Sequence Quantification

Peter Schachte

Department of Computer Science
The University of Melbourne
Victoria 3010
Australia

schachte@cs.mu.oz.au

Abstract. Several earlier papers have shown that bounded quantification is an expressive and comfortable addition to logic programming languages. One shortcoming of bounded quantification, however, is that it does not allow easy and efficient relation of corresponding elements of aggregations being quantified over (lockstep iteration). Bounded quantification also does not allow easy quantification over part of an aggregation, nor does it make it easy to accumulate a result over an aggregation. We generalize the concept of bounded quantification to quantification over any finite sequence, as we can use a rich family of operations on sequences to create a language facility that avoids the weaknesses mentioned above. We also propose a concrete syntax for sequence quantification in Prolog programs, which we have implemented as a source-to-source transformation.

1 Introduction

Prolog [8] has no standard construct for looping over recursive data structures or arithmetic sequences. Beginning Prolog programmers are often told that Prolog makes recursion very natural, so no looping construct is needed. While this is true, it is also true that a looping construct could make some programs clearer and more succinct. The absence of looping construct from Prolog is all the more surprising since the predicate calculus on which it is based has had two such constructs — universal and existential quantification — for more than 120 years [5].

In designing his automatic theorem proving framework, Robinson restricted his attention to clauses, proving resolution to be a sound and complete inference rule [12]. Kowalski, in proposing the paradigm of logic programming, recommended the further restriction to Horn clauses [9]. A Horn clause is a disjunction of one atom (an atomic predication) and zero or more negated atoms, though it is more commonly thought of as a conjunction of zero or more atoms (called the clause body) implying a single atom (the head). All variables in a clause are universally quantified over the whole clause. When a clause is viewed as an implication, however, variables not appearing in the clause head can be seen as existentially quantified over the clause body.

Prolog, however, was quick to loosen the Horn clause restriction. A disjunction in the body of a clause is easily accomodated by replacing the disjunction with an atom invoking a newly-created predicate whose definition comprises a clause for each disjunct. Negated atoms in a clause body can be handled using negation as failure to prove [4]. These are both now standard Prolog features.

Similarly, several earlier papers have suggested adding a restricted form of universal quantification, called *bounded quantification*, to logic programming languages. Bounded quantification is a form of universal quantification where the set of values quantified over is explicitly specified, adopting a shortcut common in much mathematical writing. Instead of writing

$$\forall x \,.\, x \in s \rightarrow p(x) \quad \text{or} \quad \exists x \,.\, x \in s \wedge p(x)$$

they write

$$\forall x \in s \,.\, p(x) \quad \text{or} \quad \exists x \in s \,.\, p(x)$$

Not only is this notation more concise, it seems quite natural to specify what a variable is to range over in the same place as how it is quantified.

Sadly, Prolog systems, and the Prolog language standard, have not been quick to accept bounded quantification. The aim of the present paper, then, is to propose a generalization of bounded quantification that is simultaneously more powerful and flexible than bounded quantification, yet still efficient, convenient, and reasonably portable.

The remainder of this paper is organized as follows. Section 2 reviews the history of bounded quantification and closely related work. Section 3 introduces the concept and semantics of sequence quantification. Section 4 discusses primitive sequences and how they are defined. It also discusses how sequences can be defined in terms of other sequences, providing the power behind sequence quantification. Section 5 presents the surface syntax we use for sequence quantification, and how the user can introduce new syntactic sugar, as well as addressing the subtle issue of resolving the scoping of variables not explicitly quantified. In section 6, we discuss our implementation of this facility, including a discussion of how we generate efficient code. Finally, we present our future work and concluding remarks in sections 7 and 8.

2 Bounded Quantification

Probably the oldest form of universal quantification in logic programming was the `all/2` [11] construct of NU Prolog [16]. Using this, one could write, *e.g.*, `all [X] p(X) => q(X)` to mean $\forall x \,.\, p(x) \rightarrow q(x)$. The most common use of this would be in what amounts to a restricted kind of bounded quantification: to check that every element of a list satisfies some constraint. Implementation was in terms of nested negation: the last example would be executed as `\+ (p(X), \+ q(X))`, except that NU Prolog would execute this negation even if `X` was not bound (ordinarily, NU Prolog suspends execution of non-ground negations). Since the implementation is in terms of negation, this effectively means that the `all` construct could only perform tests, and could not bind variables.

In the earliest paper we have found to discuss bounded quantification in English, Voronkov [18] gives a model-theoretical, least fixed point, and procedural semantics for typed logic programming with bounded quantification. In addition to supporting $\forall X \in S \,.\, p(X)$ and $\exists X \in S \,.\, p(X)$ where S is a list and X ranges over the elements of that list, he also defines $\forall X \sqsubseteq S \,.\, p(X)$ and $\exists X \sqsubseteq S \,.\, p(X)$ where X ranges over the tails of the list S. Voronkov also presents a translation from logic programs with bounded quantification to ordinary logic programs.

Barklund and Bevemyr [3] discuss bounded quantification in the context of arrays in Prolog. For this, they are more interested in quantifying over integer ranges, to be used as array indices, than lists. Thus rather than quantifying over forms such as $X \in S$, they quantify over $L \leq I < H$ forms, specifying that I ranges from L up to but not including H. Their focus is also practical: they have implemented their approach by extending the LUTHER WAM emulator with specialized features for bounded quantification. By quantifying over array indices, they are able to relate corresponding elements of two different arrays. This is a significant step forward, but it does not allow them to conveniently or efficiently relate array elements with elements of other structures, for example list or tree members. Another advance of this work is the inclusion of aggregation operators. In addition to quantifying over an integer range, they also allow computing the sum or product of an expression over such a range. They discuss numerous other such useful aggregations operators, but do not go so far as to suggest a general framework for defining them. They also observe that it may be possible to parallelize the execution of bounded quantifications, since the indices may be handled independently. Barklund and Hill [2] propose making a similar extension to Göedel.

Apt [1] gives many compelling example programs showing the power of bounded quantification in logic programming as well as constraint logic programming. Like Voronkov, he prefers typed logic programming.

The OPL language [7] provides a `forall` quantifier allowing iteration over integer ranges and enumerations, and allows the order of iteration to be explicitly specified. It also allows only some values, determined by an explicit test, to be iterated over. Additionally, it provides built in aggregations to compute the sum, product, minimum and maximum of a set of values. It does not appear to provide any facility for allowing iteration over any other domain, nor for lockstep iteration.

The Logical Loops package of the ECL^{i}PS^e system [14], developed independently of the present work, provides a facility similar similar to bounded quantification, although it eschews that label. Logical loops provide a number of iteration primitives, which can be combined to allow general lockstep iteration. One of the iteration primitives is much like our own aggregation facility (see Section 4.3), and allows fully general aggregation. In fact, this one primitive subsumes the functionality of all the other iteration primitives.

However, logical loops does not allow new iteration primitives to be defined; one must fall back on the less convenient general primitive. Nor does the package provide a primitive to iterate over part of a structure. Although the general iteration primitive can accommodate this, the author admits the technique is "rather unnatural." Finally, logical loops always commit to the fewest possible iterations. It is not possible for a logical loop to generate the list it is to iterate over or the upper bound on an arithmetic iteration, as it will commit to loop termination as early as possible.

3 Sequence Quantification

Our approach diverges from these by generalizing what data structures we can quantify over. Where the earlier works only allow quantification over integer ranges and lists, we wish to allow quantification over any *sequence* of values the user cares to define. Our belief is that a single, simple universal quantification operation, together with the ability to define new kinds of sequences to iterate over, provides a much more powerful facility than a larger set of quantification and aggregation operations limited in the constructs they quantify or aggregate over. The power and flexibility of this approach approaches that of iteration constructs in imperative languages, such as `for`, `while`, and `do` loops, without losing logic programming's declarative character. Similarly, it provides most of what is provided by the usual set of higher order functions in modern functional languages (see, *e.g.*, [17]).

The basic form of sequence quantification is much as for bounded quantification:

`forall` *variable* `in` *sequence* `do` *goal*

where *goal* is any Prolog goal, *variable* is a Prolog variable or term, and sequence is a term representing a sequence of values. (The actual syntax is slightly more general than this to allow for some syntactic sugar, as discussed in Section 5.)

We specify the semantics of sequence quantifications by translations similar to those of Voronkov [18], as shown in table 1. This translation differs from

Formula	Translation and extra clauses
$\exists y \, . \, F$	$p(vars(F) \backslash \{y\})$ with clause:
	$p(vars(F) \backslash \{y\}) \leftarrow F$
$x \in s$	$p(x, s)$ with clauses:
	$p(x, s) \leftarrow next(s, x, s')$
	$p(x, s) \leftarrow next(s, x', s') \wedge p(x, s')$
$\forall y \in s \, . \, F$	$p(vars(F) \backslash \{y, s\}, s)$ with clauses:
	$p(vars(F) \backslash \{y, s\}, s) \leftarrow empty(s)$
	$p(vars(F) \backslash \{y, s\}, s) \leftarrow next(s, y, s') \wedge F \wedge p(vars(F) \backslash \{y, s\}, s')$

Table 1. definition of quantifiers and sequence membership

Voronkov's in that we specify the semantics of $\exists$ generally, not only for bounded quantifications, and we specify the semantics of $\in$ outside the context of a quantification. The definition of $\in$ is essentially the standard Prolog `member/2` predicate, generalized to work on sequences. Our definition of universal quantification is similar to Voronkov's, but is generalized to work on any sort of sequence.

4 Sequences

A sequence is an ordered collection of terms, possibly with repetition. For our purposes, a sequence is characterized by two predicates: $empty(s)$ holds if s is an empty sequence, and $next(s, e, t)$ holds if e is the first element of sequence s, and t is the remainder of sequence s after e. We implement these in Prolog as predicates `sequence_empty/1` and `sequence_next/3`. These predicates may be defined by anyone,[1] meaning that users of the package can define their own sequences to quantify over; they are not restricted to the sequences already supported by the implementation.

4.1 Primitive Sequences

The most obvious kinds of sequences are lists and arithmetic sequences. These are defined as follows:

```
sequence_empty(list([])).
sequence_next(list([H|T]), H, list(T)).

sequence_empty(Low..High) :-
        High is Low-1.
sequence_next(Low..High, Low, Next..High) :-
        ( nonvar(High) -> Low =< High ; true ),
        Next is Low + 1.
```

Note the `list/1` wrapper around the list sequence. All sequences must have a distinguished wrapper indicating what kind of sequence they are. This permits sequences to be produced as well as consumed by quantifications. The `nonvar` test in the final clause ensures that when the upper limit on an integer range is available at run time, it will be used to ensure iteration does not exceed the limit, but when no limit is supplied, iteration can proceed indefinitely.

Many other types of sequences are possible. For example, the following clauses define `inorder(Tree)` as the sequence of the elements of `Tree` traversed inorder. We assume `empty` denotes the empty tree, and `tree(L,Label,R)` denotes a tree with root label `Label` and left and right subtrees `L` and `R` respectively.

```
sequence_empty(inorder(empty,[])).
```

[1] Here we make use of standard Prolog's `multifile` declaration, allowing users to define these predicates with clauses in any number of files.

```
sequence_next(inorder(tree(L,Label,R),Stack), First, Rest) :-
        sequence_next(inorder(L,[Label-R|Stack]), First, Rest).
sequence_next(inorder(empty,[First-R|Stack]), First,
              inorder(R,Stack)).
```

Similar definitions could be written for preorder and postorder traversal.

4.2 Sequence Operations

The power of sequence quantification becomes apparent when one considers the many ways sequences can be modified and combined.

One important facility that is not naturally supported by bounded quantification is lockstep iteration, *i.e.*, parallel quantification over two sequences. For example, we might wish to specify that a relation R holds between the corresponding elements of l and m. Given a function *zip* that maps two sequences of the same length to a sequence of pairs of their corresponding elements, we could express this as: $\forall \langle x,y\rangle \in zip(l,m) \,.\, R(x,y)$ Note that this approach is very similar to that taken by the Python language in providing its lockstep iteration facility [20].

Of course, logic programming does not have functions (at least Prolog does not), so to use this technique we would need to write something like:

```
zip(L, M, Pairs),
forall X-Y in Pairs do r(X,Y)
```

This unfortunately creates a separate list of pairs, wasting time and space. This shortcoming could be solved through deforestation [19], however this is not a common feature of Prolog compilers.

Our approach is instead to define a new kind of sequence constructed from two other sequences. This can be done by extending the definitions of `sequence_empty/1` and `sequence_next/3`:

```
sequence_empty((A,B)) :-
        sequence_empty(A),
        sequence_empty(B).

sequence_next((A,B), (A1,A2), (Ar,Br)) :-
        sequence_next(A, A1, Ar),
        sequence_next(B, B2, Br).
```

This allows us to write

```
forall (X,Y) in (L,M) do r(X,Y)
```

Note that this approach requires no deforestation for optimization; a much simpler program optimization, as discussed in section 6, can remove the unnecessary term constructions.

Functional programmers will recognize this relation as similar to the standard *map* function, which applies a function to each element of a list, collecting the results into a list. There are several differences, however. Firstly, the sequence quantification approach works on sequence of any sort, not just lists. Secondly, it generalizes to arbitrary numbers of sequences. We could as easily have written

```
forall (X,Y,Z) in (L,M,N) do r(X,Y,Z)
```

to specify that `r/3` relates corresponding elements of 3 sequences.

Sequence quantification also retains logic programming's relational character: it can be used to compute any of the sequences from any others, as long as the body of the quantification (the part after the `do`) can work in that mode. In the latter example, `L`, `M`, and/or `N` can be produced by this quantification, providing the first, second, and/or third argument of `r/3` can be output.

Voronkov [18] also defines quantification over tails of lists using the syntax $\forall T \sqsubseteq L \ldots$. We can achieve this effect by defining sequence operator *tails* which can be defined by:

```
sequence_empty(empty).

sequence_next(tails(S), S, Rest) :-
        (   sequence_next(S, _, Sr) ->
                Rest = tails(Sr)
        ;   Rest = empty
        ).
```

This allows us to define an ordered list similarly to Voronkov:

```
ordered([]).
ordered([_]).
ordered(L) :-
        forall Tail in tails(L) do
            (   Tail=[X,Y|_] ->
                    X=<Y
            ;   true
            ).
```

A better definition of this predicate will be presented in section 5.

Another operation we can perform on sequences is selecting only part of a sequence. For example, we can define a sequence filter: a sequence of the elements of another sequence with the elements not satisfying a specified filter predicate removed.

```
sequence_empty(when(S,F)) :-
        (   sequence_empty(S)
        ;   sequence_next(S, S1, Sr),
            \+ call(F, S1),
            sequence_empty(when(Sr,F))
```

```
        ).

sequence_next(when(S,F), N, Rest) :-
        sequence_next(S, S1, Sr),
        (   call(F, S1) ->
                N = S1,
                Rest = when(Sr,F)
        ;   sequence_next(when(Sr,F), N, Rest)
        ).
```

This can be used, for example, to count the positive elements of a sequence:

```
forall _ in (when(list(L),<(0)), 1..Count) do true
```

Here we specify that there are the same number of elements in `1..Count` as in `when(List,<(0))`. Note that the predicate `<(0)` will be given one more argument which will come last. Thus it specifies that zero is less than that argument.

Another useful sequence operator is `while`; this is similar to `when`, except that the sequence terminates once the first element not satisfying the given predicate is found. This can be used to simulate a *while* loop. For example,

```
forall (X,Y) in (while(inorder(Tree), >=(100)), list(list))
    do X = Y
```

will bind List to a list of the elements of Tree up to 100, stopping accumulation when the first element larger than 100 is found. Similarly, the `once` sequence operator specifies that the initial part of the sequence whose elements do not satisfy the specified test should be dropped, with all the remaining elements taken.

4.3 Accumulation

One common use of looping constructs in conventional programming languages is to accumulate a result. Barklund and Bevemyr propose a fixed set of accumulating quantifiers including `sum`, `product`, `min` and `max`. Thus far, sequence quantification only permits corresponding elements of sequences to be related, whereas accumulation requires elements to be related to previous elements of the sequence. We achieve this by introducing an accumulation sequence, which is a sequence of pairs, where the first element of each pair is equal to the second element of the previous pair. This is the only constraint on such a sequence, which we define as:

```
sequence_empty(thread(X,X)).

sequence_next(thread(Init,Final), (Init,Next),
              thread(Next,Final)).
```

We use such a sequence to accumulate by relating the two elements of the pair. We defer examples of accumulation to Section 5, where we introduce some syntactic sugar to make accumulations more attractive.

4.4 Scoping

One interesting question arises: how should variables not explicitly quantified be handled? For example, what does a goal like

```
forall X in L do p(X,Y)
```

mean? Is `Y` scoped to the `p(X,Y)` goal or to the whole clause? In the former case, it means $\forall x \in l \,.\, \exists y \,.\, p(x, y)$ whereas in the latter it means $\exists y \,.\, \forall x \in l \,.\, p(x, y)$.

Unfortunately, no single answer to this question seems intuitive in all cases. If the goal had instead been

```
Y=42, forall X in L do p(X,Y)
```

then it seems clear the intended scope of `Y` would be the whole clause. However, for the goal

```
forall X in L do (p(X,Y), q(Y))
```

the most natural reading would be that `Y` should be scoped inside the `forall` construct (i.e., this query should not require that `Y` be the same for all `X`s).

Therefore we adopt the strategy used in the Mercury language[6]: a variable appearing only inside a `forall` construct is existentially quantified inside the universal quantification, while other variables are existentially quantified over the whole clause body. When this does not achieve the desired effect, explicit existential quantification can be used to specify the desired scoping of certain variables. However, this is rarely necessary as the default behavior is almost always what the user intends and expects.

Compare this, for example, to the behavior of Prolog's `setof`/3 and `bagof`/3 predicates, which implicitly scope all variables not appearing in the template (first) argument to the whole clause. Thus the anonymous variable in the goal

```
setof(X, p(X,_), List)
```

is scoped wider than the `setof` goal, resulting in unintended behavior in this case. To scope the anonymous variable inside the `setof` goal, it must be rewritten as

```
setof(X, Y^p(X,Y), List)
```

Experience shows that this often cause confusion for inexperienced Prolog programmers.

Conversely, the logical loops package [14] implicitly scopes all variables appearing in a loop to the inside of the loop. A variable appearing both outside and inside a loop is considered to be two distinct variables, unless a `param` form is added to the loop to indicate that the variable is to be scoped outside the loop. This, too, seems likely to cause confusion.

The disadvantage of our approach is that it can cause problems in some cases when a sequence quantification is used as a parameter to a higher order predicate. However, it should be noted that sequence quantification is intended to replace most uses of higher order code, and that explicit quantification can always be used to specify the intended scoping.

5 Syntactic Sugar

For a practical implementation of sequence quantification as a Prolog extension, we believe it is important to provide a syntax which is intuitive and palatable. In this, we have followed the spirit of the CLISP package included as part of the INTERLISP language [13], with an emphasis on extensibility.

Firstly, we generalize the *Template* `in` *Sequence* form to allow other ways to specify the generation of a sequence of bindings for a template. This is accomplished by allowing users to define clauses for the `sequence_generator/3` predicate. When a goal of the form `forall` *Generator* `do` *Goal* is found,

`sequence_generator(`*Generator*, *Template*, *Sequence*`)`

is called, and the quantification is then treated as if it had been `forall` *Template* `in` *Sequence* `do` *Goal*.

This facility is used to provide some (hopefully!) more intuitive ways to write quantifications. Of course, these are only optional syntactic sugar; the syntax given earlier continues to work.

One example is the `as` operator. This allows each variable to be given together with the sequence it is quantified over, rather than requiring all the quantified variables to be bundled together and all the sequences to be bundled together. The clause:

```
sequence_generator(G1 as G2, (T1,T2), (S1,S2)) :-
        generate_sequence(G1, T1, S1),
        generate_sequence(G2, T2, S2).
```

together with an operator declaration for `as`, allows us to write:

```
forall X in list(L) as (S0,S) in thread(0,Sum) do S is S0 + X.
```

instead of

```
forall (X,S0,S) in (list(L),thread(0,Sum)) do S is S0 + X.
```

Note that `generate_sequence/3` just ensures that its first argument is not a variable, and then calls `sequence_generator/3`.

Another syntactic embellishment is provided by:

```
sequence_generator((Init->V0->V->Final),
                   (V0,V), thread(Init,Final)).
```

This provides an alternative syntax for `thread` sequences introduced in Section 4.3 that presents the current and next variables for a thread between the initial and final values, allowing us instead to code the previous example as

```
forall X in list(L) as (0->S0->S->Sum) do S is S0 + X.
```

Intuitively, this says that we accumulate a value beginning with 0 and winding up with Sum, and at each iteration S0 is the previous value and S is the next one. That is, Sum is the sum of the elements of list L.

Note that there is no requirement for there to be any relationship between the current and next variables in an accumulation, only that S be determined. For example, we could better define the ordered predicate of section 4.2 as follows:

```
ordered(Seq) :-
        (   sequence_empty(Seq) ->
                true
        ;   sequence_next(Seq, First, Rest),
            forall E in Rest as (First->Prev->This->_) do
                (   E >= Prev,
                    This = E
                )
        ).
```

Procedurally, this "initializes" Prev to the first element of Seq, and runs over the rest of the sequence verifying that each element is larger than Prev, and setting the value of Prev for the next iteration to the current element. This is similar to the code one might write in an imperative language, yet is entirely declarative.

Note that sequence generators can be used anywhere in the program, not only inside universal quantifications. When not immediately preceded by forall, they are considered to be in an existential context, and specify membership in the sequence. For example, a goal

```
X in list(L) as N in 1.._
```

would specify that X is an element of L and N is the position of X in that list.

6 Implementation

We have a Prolog implementation of sequence quantification built on the common term_expansion/2 Prolog extension. As each clause is compiled, it is scanned for explicit universal and existential quantifications. When such a goal is found, it is replaced by a call to a newly created predicate, plus the definition of that predicate.

The generated predicates are quite close to the form shown in table 1. A universal quantification of the form

forall T in S do *body*

would initially be translated to a goal

'forall i'(S)

where forall i is defined as:

```
'forall i'(S0) :-
        (   sequence_empty(S0)
        ;   sequence_next(S0, T, S)
            body,
            'forall i'(S)
        ).
```

Note that this code is entirely declarative if the *body* is. In particular, no cuts or if-then-else constructs are used to prevent backtracking. Thus a quantification may be used in any mode that the *body* and `sequence_next` and `sequence_empty` definitions will support. For example,

```
forall _ in list(L) as _ in 1..Length
```

will work to check that the length of `L` is `Length`, or to determine the length of list `L`, or to generate a list of length `Length`, or even to backtrack over longer and longer lists.

Several steps are taken to improve the efficiency of the generated code, the goal being to produce code as close as possible to the efficiency of the code an experienced Prolog programmer would write for this purpose. Firstly, users may specify improved code to generate for the empty and next predicates for particular kinds of sequences. This is done by providing clauses for the user-extensible predicates `empty_specialization/5` and `next_specialization/7`. These are certainly more complex than simply providing clauses for `sequence_empty/1` and `sequence_next/3`, but they are manageable. For example, to optimize the handling of list sequences, using `L=[]` and `L=[H|T]` in place of `sequence_empty(L)` and `sequence_next(L,H,T)`, these clauses suffice:

```
empty_specialization(list(L), _, L=[], Clauses, Clauses).
next_specialization(list(L), _, H, list(T), L=[H|T],
                    Clauses, Clauses).
```

In some cases, a sequence may be more efficiently handled if it is first transformed into another form of sequence. For example, the definition of `sequence_empty/1` for integer range sequences is:

```
sequence_empty(Low..High) :- High is Low-1.
```

This requires an arithmetic computation to be done for each iteration. We could define a more efficient integer range as:

```
sequence_empty(intseq(H,H)).
sequence_next(intseq(L,H), L1, intseq(L1,H)) :-
        ( (integer(H) -> L<H ; true),
        L1 is L+1.
```

But this is less intuitive since, *e.g.*, `intseq(1,10)` specifies the sequence *2*..10.

In such cases, users can supply a clause for the `sequence_specification/7` predicate which specifies a goal to execute before beginning iteration, a goal to execute at the end, and an alternative sequence to use. For example

```
sequence_specification((L..H), _, intseq(L1,H),
                       (L1 is L-1), true, Cl, Cl).
```

would substitute the sequence `intseq(L1,H)` for `L..H`, and insert the goal `L1 is L-1` before the call to the generated iteration predicate.

To make the job of writing this optimization code easier for those defining new sequences (and keen on achieving the highest performance), the translation performs a fairly simple-minded code specialization pass on the generated code. This pass looks for Prolog built in predicates that can be executed at compile time and replaces them with unifications achieving the same result. Furthermore, it executes unifications at compile time when they would unify a variable or atomic term with the first occurrence of a variable. In general, unifications should not be executed at compile time, since doing so may bind a variable used before a commit (cut or if-then-else) or in an impure operation, such as assert or input/output, changing the program behavior. It may also generate many occurrences of a large term where only one existed in a unification.

The most important optimization performed is in the generation of looping predicates. Rather than repeatedly taking apart and building sequence terms, the generated code passes the needed information in separate arguments. This is done by computing the most specific generalization of the head of the generated predicate and the call, and extracting its variables. Similarly, we compute the most specific generalization of each recursive call with the clause head.

Finally, we analyze the generated clause to see if one of its arguments is unified with a term with a distinct principal functor in each arm of the disjunction, and before any impure goal. If such an argument is identified, it is moved to the first argument position, and the disjunction is split into separate clauses, to take advantage of the first argument indexing supported by most Prolog implementations.

The resulting code is quite efficient. For the goal

```
forall _ in list(L) as _ in 1..N
```

which states that the length of list `L` is `N`, the generated code is `'forall 2'(L, 0, N)` where `'forall 2'/3` is defined by:

```
'forall 2'([], A, B) :-
        A=B.
'forall 2'([A|B], C, D) :-
        (   integer(D)
        ->  C<D
        ;   true
        ),
        E is C+1,
        'forall 2'(B, E, D).
```

The goal

```
forall E in list(L)
```

```
            as _ in 1..Count
            as (0->S0->S->Sum)
            do S is S0+E
```

which sums and counts the elements of a list, is translated to: `'forall 3'(L, 0, Count, 0, Sum)`, with:

```
'forall 3'([], A, B, C, D) :-
        A=B,
        D=C.
'forall 3'([A|B], C, D, E, F) :-
        (   integer(D)
        ->  C<D
        ;   true
        ),
        G is C+1,
        H is E+A,
        'forall 3'(B, G, D, H, F).
```

Finally, the goal

```
forall I in 1..10 do (write(I),nl)
```

translates to `'forall 5'(0)` with:

```
'forall 5'(A) :-
        (   A=10
        ;   A<10,
            B is A+1,
            write(B),
            nl,
            'forall 5'(B)
        ).
```

7 Future Work

Currently, adding an optimization for a new kind of sequence, as described in section 6, is more difficult than it should be. We are investigating abstractions that would allow simple definition of sequences that would automatically be optimized.

There is always scope for further improvement to the specializer. It should be possible to execute more unifications at compile time when one of the terms to be unified is the first occurrence of a variable occurring only once or twice in the clause. After performing the unification and removing the goal, the term will appear at most once. Also it should be possible to unify a subsequent occurrence of a term if no impure operation has been performed since the first occurrence.

A more important optimization would be to avoid repeating a goal in multiple arms of a disjunction. In some cases, `sequence_empty/1` may invoke the negation of `sequence_next`. In such cases, the iteration predicate will call `sequence_next` twice. It would be much more efficient to generate an if-then-else calling `sequence_next` once in the condition.

Finally, the design and implementation of a reasonably complete — but not overwhelming — set of sequence operators remains to be done. Fortunately, any operators omitted from the package can be supplied by the user without much difficulty.

We expect to release this package to the public under a suitable free or open source software license when it is completed. It will be available from

`http://www.cs.mu.oz.au/~schachte/software/`

8 Conclusions

We have described a new quantification formalism for logic programming which provides a logical facility allowing explicit iteration over any sort of sequence the user can define, as well as looping over multiple sequences in a coordinated manner, and arbitrary kinds of aggregations and accumulations. All of these things are done quite naturally using various kinds of sequences, and cannot in general be done with bounded quantification. Users can define their own kinds of sequences, including sequences defined in terms of other sequences. This facility has been implemented as a Prolog source to source translation.

Comparing the expressiveness of this facility with looping constructs of imperative languages, or with higher order operations in functional languages is quite difficult, inevitably leading one to compare apples with oranges, or to declaring all formalisms Turing equivalent. Still, since the primitive building block of this facility is simply the definition of a sequence of values, and since imperative looping constructs must loop over a sequence of values of the variables of the loop, it seems sequence quantification should be able to capture, however naturally or unnaturally, any iteration supported by imperative looping constructs. By providing the ability to define new kinds of sequences, and a facility for specifying syntactic sugar for them, we believe it should be possible to capture any iteration in a fairly natural way. Furthermore, due to the declarative semantics of the translation, all sequence quantifications have a declarative reading, providing the quantified goals and sequence definition do.

Comparison with higher order functions in functional languages is more interesting. Sequence quantification provides many of the facilities of modern functional languages. As mentioned earlier, the `,` sequence operator (`as` in the syntactically sugared version) provides a generalized version of the standard higher order `map` function. The `when`, `while`, and `once` sequence operators are closely related to Haskell's `filter`, `takewhile` and `dropwhile` functions, respectively. The `thread` sequence provides the functionality of `foldl`. However, due to the general way sequences can be composed to allow mapping over any number and any kind of sequences, in combination with folding, filtering, taking and dropping. We also benefit from Prolog's relational notation to allow any, and any number, of those sequences to be outputs. Also, since a single step of iteration is taken as the primitive building block of sequence quantification, rather than traversing a whole list, sophisticated deforestation transformations are not necessary to produce good code; a simpler local optimization is sufficient.

Note also that while Prolog does not provide any lazy evaluation, since sequences are defined to execute `sequence_next` each time the next element of the sequence is needed, sequences behave like a crude sort of lazy evaluation, computing the next element only on demand.

Of course, looping constructs have a much longer history in the imperative programming literature than in logic or functional programming. The oldest generalized approach to defining looping constructs we are aware of was proposed by Liskov and Guttag [10] in the context of the CLU language. They distinguish three kinds of abstractions: procedural abstraction, data abstraction, and iteration abstraction. Iterators in CLU are rather similar to our sequence concept, except that they are procedural, while sequences are declarative. One important difference is that iterators are defined by procedures that loop, **yield**ing sequences elements as they are found. Thus using iterators appears to be a form of coroutining. This makes defining iterators easier than defining sequences, since iterators can use local data to store state, while implementors of sequences must store state explicitly. Iterators can be defined in terms of other iterators, much as we allow sequences to be defined in terms of other sequences.

More recently, iterators have gained prominence in object oriented programming, particularly due to their appearance in the C++ Standard Template Library.[15] C++ iterators are more like sequences than CLU's iterators, as they are not defined by a loop **yield**ing values, but rather by a class with methods to get the "current" value and advance to the next value. Sequences are similar, except that their primitive operations are checking for emptiness and getting the next element and remainder sequence. Since C++ has all the facilities of an imperative language, and since iterators are used by explicitly asking for the next value, it does not need to define iterators in terms of other iterators. However, that is possible, should it be desired.

References

1. Krzysztof 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 P. Hill. Extending godel for expressing restricted quantifications and arrays. Technical Report 102, UPMAIL, Uppsala University, Box 311, S-751 05, Sweden, 1995. Available from `http://citeseer.nj.nec.com/barklund95extending.html`.
3. Jonas Barklund and Johan Bevemyr. Prolog with arrays and bounded quantifications. In A. Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning (LPAR'93)*, volume 698 of *LNAI*, pages 28–39, St. Petersburg, Russia, July 1993. Springer Verlag.
4. Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

5. Gottlob Frege. *Begriffsschrift, eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens.* Halle, 1879. English translation in *From Frege to Gödel, a Source Book in Mathematical Logic* (J. van Heijenoort, Editor), Harvard University Press, Cambridge, 1967, pp. 1–82.
6. Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. The Mercury language reference manual. Available from `<http://www.cs.mu.oz.au/mercury/>`, 2000.
7. Pascal Van Hentenryck. *The OPL Optimization Programming Language.* MIT Press, 1999.
8. ISO. *Standard for the Programming Language Prolog.* ISO/IEC, 1995.
9. Robert A. Kowalski. Predicate logic as a programming language. In *Proceedings of IFIP 4*, pages 569–574, Amsterdam, 1974. North Holland.
10. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Developement.* MIT Press, Cambridge, Mass., 1986.
11. Lee Naish. Negation and quantifiers in NU-Prolog. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Lo gic Programming*, pages 624–634, London, England, July 1986.
12. J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.
13. M. Sanella. *InterLISP Reference Manual.* Xerox PARC, Palo Alto, CA, October 1983.
14. Joachim Schimpf. Logical loops. In Peter J. Stuckey, editor, *Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 224–238. Springer-Verlag, July 29–August 1 2002.
15. B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 3 edition, 1997.
16. James Thom and Justin Zobel. NU-Prolog reference manual, version 1.0. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.
17. Simon Thompson. *The Craft of Functional Programming.* Addison-Wesley, second edition, 1999.
18. Andrei Voronkov. Logic programming with bounded quantification. In Andrei Voronkov, editor, *Logic Programming—Proc. Second Russian Conf. on Logic Programming*, number 592 in Lecture Notes in Computer Science, pages 486–514. Springer-Verlag, Berlin, 1992.
19. Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
20. Barry A. Warsaw. Lockstep iteration. Python Enhancement Proposal, 2000. available from `http://python.sourceforge.net/peps/pep-0201.html`.