdown factor is around 1.5 which is very encouraging taking into account the simplicity of the implementation and the lack of optimization (since the few ones implemented were turned off). Finally it is worth noticing that on a real-life application like the `pl2wam` compiler the slowdown factor is limited to 1.25.

Program × **10 iter.**	**gprolog 1.2.18**	**contexts no opt.**	**slowdown factor**
`boyer`	0.610	0.795	1.303
`browse`	0.744	1.024	1.376
`cal`	0.093	0.147	1.581
`chat_parser`	0.161	0.231	1.435
`ham`	0.538	0.881	1.638
`nrev`	0.088	0.178	2.023
`poly_10`	0.043	0.063	1.465
`queens`	0.532	0.836	1.571
`queensn`	2.153	3.530	1.640
`reducer`	0.039	0.049	1.256
`sendmore`	0.053	0.067	1.264
`tak`	0.089	0.127	1.427
`zebra`	0.041	0.053	1.293
`pl2wam`	4.600	5.800	1.260
average slowdown factor			1.466

Table 2. Worst overhead evaluation

4.5 Evaluation of the Contextual Logic Programming Implementation

In order to evaluate the context implementation, we follow a methodology similar to that of Denti et al. [6]: a goal is evaluated in a context which is made up of a unit which implements the goal predicate, below a variable number of "dummy" units which serve to test the overhead introduced by the context search. We use the exact same program as in [6]; the goal being used is 100,000 executions of: `list :> {`*`dummy :>`*` } mem([a,b],[e,d,[f,g],h,[b,a]])`, in which there are `N` "dummy" units in the initial context, as per [6]. The results are shown in table 3: these stem from the arithmetic mean of 10 runs on a 1GHz Pentium III running Linux.

`N`	**Time (sec)**	**perf. loss**	**CSM perf. loss**
0	0.971	0.0%	0.0%
1	0.986	1.5%	10.3%
2	1.004	3.4%	20.6%
5	1.043	7.4%	51.6%
10	1.102	13.5%	n/a
20	1.235	27.2%	n/a
50	1.595	64.3%	n/a
100	2.238	130.5%	n/a

Table 3. Varying context depth

When compared to CSM, the observed relative performance is much better in GNU Prolog/CX: even in CSM's most favorable situation (the modified WAM), there is a 50% performance hit as soon as there are 5 "dummy" units in the context. What this indicates is the effective ability to use *deep contexts* in the present implementation, as the "50% performance degradation" threshold is only reached when the context comprises about 40 dummy units. Incidentally, we observe that both implementations exhibit a performance hit which is linear on the number of dummy units, as expected.

These are a very encouraging results, as GNU Prolog is an arguably more efficient implementation than SICStus Prolog 2.1, on which CSM is based, thereby exacerbating the impact of performance-impairing extensions such as this one.

5 Conclusions and Directions for Future Work

We successfully developed a working prototype of a Contextual Logic Programming implementation on top of GNU Prolog which, while straightforward, has exhibited reasonable efficiency in

different benchmarks. One goal of this work was to experiment extensively with the mechanism of unit arguments and explore its possibilities: the applications which have been developed support the claim that this construct is useful in the development of large projects with a Logic Programming language.

Several lines of work are still open and will receive attention in the near future. These include refinements to the language such as predicate concealment, implementation usability developments such as dynamically loadable units and various kinds of optimizations, both in the compiler and in the runtime support system.

References

1. Salvador Abreu. A Logic-based Information System. In Enrico Pontelli and Vitor Santos-Costa, editors, *2nd International Workshop on Practical Aspects of Declarative Languages (PADL'2000)*, volume 1753 of *Lecture Notes in Computer Science*, pages 141–153, Boston, MA, USA, January 2000. Springer-Verlag.
2. Salvador Abreu. Isco: A practical language for heterogeneous information system construction. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. INAP.
3. M. Bugliesi. A declarative view of inheritance in logic programming. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 113–127, Washington, USA, 1992. The MIT Press.
4. Michele Bugliesi, Anna Ciampolini, Evelina Lamma, and Paola Mello. Optimizing modular logic languages. *ACM Computing Surveys (CSUR)*, 30(3es):10, 1998.
5. Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19/20:443–502, 1994.
6. Enrico Denti, Evelina Lamma, Paola Mello, Antonio Natali, and Andrea Omicini. Techniques for implementing contexts in Logic Programming. In Evelina Lamma and Paola Mello, editors, *Extensions of Logic Programming*, volume 660 of *LNAI*, pages 339–358. Springer-Verlag, 1993. 3rd International Workshop (ELP'92), 26–28 February 1992, Bologna, Italy, Proceedings.
7. Daniel Diaz and Philippe Codognet. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 2001(6), October 2001.
8. Patricia Hill and John Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge, MA, 1994. ISBN 0-262-08229-2.
9. E. Lamma, P. Mello, and A. Natali. The design of an abstract machine for efficient implementation of contexts in logic programming. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 303–317, Lisbon, 1989. The MIT Press.
10. Francis G. McCabe. *Logic and Objects*. Prentice Hall, 1992.
11. Dale Miller. A logical analysis of modules in logic programming. *The Journal of Logic Programming*, 6(1 and 2):79–108, January/March 1989.
12. Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993.
13. Paulo Moura. Logtalk web site. `http://www.logtalk.org/`.
14. Paulo Moura. Logtalk 2.6 Documentation. Technical Report DMI 2000/1, University of Beira Interior, Portugal, 2000.
15. Antonio Natali and Andrea Omicini. Objects with State in Contextual Logic Programming. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 714 of *LNCS*, pages 220–234. Springer-Verlag, 1993. 5th International Symposium (PLILP'93), 25–27 August 1993, Tallinn, Estonia, Proceedings.