Logical Loops

Joachim Schimpf

IC-Parc, Imperial College,
London SW7 2AZ
United Kingdom
J.Schimpf@ic.ac.uk

Abstract. We present a concrete proposal for enhancing Prolog and Prolog based Constraint Logic Programming languages with a new language construct, the logical loop. This is a shorthand notation for the most commonly used recursive control structure: the iteration or tail recursion. We argue that this enhancement fits well with the existing language concepts, enhances productivity and maintainability, and helps newcomers to the language by providing concepts that are familiar from many other programming languages. The language extension is implemented and has been in everyday use over several years within the ECL^{i}PS^{e}system.

1 Introduction

Almost 30 years after its inception, Prolog and Prolog-based systems are still the most widespread implementations of logic programming languages. Prolog exhibits a characteristic simplicity, and economy of concepts, which makes some fall in love with the language, but confuses many newcomers, and prompts others not to take it seriously.

More recent attempts at better logic programming languages have taken quite radical measures, like adding strict typing and moding (Gödel [6], Mercury [9]), integrating logic and functional styles (Curry [5], Mercury) or dismissing most of the Prolog heritage, keeping essentially logical variables in a host language that concentrates on other main aspects (Oz [10], Ilog solver [8]).

The enhancements presented in this paper are not about such fundamental modifications. They keep the underlying concepts unchanged, while at the same time making Prolog programming

- more effective and maintainable, by allowing shorter programs and reducing the likelihood of errors
- more elegant and readable, by making the programmer's intention more explicit
- more accessible to newcomers, by providing familiar concepts

The features we introduce can be explained, understood, and implemented by program transformation or preprocessing. They could, therefore, be dismissed as mere 'syntactic sugar'. However, we believe that the substantial benefit gained from such comparatively simple measures warrants this presentation.

For a new language feature to make a difference and become fully accepted by programmers, there are three prerequisites:

1. it has to fit naturally with the existing language concepts as well as the programmer's preconceptions
2. it has to provide a clear advantage when used (be it in code size, elegance, maintainability, robustness or otherwise)
3. it must not have an overhead cost when used (otherwise programmers will be tempted to use more efficient, lower level methods)

The enhancements described here are all implemented and have been part of the ECL^{i}PS^e constraint logic programming system [4] since 1998. They are and have been in everyday use by a substantial number of programmers and are, to the author's best knowledge, well accepted and appreciated by the users of the system. Although we report experiences with a constraint logic programming system, rather than plain Prolog, this distinction is not essential for our presentation. The stronger declarative character of constraint logic programming does however make the proposed features even more attractive in that setting.

The rest of the paper is divided into motivation, introduction of the loop construct, comparison with higher-order approaches, comparison with bounded quantification, and a discussion of some remaining semantic issues.

2 Motivation

Our work was motivated by our observation of different classes of Prolog users. First, newcomers to Prolog, who already struggle with the unfamiliar concepts of logical variables and backtracking, are uncomfortable with the requirement to do everything by recursion - the hope was that an iteration construct would lower the threshold for them. Second, in our particular area of research, we had a need to convince mathematical programmers that one could model mathematical optimization problems in Prolog - the loop construct (together with additional support for arrays) helped us in this respect. Third, there are well-known software engineering issues when Prolog is being used for larger applications. Although the features described here present only a part of our efforts to address those issues, they do make a contribution by helping to make programs more readable, understandable and thus easier to maintain.

3 Loops

A look at the average Prolog program shows that the vast majority of all recursions are in fact iterations. Most of them are iterations over lists, some are iterations over integers (which may or may not represent structure/array indices), and very few iterate over other structures or index sets.

Not only novice programmers, and programmers that have been trained on imperative languages, resent to being forced to express everything through recursion. Seasoned Prolog programmers also find it tedious to have to write an

auxiliary predicate for every iteration. We note that these auxiliary predicates often serve no other purpose: they are only invoked once, they are not useful on their own, they are not abstractions of any useful concept. Incidentally, this often makes it difficult to invent sensible names for these predicates: in practice they often just inherit the parent predicate's name adorned with some suffix.

Our initial idea for improving the situation was to provide a comprehensive library of higher-order primitives (see e.g. [7]). A limited version of such a library had been available before, providing efficient versions of the basic higher-order predicates like map/3, foldl/4, filter/3 etc. However, this library was under-used and never seemed to provide quite the right tool. During the redesign, we soon realised that going beyond these basics would require to consider additional concepts like lambda expressions, composition of higher-order predicates, and program transformation for efficiency. This however, would have conflicted with our initial objective: it was unlikely that anything based on such complex concepts would be readily accepted by novices, and it was doubtful whether as such it would have constituted a simplification compared to the recursions we wanted to replace.

3.1 Iteration

Our eventual solution is more easily explained as a shorthand for common programming patterns (we will discuss its relationship with the higher order approach later in section 6). Given the importance of iteration, and the ubiquity of loop constructs in imperative languages, why should it not be possible to come up with a loop construct that would take into account the particularities of logic programming?

Some requirements were clear. We definitely wanted to be able to replace the verbose and tedious

```
write_list(List) :-
    write("List: "),
    write_list1(List).

write_list1([]).
write_list1([X|T]) :-
    write(X),
    write_list1(T).
```

with something straightforward like

```
write_list(List) :-
    write("List: "),
    ( foreach(X,List) do write(X) ).
```

This can obviously be implemented quite easily by automatically generating a recursive auxiliary predicate from the do-loop construct, and replacing the do-construct with a call to this auxiliary.

Similarly, we would like to iterate over the arguments of a structure by writing

```
?- ..., ( foreacharg(X,Structure) do write(X) ).
```

or over consecutive integers by writing

```
?- ..., ( for(I,1,100) do write(I) ).
```

3.2 Aggregation

Iteration is usually not done to perform side effects like in the above example, but to accumulate information, for instance compute the sum of list elements. In Prolog, one would use an accumulator pair of arguments to a recursive predicate as in

```
?- ..., sumlist(Xs, 0, Sum).

sumlist([], S, S).
sumlist([X|Xs], S0, S) :- S1 is S0+X, sumlist(Xs, S1, S).
```

In our shorthand notation, we introduce the fromto-specifier which is used in the following way:

```
?- ..., ( foreach(X,Xs),fromto(0,S0,S1,Sum) do S1 is S0+X ).
```

The intuition is that the aggregation process starts *from* the constant `0` and eventually gets *to* the resulting `Sum`. In between, each individual iteration step starts from `S0` (the result that has been accumulated so far) and computes the next intermediate result `S1`.

3.3 Mapping

Mapping (in the sense of the higher-order predicate map/3) means to establish a relationship between two lists (or arrays) by stating that a particular relationship holds between all pairs of corresponding list elements. Corresponding elements are those that occupy the same position in their respective list. Mapping is an extremely common concept in logic programming.

In Prolog, mapping is achieved in a straightforward way by recursing over two lists in parallel:

```
?- ..., one_up(Xs, Ys).

one_up([], []).
one_up([X|Xs], [Y|Ys]) :- Y is X+1, one_up(Xs, Ys).
```

Unlike in functional programming, mappings in Prolog have the nice property that they work in multiple modes (as long as the predicate establishing the relationship between the elements works in multiple modes). This means they can be used to test a relationship between two lists, to construct the second list given the first, construct the first list given the second, or even generate all valid pairs of lists.

Mapping can be covered simply by extending our loop syntax to allow iteration over two lists at the same time, allowing us to write the above example as

```
?- ..., ( foreach(X,Xs),foreach(Y,Ys) do Y is X+1 ).
```

Obviously, every *foreach* specifier corresponds to one argument in the recursive formulation. It is therefore a simple generalisation to allow an arbitrary number of *foreach* specifiers in one loop, this allowing iteration over many lists in parallel. In terms of higher-order predicates, this corresponds to map/4, map/5 etc. For example the above example can be generalised to

```
?- ..., ( foreach(X,Xs),foreach(Y,Ys),foreach(Z,Zs) do Z is X+Y ).
```

3.4 Full Functionality

The ideas introduced above are the basic ingredients for our general loop construct: It has a close correspondence to a simple tail-recursive predicate, and it can have one or more *iteration specifiers*, each of which corresponds to one or two arguments in a recursive predicate. The general form of a logical loop is

`(` ***IterationSpecs*** `do` ***Body*** `)`

where *IterationSpecs* is a comma-separated sequence of iteration specifiers, and *Body* is a general goal (possibly compound). Valid iteration specifiers (in our actual implementation) and their informal meanings are

fromto(First,In,Out,Last) Iterate *Body* starting with In=First and stopping with Out=Last. In and Out are local variables in *Body*.

foreach(X,List) Iterate *Body* with X ranging over all elements of List from first to last. X is a local variable in *Body*. This can be used both for iterating over an existing list or for constructing a new list.

foreacharg(X,StructOrArray) Iterate *Body* with X ranging over all arguments of StructOrArray from left to right. X is a local variable in *Body*. Cannot be used for constructing a term.

for(I,MinExpr,MaxExpr) Iterate *Body* with I ranging over numbers from MinExpr to MaxExpr. I is a local variable in *Body*. MinExpr and MaxExpr can be arithmetic expressions. Can be used only for controlling iteration, i.e. MaxExpr cannot be uninstantiated.

for(I,MinExpr,MaxExpr,Step) The same as above, but a step width different from 1 can be specified.

count(I,Min,Max) Iterate *Body* with I ranging over ascending integers from Min up to Max. I is a local variable in *Body*. This is similar to the for-specifier, but its main use is for counting iterations rather than controlling them, i.e. Max can be uninstantiated.

param(Var1,Var2,...) For declaring variables in *Body* as shared with the context. By default, variables in *Body* are local. For a more detailed discussion see section 8.

In principle, the *fromto* specifier alone would be sufficient: it is the most general one and can be used to express all the others (which we leave as an exercise for the interested reader). On the other hand, one could have introduced even more specifier shorthands, for instance a reverse list iterator, or a list suffix iterator. We have settled with the above set because it provides reasonably intuitive shorthands for the most common cases, in addition to the general *fromto*.

4 Transformation scheme

We now give the precise semantics of our loop construct by transformation to plain Prolog. Every goal of the form

`(` ***IterationSpecifiers*** `do` ***Body*** `)`

is substituted by a goal

PreCallGoals**`,`** ℓ`(`***CallArgs***`)`

where ℓ is a new, unique predicate symbol, ***CallArgs*** is a sequence of arguments to ℓ, and ***PreCallGoals*** is a possibly empty conjunction of goals to be executed before the call to ℓ. In addition, the transformation creates a definition for ℓ which is always of the following form[1]:

ℓ`(`***BaseArgs***`) :- !.`
ℓ`(`***HeadArgs***`) :-` ***PreBodyGoals*****`,`** ***Body*****`,`** ℓ`(`***RecArgs***`).`

Here, ***BaseArgs***, ***HeadArgs*** and ***RecArgs*** are sequences of arguments, ***PreBodyGoals*** is a possibly empty conjunction of goals, and ***Body*** is a literal copy of the original loop body.

Figure 1 shows a detailed tabular rendering of the translation rules. The translation algorithm looks up the matching rule for every specifier and collects each specifier's contribution to the six argument/goal sequences. Finally, a loop replacement goal and an auxiliary predicate definition are assembled from these collected sequences. The order of the specifiers in the do-construct is not important[2]. It is merely a specification of what the loop iterates over.

Example Consider the following loop with three iteration specifiers:

```
?- ( foreach(X,List), count(_,1,N), fromto(0,S0,S1,Sum) do
       S1 is S0+1
   ).
```

According to the above specification, the transformation results in the following goal and predicate definition:

```
?-  From is 1-1,                    % PreCallGoals
    do_1(List, From, N, 0, Sum).    % Initial call

do_1([], _1, _1, _2, _2) :- !.      % Base clause
do_1([X|_1], _2, _3, S0, _4) :-     % Recursive clause head
    I is _2 + 1,                    % PreBodyGoals
    S1 is S0+1,                     % Body
    do_1(_1, I, _3, S1, _4).        % Recursive call
```

[1] The cut (!) in the definition should be ignored for now, its role is discussed in detail in section 8

[2] Although with some compilers the indexing in the auxiliary may be affected

Translation Scheme for Iteration Specifiers

Iteration Specifier ***IterationSpecifier***	Transformation-time condition	Pre-Call goals ***PreCallGoals***	Initial call arguments ***CallArgs***	Base clause head arguments ***BaseArgs***	Recursive clause head arguments ***HeadArgs***	Pre-Body goals ***PreBodyGoals***	Recursive call arguments ***RecArgs***
`fromto(`***From***`,`***I0***`,`***I1***`,`***To***`)`	`nonground(`***To***`)`		***From***`, `***To***	`L0, L0`	***I0***`, L1`		***I1***`, L1`
`fromto(`***From***`,`***I0***`,`***I1***`,`***To***`)`	`ground(`***To***`)`		***From***	***To***	***I0***		***I1***
`foreach(`***X***`,`***L***`)`			***L***	`[]`	`[`***X***`|T]`		`T`
`foreacharg(`***A***`,`***S***`)`		`functor(`***S***`,_,N), N1 is N+1`	***S***`, 1, N1`	`_, I0, I0`	***S***`, I0, I2`	`I1 is I0+1, arg(I0,`***S***`,`***A***`)`	***S***`, I1, I2`
`count(`***I***`,`***FromExpr***`,`***To***`)`	`var(`***I***`), integer(`***To***`)`	`From is `***From-Expr***`-1`	`From`	***To***	`I0`	***I*** `is I0+1`	***I***
`count(`***I***`,`***FromExpr***`,`***To***`)`	`var(`***I***`)`	`From is `***From-Expr***`-1`	`From, `***To***	`L0, L0`	`I0, L1`	***I*** `is I0+1`	***I***`, L1`
`for(`***I***`,`***FromExpr***`,`***To***`)`	`var(`***I***`), number(`***To***`), Stop is `***To***`+1`	`From is min(`***FromExpr***`, Stop)`	***From***	`Stop`	***I***	`I1 is `***I***`+1`	`I1`
`for(`***I***`,`***FromExpr***`,`***ToExpr***`)`	`var(`***I***`)`	`From is `***From-Expr***`, Stop is max(From, ToExpr+1)`	`From, Stop`	`L0, L0`	***I***`, L1`	`I1 is `***I***`+1`	`I1, L1`
`param(`***P***`)`			***P***	***P***	***P***		***P***

Fig. 1. Translation scheme for Iteration Specifiers

The meta-level variables in bold italic style (e.g.***From***) stand for arbitrary terms occurring in the source program. The symbols T, N1, I0, I1, I2, L0, L1, Stop represent auxiliary variables which get introduced by the transformation (new instances for every iteration specifier). Specifiers which do not match any of these rules are treated as compile-time errors. The translation of the for/4 specifier has been omitted for space reasons.

Implementation In our system, this transformation is normally performed by the inlining facility of the compiler. In most Prolog systems, a similar effect can be achieved by means of the *term_expansion* mechanism.

In case a do-loop is constructed at runtime and meta-called, the system performs the same transformation, but meta-calls (i.e. interprets) the resulting code rather than actually generating a recursive predicate. One one hand, this is based on the guess that the compilation overhead might outweigh the gains when the loop is only run once, on the other hand this avoids the generation of an unbounded number of auxiliary predicates, and related garbage collection issues.

5 Loops vs Recursion

As opposed to the equivalent recursive formulation, the loop construct has a number of advantages which as (not necessarily in order of importance):

Conciseness No need to write an auxiliary predicate, in particular no need to invent a name for the recursive predicate, and no need to worry about the arity of the recursive predicate. This leads to 2-3 times shorter code (in terms of token count) in the above examples[3].

Modifiability If an additional value needs to be computed by the iteration, rather than having to add an argument or an accumulator pair in 4 places in the code (call, base clause, recursive clause head, recursive call), a single iteration specifier is added to the loop.

Structure Loops can be freely nested. This will usually show the code structure more clearly than a flat collection of predicates. Also, the iteration specifiers group conceptually related information better than scattered predicate arguments.

Abstraction An iteration specifier is an abstraction for a single induction argument or an accumulator pair. For example in this efficient list reversal predicate

```
reverse(L, R) :-
   ( fromto(L,[X|Ls],Ls,[]),fromto([],Rs,[X|Rs],R) do true ).
```

the first fromto translates into a single argument of do_2/3, while the second translates into an argument pair:

```
reverse(L, R) :- do_2(L,[],R).
do_2([],R,R) :- !.
do_2([X|Ls],Rs,R) :- do_2(Ls,[X|Rs],R).
```

The programmer does not need to be concerned about this detail. Both fromto-specifiers look completely symmetric, they graphically specify the order (from L to [], and from [] to R) in which the two lists are being traversed. In fact the predicate works both ways.

[3] assuming the common case that no recursive predicate was called more than once

Usability Although we have only anecdotal evidence, loops have clearly become very popular among the users of our implementation. Fears, that the additional feature would confuse new users more than it helped, seem to have been unjustified. On the contrary, it seems that loops with fromto-specifiers help with understanding the equivalent concept of accumulator pairs in recursive code.

6 Loops vs Higher-Order Constructs

Our loop construct offers an alternative to three of the most commonly used higher-order programming constructs which have found their way from functional programming into logic programming: map/3, foldl/4 and filter/3 (cf. [7]). For instance:

```
map(plus(1),Xs,Ys)      ( foreach(X,Xs), foreach(Y,Ys) do
                            plus(1, X, Y)
                        )

foldl(plus,Xs,0,Sum)    ( foreach(X,Xs), fromto(0,S0,S1,Sum) do
                            plus(X,S0,S1)
                        )

filter(<(5),Xs,Ys)      ( foreach(X,Xs), fromto(Ys,Ys1,Ys0,[]) do
                            ( X > 5 -> Ys1=[X|Ys0] ; Ys1=Ys0 )
                        )
```

In those simple instances, the loop formulation is somewhat more verbose than the higher order one. The reason for this is that the higher order formulation relies on an auxiliary predicate with a fixed argument convention (e.g. plus/3 with the last two arguments being the input and output of the mapping). Except in lucky circumstances, this auxiliary predicate will have to be purpose-written for each use, a development overhead that we would rather avoid.

Let us therefore consider a more flexible higher-order formulation with lambda-terms (a syntax for anonymous predicates similar to the one introduced in [12]), which would avoid the need for the auxiliary, e.g.

```
foldl(lambda([X,S0,S1], S1 is S0+X), Xs, 0, Sum)
```

This formulation is now not only of the same length, but also structurally very similar to our loop formulation. In fact there is a one-to-one correspondence of constants and variables:

```
foreach(X,Xs), fromto(0,S0,S1,Sum) do S1 is S0+X
```

Which version is preferable to a programmer is partly a matter of taste and will depend on training and on experience with other languages and programming paradigms. It can however be argued that the loop formulation exhibits a clearer grouping of related items: the variables X and Xs which are related to the list

iteration aspect, and the variables S0,S1,Sum together with the constant 0 which are related to the aggregation aspect of the code fragment.

A disadvantage of the higher-order constructs is the need for more and more: map/4, map/5, and combinations like map_foldl/5, etc. are frequently needed, but the provision of all these special cases can only be avoided by sophisticated program transformation, or the use of auxiliary data tuples [7]. In contrast, the same loop construct can be used for all these generalisations and combinations.

While it is clear that the higher-order approach has other uses, we argue that, for expressing iterations, our loop construct is preferable because

1. the loop construct can replace the vast majority of recursions in a form that is similarly compact and at the same time more explicit in stating the programmer's intent. Iteration seems an important enough concept to warrant a special language construct.
2. higher-order constructs implement arbitrary (not necessarily iterative) traversals of a particular data structure, while loops implement only iterative traversals but over arbitrary data structures.
3. the loop construct encourages the use of an efficient form of recursion, viz. tail recursion. This contrasts with the higher order approach which makes the inefficient foldr and the efficient foldl look interchangeable.
4. the single loop language construct covers the ground of several higher-order predicates families: map/(2+N), foldl/(2+2N), filter/(2+N).
5. the loop construct provides the building blocks to formulate arbitrary combinations of map / foldl / filter.
6. arguably, both novices and experienced programmers have fewer problems reading and understanding the meaning of a loop than they have understanding the meaning of a higher-order formulation.
7. unlike the higher-order solution, no complex higher-order typing is involved.

7 Loops vs Bounded Quantification

Voronkov [11], Barklund and Bevemyr [2], Barklund and Hill [3], as well as Apt [1] have advocated the introduction of bounded quantifiers. Their motivation is similar to ours: to express iteration more concisely, and in a way that is often closer to the original specification.

A bounded quantification requires a single finite set (e.g. an integer range, the elements of a list, etc.) over which the quantification ranges. But while [2] and [1] consider only the case where the quantification is bounded *a priori*, [11] is more general in allowing the termination condition to be depend on the quantified formula itself. The latter is what makes Voronkov's language Turing-complete, even without recursion.

Although bounded quantifiers do provide a significant gain in expressive elegance, many simple tasks cannot be expressed at all using *a priori* bounded quantification, and are still difficult to express in Voronkov's more powerful language. Consider the simple problem of determining that two lists are identical (or satisfy any other mapping property in the sense of map/3). This cannot be

expressed with *a priori* bounded quantification because there is no way to express which list elements correspond to each other. The obvious workarounds that come to mind are either to quantify over a list of pairs, or to convert the lists to arrays and then to quantify over the array index. However, this just raises the equally unsolvable problems of how to construct a list of pairs from two simple lists, or of how to construct an isomorphic array from a list.

With Voronkov's quantifiers the formulation is of course possible, but the best we could come up with is the following rather unnatural solution, employing the list-suffix-quantifier:

```
same_lists(XXs, YYs) :-
   SameTails = [XXs-YYs|_],
   (∀ T ⊑ SameTails) T = [[]-[]] ∨ T = [[X|Xs]-[X|Ys],Xs-Ys|_].
```

The loop construct overcomes this problem simply by having a concept of implicitly ordered iteration steps, and by allowing multiple iteration specifiers to synchronously traverse multiple data structures or index ranges[4]:

```
same_lists(XXs, YYs) :-
    ( foreach(X,XXs), foreach(Y,YYs) do X=Y ).
```

It can be argued that these difficulties with expressing mappings in the quantifier approach are partially overcome (or obscured) by the use of arrays. When arrays are used instead of lists, the above example is easily expressed, by quantification over a common array index. Index positions are a way to establish mappings explicitly, but work only with array-like data structures. It is therefore no coincidence that all the work on *a priori* bounded quantification has found it necessary to introduce arrays as a supporting feature.

Iteration specifiers in loops also play the role of the aggregation operators employed by [2], [3] and [1]. Without aggregation operators, *a priori* bounded quantifiers are very limited in their expressive power. Note again that Voronkov [11] does not need aggregation operators, because his quantifiers (like our iteration specifiers) can play this role as well.

In our loops, the general *fromto*-specifier, the list iterator *foreach* and the integer iterator *count* can all serve either as quantifiers (controlling the iteration) or as aggregators. We have deliberately avoided to define additional specifiers that would serve only as aggregators, because arbitrary aggregators can be so easily expressed using the general *fromto*. For example, the arithmetic maximum-aggregator is

```
maxlist([X0|Xs], Max) :-
    (   foreach(X,Xs), fromto(X0,M0,M1,Max) do
        ( X > M0 -> M1 = X ; M1 = M0 )
    ).
```

[4] We suggest that the more mathematically inclined reader pronounces all occurrences of the word 'do' in our code as 'holds' or ':' in order to eliminate the procedural taste

The cited works on bounded quantification investigate not only universal quantifiers but also existential quantifiers. We have not addressed this issue here at all, but it would seem that, unless one provides means to choose different control strategies for every existential quantifier, these do not give any significant advantage over the use of member/2 for existential quantification over list elements, or between/3 for existential quantification over integers.

8 Remaining Issues

Variable Scope in Loop Bodies One aspect that can potentially cause confusion is that the loop body is really an embedded predicate body with its own local variable scope. The following incorrect code illustrates this. The two occurrences of 'Array' are different variables:

```
sum_array(Array, N, Sum) :-
    ( for(I,1,N), fromto(0,S0,S1,Sum) do
        arg(I, Array, Elem), S1 is S0 + Elem
    ).
```

The programmer instead has to write

```
sum_array(Array, N, Sum) :-
    ( for(I,1,N), fromto(0,S0,S1,Sum), param(Array) do
        arg(I, Array, Elem), S1 is S0 + Elem
    ).
```

This situation has no direct counterpart in normal Prolog. The closest analogy are the bagof/setof predicates which allow locally quantified variables[5]. Our reason to opt for the opposite default (variables are quantified locally inside the loop, unless passed as *param*) was that this makes the loop semantics independent of the context (the presence or absence of a variable in the loop context will not affect the loop semantics). This not only relieves the compiler from the need to analyze the loop context, it also makes meta-calling of loops feasible.

The programmer needs to be aware that every iteration of the loop corresponds to a new instance of the loop body. Given the single assignment property of Prolog variables, this seems to be sufficiently intuitive, and our experience suggests that programmers do not have a problem with this. The only problem that does arise in practice is that the programmer forgets to specify the global variables. Fortunately, in many cases this situation leads to singleton variables in the loop body and our compiler gives a warning, suggesting that a *param* might be missing.

Nondeterminism Another point we have glossed over in the above is that our loop transformation, as implemented, always puts a cut into the base clause, i.e. the transformation template for the recursive predicate is in fact

[5] At the expense of a considerable implementation overhead

```
aux(...) :- !.
aux(...) :- ..., aux(...).
```

Most of the time, this makes no difference to the semantics (i.e. it is a so-called *green cut*). It does of course prevent applications where the number of iterations is nondeterministic and increases on backtracking.

Nevertheless we consciously made this restriction. The reason was that we would have been carried too far away from the intuitive idea of a loop. A choicepoint left by the loop construct itself would normally be unexpected and most likely constitute a (hard to find) bug. This is in fact analogous to Prolog's if-then-else construct `(... -> ... ; ...)`, where choicepoints within the condition are also cut, for very similar reasons.

Note that, of course, the loop body can be nondeterministic and generate multiple solutions. All we prevent is the number of iterations being nondeterministic.

Termination The termination condition of our loops is restricted to unification, or a conjunction of unifications. Loops that are terminated by a more complex condition can be expressed, but only indirectly.
Consider the iteration pattern:

```
p(... X0 ...) :-
    ( termination_condition(X0) ->
        true
    ;    ...,
        p(... X1 ...)
    ).
```

This can be expressed through a loop by introducing an explicit control variable `Continue`:

```
(
    fromto(continue, _, Continue, stop),
    fromto(..., X0, X1, ...), ...
do
    ( termination_condition(X0) ->
        Continue = stop
    ;   ...,
        Continue = continue
    )
)
```

This is rather unnatural, and in such cases the use of the loop construct will often not be appropriate. The example also shows that termination of loops with fromto-specifiers is not decidable in the general case. But in many special cases (where iteration specifiers correspond to *a priory* bounded quantifiers) termination is trivially guaranteed. This is true for the *foreacharg* and the *for* specifier, and for the *foreach* specifier when the list position is instantiated to a proper list.

Computational Power Prolog with logical loops, but without recursion, is still Turing-complete. The following is a recursion-free meta-interpreter for pure Prolog:

```
solve(Q) :-
    ( fromto([Q], [G|C0], C1, []) do solve_step(G, C0, C1) ).

solve_step(true, C, C).
solve_step((A,B), C, [A,B|C]).
solve_step(A, C, [B|C]) :- clause(A, B).
```

Typing Unlike most of the related work, we have not found it inevitable to introduce typing into our language. In the bounded quantifier framework, typing is used for specifying the semantics, in particular the domains over which the quantifiers range. In our loop framework, the semantics is formally defined by way of program transformation, which does in itself not provide a motivation for typing.

9 Conclusion

We have presented an addition to the Prolog programming language that makes programs more concise, more readable, easier to modify, less error-prone and more accessible to newcomers. Iteration often makes it possible to express a problem in a way that is closer to the original problem specification and also closer to the programmer's intuition.

We have not introduced any fundamental change to the language. In particular, we have neither introduced typing nor a concept of function evaluation that goes beyond what is already present in the basic language. Our loop construct can be entirely specified in terms of preprocessing, but is also easy to understand directly.

We have argued that our proposal is closely related to certain well-known higher-order constructs, but can have advantages over a corresponding higher-order formulation. Similarly, we have looked at the relationship with bounded universal quantifiers and shown that our approach in many cases allows a more natural formulation.

One direction of future work could be to look at ways to compile iterations more efficiently than the equivalent recursion. We would expect the techniques investigated in [2] to be applicable to our language. We intend to make the full loop transformation code available under `http://www.icparc.ic.ac.uk/eclipse/software/loops/`.

10 Acknowledgements

I would like to thank Mark Wallace and Stefano Novello for many discussions on the subject. Part of the work presented here was done in the context of the CHIC-2 project and I would like to thank our partners, in particular at EuroDecision, for

motivating me to make Prolog more suitable for mathematical modelling. Further thanks to Carmen Gervet, Kish Shen, Josh Singer and Warwick Harvey for their comments on earlier drafts of this paper.

References

1. K. R. Apt. Arrays, bounded quantification and iteration in logic and constraint logic programming. *Science of Computer Programming*, 26(1-3):133–148, 1996.
2. J. Barklund and J. Bevemyr. Prolog with arrays and bounded quantifications. In A. Voronkov, editor, *Proceedings of LPAR'93*, pages 28–39. Springer, 1993.
3. J. Barklund and P. Hill. Extending Gödel for expressing restricted quantifications and arrays. Technical Report No. 102, Uppsala University, March 1995.
4. ECLiPSe Team. ECLiPSe User Manual Version 4.0. Technical report, IC-Parc, Imperial College, London, July 1998.
5. M. Hanus. Curry: An integrated functional logic language. Technical report, Univerity of Kiel, Kiel, Germany, June 2000.
6. P. Hill and J. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
7. L. Naish. Higher-order logic programming in Prolog. Technical Report 96/2, University of Melbourne, Feb. 1996.
8. J.-F. Puget. A C++ implementation of CLP. In *Proceedings of SPICIS 94*, Singapore, November 1994.
9. Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.
10. P. Van Roy. Logic programming in Oz with Mozart. In D. D. Schreye, editor, *International Conference on Logic Programming*, pages 38–51, Las Cruces, NM, USA, Nov. 1999. The MIT Press.
11. A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming, First and Second Russian Conference*, pages 486–514. Springer LNAI, 1990/1991.
12. D. H. D. Warren. Higher-order extensions to Prolog - are they needed? *Machine Intelligence*, 10:441–454, 1982.