



## **Logical Loops**

**Joachim Schimpf**

**ICLP 2002**

# Overview

---

- **A general iteration construct for Prolog and Prolog-based (CLP) languages**
- **Definition via program transformation**
- **Comparison with alternatives**
  - Recursion
  - Higher-order constructs
  - Bounded Quantifiers
- **Expressive Power**
- **Experience**

# Motivation

---

## Introduced in ECLiPSe to support

- **Prolog beginners**  
Provide a familiar feature: loops
- **Application writers**  
Improved productivity, code structure, maintainability
- **Constraint Problem Modellers**  
Provide equivalent of bounded quantification

## Iteration (over list elements)

---

### ● Replace this

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

### ● With this

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

## Iteration (over range of numbers)

---

### ● Replace this

```
write_nat(N) :-  
    write("Numbers: "),  
    write_nat1(0,N).
```

```
write_nat1(N, N) :- !.
```

```
write_nat1(I0, N) :-  
    I is I0+1,  
    write(I),  
    write_nat1(I, N).
```

### ● With this

```
write_nat(N) :-  
    write("Numbers: "),  
    ( for(I,1,N) do  
        write(I)  
    ).
```



# Aggregation

---

## ● Replace this

```
sumlist(Xs, S) :-  
    sumlist(Xs, 0, S).  
  
sumlist([], S, S).  
  
sumlist([X|Xs], S0, S) :-  
    S1 is S0+X,  
    sumlist(Xs, S1, S).
```

## ● With this

```
sumlist(Xs, S) :-   
    (   foreach(X, Xs),  
        fromto(0, S0, S1, S)  
    do   
        S1 is S0+X  
    ).
```

# Mapping

---

## ● Replace this

```
map :- ...,
      add_one(Xs, Ys), ...

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

## ● With this

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

**iterate**

**construct**

# Programming idioms covered

---

## Fully:

- Iteration
- Aggregation
- Mapping

## Partly:

- Filtering
- While-loop



# General Form

---

( *IterationSpecs* do *Body* )

sequence of



**foreach**(Elem, List)

list iterator / aggregator

**foreacharg**(Arg, Structure)

structure iterator

**count**( I, Min, Max)

integer iterator / aggregator

**for**( I, Min, Max [,Step])

numeric iterator

**param**(X1, X2, ...)

constant iterator

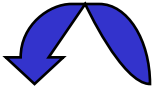
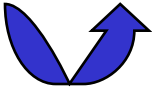
**fromto**(First, In, Out, Last)


generic iterator / aggregator

## The generic fromto-iterator

---

  
( fromto(**First**, In, Out, Last) do ... In→Out ... )

  
( fromto(First, **In**, **Out**, Last) do ... **In**→**Out** ... )  


( fromto(First, In, Out, **Last**) do ... In→Out ... )  


Fromto can express all other iterators, e.g.

fromto(**List**, [**X**|**Xs**], **Xs**, []) ⇔ foreach(**X**, **List**)

## Transformation scheme

---

A goal

..., ( *IterationSpecifiers* do *Body* ), ...

Is replaced by

..., *PreCallGoals*,  $\mu$ (*CallArgs*), ...

With an auxiliary predicate  $\mu$  defined as

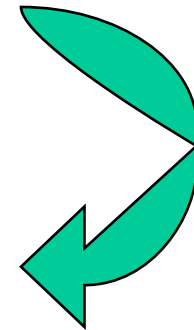
$\mu$ (*BaseArgs*) :- !.

$\mu$ (*HeadArgs*) :-

*PreBodyGoals*,

*Body*,

$\mu$ (*RecArgs*).



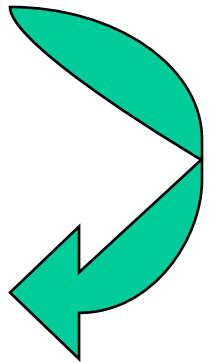
# Sample translation

```
?- ( foreach(X,List) , count(I,N0,N) , fromto(0,S0,S1,Sum) do
      S1 is S0+X
    ).
```

---

```
?- From is N0-1,                                % 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+X,                                  % Body
    do_1(_1, I, _3, S1, _4).                    % Recursive call
```



# Loops vs. Recursion

---

- **Conciseness**

  - Auxiliary predicate and its arity are hidden.

- **Modifiability**

  - Add/remove single iteration specifier rather than argument(s) in 4 places.

- **Structure**

  - Nested loops can be clearer than a flat collection of predicates.  
Iteration specifiers group related information together.

- **Abstraction**

  - Intention of iteration is explicit.

  - Fromto can help beginners understand the concept of accumulators.

# Abstraction

```
reverse(L, R) :-
```

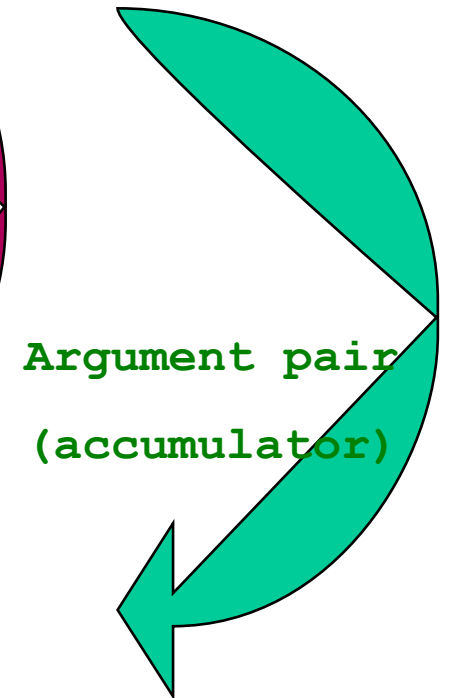
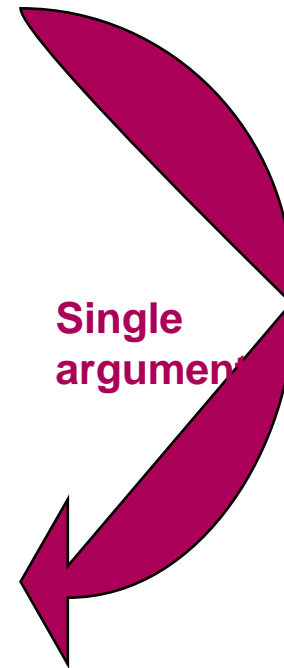
Symmetry

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

```
reverse(L, R) :- do_2(L, [], R).
```

```
do_2([], R, R) :- !.
```

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



# Loops vs. Higher-Order Constructs (1)

## Higher-Order constructs for similar tasks:

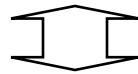
```
map (plus (1) , Xs , Ys)  ⇔ ( foreach (X, Xs) , foreach (Y, Ys) do
                             plus (1, X, Y)
                             )
foldl (plus , Xs , 0 , Sum) ⇔ ( foreach (X, Xs) , fromto (0, S0, S1, Sum) do
                             plus (X, S0, S1)
                             )
filter (<(5) , Xs , Ys)  ⇔ ( foreach (X, Xs) , fromto (Ys, Ys1, Ys0, []) do
                             ( X > 5 -> Ys1=[X|Ys0] ; Ys1=Ys0 )
                             )
```

## Loops vs. Higher-Order Constructs (2)

---

More realistic comparison with a lambda-term syntax:

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



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

Loop formulation has same size and better grouping



## Loops vs Higher-Order Constructs (3)

---

### Higher-Order map/fold/filter

- Data structure (list) specific
- Arbitrary traversal orders
- foldl / foldr symmetry
- Families needed: map/(2+N), fold/(2+2N), filter/(2+N)
- Combinations needed: map\_foldl/5, ...
- Higher-order-typed arguments

### Do-loop

- Any data structure (fromto)
- Only iteration / tail recursion
- Only efficient foldl-equivalent
- Arbitrary number of iteration specifiers allowed
- Arbitrary combinations of iteration specifiers allowed
- Same typing as recursive form

**Claim: Higher-order constructs are powerful, but not the best for iteration**

# Loops vs. Bounded Quantification

- **Quantification over finite sets**

Mathematical modelling languages (AMPL and others)

Extension to LP (Voronkov 90, Barklund et al 93/95, Apt 96)

`pos_list(List) :-  $\forall X \in \text{List}$  X > 0.`

`pos_array(A,N) :-  $\forall I \in 1..N$  A[I] > 0.`

- **But: mapping of lists is impossible**

`add_one_to_all(Xs,Ys) :-  $\forall X \in Xs \forall Y \in Ys$  Y is X+1.`

Does not work: No concept of “corresponding elements”

Needs arrays to be useful !

- **But: aggregation operators**

E.g. minimum, maximum, sum, ... must be introduced separately

`sum_list(Xs,S) :- S= $\sum X$  : X $\in$ Xs.`

# Expressive Power of Loops (1)

---

- We can write a while-loop, albeit awkwardly:

```
(
    fromto(cont, _, Continue, stop)
do
    ( termination_condition(...) ->
        Continue = stop
    ;   Continue = cont
    ),
    ...
)
```

## Expressive Power of Loops (2)

---

### 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).
```

# Evaluation criteria

---

## A language feature should

- **Fit with existing language concepts / user ideas**  
“Principle of least astonishment”
- **Provide a clear advantage**  
Code size, elegance, maintainability, robustness, ...
- **Not have an overhead cost**  
Otherwise programmers will use lower-level methods

# Programmer Acceptance

---

**Loops have been in ECLiPSe officially since 1998.**

**Analysis of commercial code developed with ECLiPSe:**

- **Application A (1997-2001)**

  - 254 predicate in 24 modules

  - 34 loops

- **Application B (2000-2002)**

  - 528 predicates in 35 modules

  - 210 loops

# Conclusion

---

- **Language extension with a minimum of new concepts**  
no types, modes, arrays were needed
- **Efficiently implementable**  
virtually no runtime overheads
- **Accepted by programmers**  
this has always been a problem with higher-order constructs
- **Translation code available at**  
<http://www.icparc.ic.ac.uk/eclipse/software/loops/>

# Rejected Ideas

---

- **Nondeterministic iteration**  
Once the termination conditions are met, the loop terminates.  
Allowing more iterations on backtracking seems too error-prone.
- **Better support for while/until-loops**  
Makes the semantics much more complex.
- **Clearly separate iterators and aggregators**  
Would lose some multi-directionality, but a good idea with modes.
- **More/less iterator shorthands**  
A matter of taste.



# Loops and Arrays

---

Array notation makes loops even more useful:

```
... /
( for (I, 1, N), param(Board, N) do
    ( for (J, I+1, N), param(Board, I) do
        Board[I] #\= Board[J],
        Board[I] #\= Board[J]+J-I,
        Board[I] #\= Board[J]+I-J
    )
) .
```

# Bounded Quantification

---

- **“A priori” Bounded Quantification**

- ( Barklund and Bevemyr (Reform Prolog) 1993 )

- ( Barklund and Hill (Gödel) 1995 )

- ( Apt 1996 )

- Mathematical modelling languages

Quantification over finite sets:

`pos_list(List) :-  $\forall X \in \text{List}$  X > 0.`

`pos_array(A,N) :-  $\forall I \in 1..N$  A[I] > 0.`

- **Bounded Quantification (Voronkov 1990)**

- More powerful due to list-suffix quantifier

- Termination can depend on quantified formula

- Turing-complete

## Mapping with Bounded Quantifiers (BQ)

---

- **Mapping of lists is impossible with *a priori* BQ**

`add_one_to_all(Xs, Ys) :-  $\forall X \in Xs \ \forall Y \in Ys \ Y \text{ is } X+1.$`

Does not work: No concept of “corresponding elements”

- ***A priori* BQ needs arrays**

`add_one_to_all(A, B) :-  $\forall I \in 1..N \ B[I] \text{ is } A[I]+1.$`

Indeed all authors introduce arrays into their proposed languages.

But still cannot map a list to an array ...

- **Voronkov’s full BQ can express list mapping**

However, very unintuitive

- **Loop solution:**

We have a concept of implicitly ordered iteration steps

# Aggregation with Bounded Quantifiers

---

- Aggregation not covered by basic *a priori* BQ
- Specific aggregation operators must be introduced  
e.g. minimum, maximum, sum, ...  
`sum_list(Xs,S) :- S=ΣX : X∈Xs.`
- **Loop solution:**  
The fromto-specifier can function as iterator and aggregator.  
Any aggregation function can be expressed.