PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

Providing Iteration and Concurrency in Logic Programs through Bounded Quantifications

Jonas Barklund and Håkan Millroth, UPMAIL
Computing Science Dept., Uppsala University,
Box 520, S-751 20 Uppsala, Sweden
E-mail: `jonas@csd.uu.se` or `hakanm@csd.uu.se`

Abstract

Programs operating on inductively defined data structures, such as lists, are naturally defined by recursive programs. Millroth has recently shown how many such programs can be transformed or compiled to iterative programs operating on arrays. The transformed programs can be run more efficiently than the original programs, particularly on parallel computers.

The paper proposes the introduction of 'bounded quantifications' in logic programming languages. These formulas offer a natural way to express programs operating on arrays and other 'indexable' data structures. 'Bounded quantifications' are similar to 'array comprehensions' in functional languages such as Haskell. They are inherently concurrent and can be run efficiently on sequential computers as well as on various classes of parallel computers.

1 PROCESSING DATA STRUCTURES

There are two principal ways of building a data structure in a logic program.

A1. Use a recursive relation which defines explicitly the contents of a finite part of the data structure and then uses itself recursively to define the rest of the data structure.

B1. Express directly the contents of each element of the data structure, preferrably through an 'indexing' of the elements of the data structure.

Correspondingly there are two principal ways of traversing a data structure in a logic program.

A2. Use a recursive relation which examines explicitly the contents of a finite part of the data structure and then uses itself recursively to traverse the rest of the data structure.

B2. Access directly the contents of each element of the data structure, preferrably through an 'indexing' of the elements of the data structure.

(There is, of course, an obvious duality between these operations.)

Method A is often natural when one uses inductively defined data structures, including lists, trees, etc. Method B is often natural when one uses data structures whose elements can be indexed. Some data structures, most importantly lists, fall in both categories and which method is most natural depends on the context.

2 RECURSION

We can broadly classify recursive programs in 'conjunctive' and 'disjunctive' programs (some are a mixture). The former category use recursion to compute a conjunction, like the following *lessall* program.[1]

$\mathit{lessall}(A, [B|X]) \leftarrow A < B \wedge \mathit{lessall}(A, X).$
$\mathit{lessall}(A, [\,]).$

A formula $\mathit{lessall}(A, [B_1, B_2, \ldots, B_n])$ reduces to the finite conjunction

$$A < B_1 \wedge A < B_2 \wedge \cdots \wedge A < B_n$$

which could be expressed more briefly as

$$\forall i\{1 \le i \le n \rightarrow A < B_i\}.$$

This reduction can be performed at compile time, except that the value of n is the length of the list actually supplied to the program. Such a program can be run efficiently as an iteration on a sequential computer.

The latter category uses recursion to compute a disjunction, for example the member program.

$\mathit{member}(A, [B|X]) \leftarrow A = B.$
$\mathit{member}(A, [B|X]) \leftarrow \mathit{member}(A, X).$

A formula $\mathit{member}(A, [B_1, B_2, \ldots, B_n])$ reduces to the finite disjunction

$$A = B_1 \vee A = B_2 \vee \cdots \vee A = B_n$$

[1]Our language consists (initially) of clauses whose bodies may contain conjunctions, disjunctions and negations. We assume "Herbrand" equality except for arithmetic expressions and array elements. All examples can be easily translated into Prolog or Gödel (Hill & Lloyd, 1991).

which could, in turn, be expressed more briefly as

$$\exists i\{1 \leq i \leq n \wedge A = B_i\}$$

which can, similarly, be run efficiently. Millroth's compilation method (1990, 1991), based on Tärnlund's Reform inference system (1992) transforms 'conjunctive' and 'disjunctive' recursive programs to the iterative programs above.

2.1 Concurrency

The conjunction, or disjunction, in a logic program can be interpreted as a concurrent operator, as in AND-parallel and OR-parallel logic programming systems. This does not yield sufficient concurrency for running recursive programs efficiently on parallel computers. Even using a concurrent connective, work is only initiated on one 'recursion level' in each step. This implies a linear run time which can be approximated by an expression $An+B$ (where A is the overhead for each recursion level, n is the recursion depth and B is the time spent in each recursion level). The number of literals in a recursive clause is typically much smaller than the depth of the recursion. For recursive programs with simple bodies, such as *lessall* or *member*, the An term will always dominate Only for small recursion depths and complex bodies will the B term be significant.

Recursive programs transformed by Millroth's method have a much larger potential to run efficiently on parallel computers. The iterative programs can be run in parallel on n processors unless prohibited by data dependencies etc. Techniques for parallelizing this kind of iterations have been developed for, and applied to, FORTRAN programs for some time.

3 EXPLICIT QUANTIFICATION

It is possible to build arrays and other indexable data structures, or express relations over them using recursive programs. It is often more natural to use a universal or existential quantification over the members of the data structure.

We may express the *lessall* relation over arrays as

$$lessall(A, X) \leftarrow \forall B \forall I\{X[I] = B \rightarrow A < B\},$$

provided that the the value of the expression $X[I]$ is the Ith element of the array X.

We may express reversal of the elements in an array:

$$\begin{aligned}
&reverse(X_1, X_2) \leftarrow \\
&\quad size(0, X_1, L) \wedge size(0, X_2, L) \wedge \\
&\quad \forall A \forall I\{X_1[I] = A \rightarrow X_2[L-I-1] = A\}.
\end{aligned}$$

(Our notation assumes that the expression $L-I-1$ is evaluated and replaced by its value. We also assume that array indices are zero based. Finally, we let $size(D, X, S)$ express that the size of the array X in dimension D is S.)

We may express one generation of Conway's game of Life:

$$\begin{aligned}
&step(G_1, G_2) \leftarrow \\
&\quad size(0, G_1, S_0) \wedge size(0, Q, S_0) \wedge size(0, G_2, S_0) \wedge \\
&\quad size(1, G_1, S_1) \wedge size(1, Q, S_1) \wedge size(1, G_2, S_1) \wedge \\
&\quad \forall I \forall J\{Q[I, J] = G_1[I-1 \bmod S_0, J-1 \bmod S_1] + \\
&\qquad G_1[I-1 \bmod S_0, J] + \\
&\qquad G_1[I-1 \bmod S_0, J+1 \bmod S_1] + \\
&\qquad G_1[I, J-1 \bmod S_1] + \\
&\qquad G_1[I, J+1 \bmod S_1] + \\
&\qquad G_1[I+1 \bmod S_0, J-1 \bmod S_1] + \\
&\qquad G_1[I+1 \bmod S_0, J] + \\
&\qquad G_1[I+1 \bmod S_0, J+1 \bmod S_1] \rightarrow \\
&\quad (Q[I, J] < 2 \wedge G_2[I, J] = 0 \vee \\
&\quad Q[I, J] = 2 \wedge G_2[I, J] = 1 \vee \\
&\quad Q[I, J] = 3 \wedge G_2[I, J] = 1 \vee \\
&\quad Q[I, J] > 3 \wedge G_2[I, J] = 0)\}.
\end{aligned}$$

We can also present a simple example of the use of explicit existential quantifiers. The problem is to find the position I in a array X of some element which is smaller than a given value A.

$$small(I, X, A) \leftarrow \exists J\{X[J] = B \rightarrow B < A \wedge J = I\}.$$

In all these examples we have quantified over the elements of an indexable data structure. There are other useful relations which can be expressed naturally in this way, and run efficiently. Specifically we want to include all quantifications over the elements of a finite set, whose members are 'obvious'. Below we will be somewhat more precise what this means.

4 BOUNDED QUANTIFICATION

Consider those universally quantified formulas which are instances of the schema

$$\forall x\{\Theta[x] \rightarrow \Phi[x]\}$$

where Θ is a formula which is "obviously" true for only a finite number of values of x, denoted by, say, $\{c_0, c_1, \ldots, c_{k-1}\}$. In this case the quantification is clearly equivalent to the finite conjunction

$$\begin{aligned}
&(\Theta[c_0] \rightarrow \Phi[c_0]) \wedge \\
&(\Theta[c_1] \rightarrow \Phi[c_1]) \wedge \cdots \wedge \\
&(\Theta[c_{k-1}] \rightarrow \Phi[c_{k-1}])
\end{aligned}$$

which is, by the definition of Θ, equivalent to

$$\Phi[c_0] \wedge \Phi[c_1] \wedge \cdots \wedge \Phi[c_{k-1}].$$

Similarly, a formula which is an instance of the schema $\exists x\{\Theta[x] \wedge \Phi[x]\}$ is under the same assumptions equivalent to

$$\Phi[c_0] \vee \Phi[c_1] \vee \cdots \vee \Phi[c_{k-1}].$$

We propose to

1. identify a set of formulas which always are true for only a finite number of objects, we call them *range formulas*,

2. make a system which recognizes those instances of the schema above where Θ is a range-formula, we call them *bounded quantifications*, and

3. interpret bounded quantifications concurrently. The conjuncts obtained from a bounded quantification may be run in any order, even simultaneously, provided that any data dapendencies (arising, e.g., from numerical expressions) are satisfied.

Since a range formula is required to hold for a finite number of objects, it is possible to enumerate them (as we have indeed done above with $\{c_0, c_1, \ldots, c_{k-1}\}$). It will become apparent from examples below that it is very useful to have range formulas relate each object with a unique integer in $\{0, 1, \ldots, k-1\}$.

In the following sections we will first identify a few useful range formulas and then show how to run bounded quantifications efficiently on sequential and parallel computers.

5 RANGE FORMULAS

The following is an incomplete set of interesting range formulas.

5.1 Array and "structure" elements

As we have seen, it is useful to quantify over all elements of a data structure. In an array, each element is associated with a unique integer in the range, say, $\{0, 1, \ldots, n\}$. We could, for example let $X[I] = E$ (where X is an array, I is a variable and E is a term) be a range formula and the *lessall* and *reverse* programs above are examples of its use. It may be difficult to write a compiler which recognizes precisely this use of an equality as a range formula. One solution would be to predefine, say, the predicate symbol *elt* by

$$elt(I, X, E) \leftarrow X[I] = E.$$

and only recognize predications on the form $elt(\cdot,\cdot,\cdot)$ as range formulas.

5.2 Integer ranges

An obviously useful range formula would be one which is true for the first k integers ($[0, k-1]$). Again, the formula $0 \leq X \wedge X < K$ expresses exactly that relation, but for practical reasons it may be wise to define the binary predication $cardinal(X, K)$ to stand for the binary relation which is true whenever $0 \leq X < K$. Note that the enumeration in this case coincides with the objects themselves.

Note, moreover, that it is trivial to obtain a range formula which is true for all integers in an arbitrary range $[I, J]$ using the binary *cardinal* predicate.

5.3 Enumerable types

A logic programming language with types is likely to contain "enumerable" types, for example, finite sets of distinct constants. One may wish to consider any predication, whose predicate symbol coincides with the name of such a type, a range relation. For example, suppose that colour is a type with the elements *spades*, *hearts*, *clubs*, and *diamonds* (in that order). Then $colour(I, X)$ is a range formula which is true if and only if I is 0 and X is *spades*, I is 1 and X is *hearts*, I is 2 and X is *clubs*, or I is 3 and X is *diamonds*.

Note that in this view an enumerable type of K elements is isomorphic with the integer range $[0, K-1]$, so it does not really add anything to the language as such.[2]

5.4 List elements and list suffixes

Lists are usually operated upon by recursively defined programs. Still, there are occasionally reasons for expressing programs through bounded quantifications. We propose two range formulas involving lists. The first associates every element of some list with its (zero-based) position in the list. The second enumerates every (not necessarily proper) suffix of some list (with the list itself being suffix 0). We propose to recognize the predication $member(I, L, X)$ as a range formula which is true if and only if X is the Ith element of the list L.

The predication $suffix(I, L, X)$ is a range formula which is true if and only if X is the Ith suffix of the list L. Note that if the length of L is K and [] denotes an empty list, then $suffix(0, L, L)$ and $suffix(K, L, [])$ are true formulas. (Since Prolog has no occur check, a programmer in that language could apply these predicates to cyclic "terms". We leave the behaviour in such a case undefined.)

5.5 Finite sets

Given that finite sets are provided as a data structure it would make sense to have range formulas for sets (e.g., membership), as has been suggested by Omodeo (personal communication). This is an interesting proposal, but is is difficult to represent arbitrary sets efficiently in a way that allows the elements to be enumerated. Multisets (bags) are easier to implement, but these are, on the other hand, quite similar to lists, except that the order in which elements occur is irrelevant.

6 SEQUENTIAL ITERATION

Consider a bounded quantification $\forall x\{\Theta[x] \rightarrow \Phi[x]\}$, such that $\Theta[x]$ is true when (and only when) the value of x is one of $\{c_0, c_1, \ldots, c_{k-1}\}$. We may run the conjuncts $\Phi[c_0] \wedge \Phi[c_1] \wedge \cdots \wedge \Phi[c_{k-1}]$ in any order, provided that any data dependencies are satisfied.

[2]They do, however, seem to make programs easier to understand and debug.

We consider now a bounded quantification without dependencies. Running it on a sequential computer is straightforward: translate the quantified formula into an iteration which evaluates, in sequence, the formulas $\Phi[c_0], \Phi[c_1], \ldots, \Phi[c_{k-1}]$.

Since the compiler knows in advance about the possible range formulas, it may generate specialized code for each kind of range formula. For example, if the range formula $\Theta[x]$ is $member(I, X, L)$ then we can illustrate the resulting code as

```
allocate_environment;
y = deref(l);
while (y != NIL)
{
    x = deref(y->head);
    code for Φ[x];
    y = deref(y->tail);
}
deallocate_environment;
```

using a C-style notation. (Note that we ignore the enumeration of the list elements in this example.) Assuming that the implementation is based on WAM (Warren, 1983) the "code for $\Phi[x]$" may introduce choice points (and thus be unable to deallocate environments) if there are alternative solutions for $\Phi[x]$.

In the important case that the proof for $\Phi[x]$ is deterministic, every pass through the loop will begin in the same environment. This is more efficient than the corresponding recursive computation in Prolog (under WAM) which will allocate and deallocate an environment for each recursive call. Most implementations will also refer to the symbol table when making the recursive call. That is somewhat less efficient than the (conditional) jump performed at the end of a loop. We predict that together these improvements will result in substantial savings, particularly when proofs are deterministic, the bodies of recursive clauses are small and recursion is deep. Meier also notes these advantages when compiling some recursive programs as iterations (1991).

7 PARALLEL ITERATION

On sequential computers bounded quantification, when at all appropriate, is likely to offer significant improvements over the corresponding recursive programs, run in the usual way. The potential speed-ups on parallel computers are still more dramatic.

Consider the conjunction

$$\Phi[c_0] \wedge \Phi[c_1] \wedge \cdots \wedge \Phi[c_{k-1}]$$

obtained from a bounded quantification $\forall x\{\Theta[x] \rightarrow \Phi[x]\}$. Since we may run the conjuncts in any order, we may also run them all in parallel (similarly for disjunctions), provided that we add synchronization to satisfy dependencies.

7.1 Running deterministic programs

There are several methods for running deterministic iterations in parallel; these ideas have successfully applied to FORTRAN programs for a long time. The following is one of the simplest. If there are k processors, numbered from 0 to $k-1$, simply let processor i evaluate $\Phi[c_i]$, for each i, $0 \leq i < k$. If there are fewer than k processors, say k' processors, simulate k processors by letting processor i evaluate $\Phi[c_j]$, for each j, $1 \leq j < k$, such that j modulo k' is i. If the computation of each $\Phi[c_i]$ is deterministic, then this is quite straightforward.

7.2 Running nondeterministic programs

Suppose that the formula Φ is such that there is a choice of two or more potential proofs for some conjunct $\Phi[c_i]$. If no two conjuncts $\Phi[c_i]$ and $\Phi[c_j]$, $i \neq j$, share any variables, then we have independent parallelism in which backtracking is 'local' and easily implemented, cf., e.g., DeGroot (1984).

This is a special case of the more general situation in which one can compute the variable assignments satisfying each conjunct independently of each other. For example, the conjuncts may share a variable, whose value at runtime is an array, and only access distinct elements of it. In general it is not possible to verify this condition statically so some run time tests will be necessary.

Consider the other case: that the free variables of conjuncts interact in such a way that it is not possible to compute variable assignments independently for each conjunct. In that case the corresponding recursive program, if run in the usual way using depth-first search of the proof tree, has to perform deep backtracking to earlier recursion levels. When investigating this class of programs we have noted that they occur surprisingly infrequently. Running such programs often leads to a combinatorial explosion of potential proofs which is only feasible when backtracking over a few recursion levels. The programs also do not behave nicely when running on, e.g., WAM. They tend to consume stack space rapidly if choice information prevents environments from being deallocated.

The problem of simultaneously finding variable assignments for a set of non-independent and non-deterministic conjuncts is also very difficult. Earlier research on backtracking in AND-parallel logic programming systems by, e.g., Conery (1987) confirms this claim.

Our current position is therefore to refuse to run in parallel any bounded quantification for which we cannot show statically, or at least with simple run time tests, that the conjuncts are independent. In the context of AND-parallel logic programming systems, DeGroot among others have investigated appropriate run time tests for independency. Note that the overhead for such tests is lower in our context. One test (say, for determining whether a free variable in a bounded quantification is instantiated at run time) is sufficient for starting arbitrarily many independent computations.

By applying these requirements also when running bounded quantifications on sequential processors it is guaranteed that the stack size when starting the proof of each conjunct will be constant.

8 SIMD AND MIMD PARALLEL COMPUTERS

We believe that bounded quantifications will run efficiently on both SIMD and MIMD parallel computers. When the bodies of bounded quantifications are simple and no backtracking is needed inside them, the capabilities of SIMD parallel computers are sufficient. It seems that most programs belong, or can be made to belong, to this class.

For those programs which do more complicated processing in the bodies of bounded quantifications, e.g., backtracking, not all processors of a SIMD parallel computer will be active simultaneously. This will reduce the efficiency of such a computer, while it may still be possible to fully utilize a MIMD parallel computer.

9 OTHER OPERATIONS

We think it is also beneficial to predefine certain useful operations, such as reductions and 'scans' over lists and arrays. Such operations will make it easy to eliminate many parallelization problems with variables shared between conjuncts in bounded universal quantifications.

For example, this is a program which computes the inner product S of two arrays X and Y.

$$\begin{aligned}
&\mathit{i\_p}(X,Y,S) \leftarrow \\
&\quad \mathit{size}(0,X,Z) \land \mathit{size}(0,Y,Z) \land \mathit{size}(0,T,Z) \land \\
&\quad \forall I \forall Q\{Y[I]=Q \rightarrow T[I]=X[I]\times Q\} \land \\
&\quad \mathit{reduce}(+,T,S).
\end{aligned}$$

The arrays X, Y and T are shared between all conjuncts but they all access distinct elements of the arrays. (The variable Q was only introduced to maintain the standard form of bounded quantifications. It seems convenient and possible to relax the syntax to recognize expressions such as $\forall I\{T[I]=X[I]\times Y[I]\}$ as bounded quantifications, which is certainly even more elegant.

Sometimes the partial sums are also needed in the computation. In this case it is useful to compute a 'scan' with plus over an array. The result is an array of the same length but where each element contains the sum of all preceding elements in the first array.

10 FURTHER EXAMPLES

We now turn to a few more examples written using bounded quantifications. In the authors' opinion these formulas express at a high level the essentials of the algorithms they implement. In some cases they contain formulas reminiscent of what would be (informally expressed) loop invariants when programming in another language.

10.1 Factorial

The following program computes the factorial of N. The program shows the use of the *cardinal* range formula.

$$\begin{aligned}
&\mathit{factorial}(N,F) \leftarrow \\
&\quad \mathit{size}(0,T,N) \land \\
&\quad \forall I\{\mathit{cardinal}(I,N) \rightarrow T[I]=I+1\} \land \\
&\quad \mathit{reduce}(\times,T,F).
\end{aligned}$$

10.2 Fibonacci

The following program computes the Nth Fibonacci number. The program is remarkable in being both simple and efficient, since it does not recompute any Fibonacci numbers. Similar effects have been accomplished using 'memo' relations and 'bottom-up' resolution, etc., but this solution appears both simple, elegant and semantically impeccable.

$$\begin{aligned}
&\mathit{fibonacci}(N,F) \leftarrow \\
&\quad \mathit{size}(0,T,N+1) \land \\
&\quad \forall I\{\mathit{cardinal}(I,N) \rightarrow \\
&\qquad\quad I=0 \land T[I]=1 \lor \\
&\qquad\quad I=1 \land T[I]=1 \lor \\
&\qquad\quad I>1 \land T[I]=T[I-1]+T[I-2]\} \land \\
&\quad F=T[N-1].
\end{aligned}$$

10.3 Finding roots in oriented forests

Suppose that the array P represents an oriented tree.[3] Each element of P contains the index of the parent of some node; roots contain their own index. The following program returns a new array in which each element points immediately to the root of its forest. This is an example of a parallel-prefix algorithm and it also illustrates how bounded quantifications and recursion can be used together.

$$\begin{aligned}
&\mathit{find}(P,P) \leftarrow \forall I\{P[I]=P[I] \rightarrow P[I]=P[P[I]]\}. \\
&\mathit{find}(P_0,P) \leftarrow \\
&\quad \forall I \forall J\{P_0[I]=J \rightarrow (J=P_0[J] \land P_1[I]=J \lor \\
&\qquad\qquad\qquad\qquad J \neq P_0[J] \land P_1[I]=P_0[J])\} \land \\
&\quad \mathit{find}(P_1,P).
\end{aligned}$$

10.4 Matrix transposition

The following little program transposes a matrix.

$$\begin{aligned}
&\mathit{trans}(M_1,M_2) \leftarrow \\
&\quad \mathit{size}(0,M_1,A) \land \mathit{size}(1,M_1,B) \land \\
&\quad \mathit{size}(0,M_2,B) \land \mathit{size}(1,M_2,A) \land \\
&\quad \forall I \forall J \forall Q\{M_1[I,J]=Q \rightarrow M_2[J,I]=Q\}.
\end{aligned}$$

[3]Recall that an oriented tree is a "directed graph with a specified node R such that: each node $N \neq R$ is the initial node of exactly on arc; R is the initial node of no arc; R is a root in the sense that for each node $N \neq R$ there is an oriented path from N to R" (Knuth, 1968).

10.5 Numerical integration

The following program computes an approximation to the integral

$$\int_a^b f(x)\,dx$$

using Simpson's method (a quadrature method). In the program below we let A and B be the limits, N the number of intervals and I the resulting approximation of the integral. We assume that the relation $r(X,Y)$ holds if and only if $f(X) = Y$, where f is the function being integrated.

$$\begin{aligned}
&\mathit{intsimp}(A,B,N,I) \leftarrow \\
&\quad W = (B-A)/N \land \\
&\quad \mathit{size}(0,G,2\times N+1) \land \mathit{size}(0,Z,N) \land \\
&\quad \forall I \forall Y\{G[I] = Y \rightarrow r(A + I \times W/2, Y)\} \land \\
&\quad \forall I \forall S\{Z[I] = S \rightarrow \\
&\qquad\qquad S = W \times (G[2\times I] + \\
&\qquad\qquad\qquad 4 \times G[2\times I+1] + \\
&\qquad\qquad\qquad G[2\times I+2])/6\} \land \\
&\quad \mathit{reduce}(+,Z,I).
\end{aligned}$$

The array G is set up to contain the $2\times N+1$ values $f(a)$, $f(a+w/2)$, $f(a+w)$, …, $f(b-w)$, $f(b-w/2)$, $f(b)$. These values are used to compute the area for each of the intervals, stored in Z. Finally the sum of the areas is computed.

10.6 Linear regression

This is an example of a more involved numeric computation, adopted from Press *et al.* (1989). The problem is to fit a set of n data points (x_i, y_i), $0 \le i < n$, to a straight line defined by the equation $y = A + Bx$. We assume that the uncertainty σ_i associated with each item y_i is known, and that all x_i (values of the dependent variable) are known exactly.

Let us first define the following sums.

$$S = \sum_{i=0}^{n-1} \frac{1}{\sigma_i^2} \qquad S_x = \sum_{i=0}^{n-1} \frac{x_i}{\sigma_i^2} \qquad S_y = \sum_{i=0}^{n-1} \frac{y_i}{\sigma_i^2}$$
$$S_{xx} = \sum_{i=0}^{n-1} \frac{x_i x_i}{\sigma_i^2} \qquad S_{xy} = \sum_{i=0}^{n-1} \frac{x_i y_i}{\sigma_i^2}$$

The coefficients A and B of the equation above can now be computed as

$$\begin{aligned}
\Delta &= SS_{xx} - S_xS_x \\
A &= \frac{S_{xx}S_y - S_xS_{xy}}{\Delta} \\
B &= \frac{SS_{xy} - S_xS_y}{\Delta}
\end{aligned}$$

The following program computes A and B from three arrays X, Y and U.

$$\begin{aligned}
&\mathit{linear\_regression}(X,Y,U,A,B) \leftarrow \\
&\quad \mathit{size}(0,X,N) \land \mathit{size}(0,Y,N) \land \mathit{size}(0,U,N) \land \\
&\quad \mathit{size}(0,Z,N) \land \mathit{size}(0,Z_x,N) \land \mathit{size}(0,Z_y,N) \land \\
&\quad \mathit{size}(0,Z_{xx},N) \land \mathit{size}(0,Z_{xy},N) \land \\
&\quad \forall I\{\mathit{cardinal}(I,N) \rightarrow \\
&\qquad\qquad Z[I] = 1/(U[I] \times U[I]) \land \\
&\qquad\qquad Z_x[I] = X[I]/(U[I] \times U[I]) \land \\
&\qquad\qquad Z_y[I] = Y[I]/(U[I] \times U[I]) \land \\
&\qquad\qquad Z_{xx}[I] = (X[I] \times X[I])/(U[I] \times U[I]) \land \\
&\qquad\qquad Z_{xy}[I] = (X[I] \times Y[I])/(U[I] \times U[I])\} \land \\
&\quad \mathit{reduce}(+,Z,S) \land \\
&\quad \mathit{reduce}(+,Z_x,S_x) \land \mathit{reduce}(+,Z_y,S_y) \land \\
&\quad \mathit{reduce}(+,Z_{xx},S_{xx}) \land \mathit{reduce}(+,Z_{xy},S_{xy}) \land \\
&\quad \mathit{Delta} = S \times S_{xx} - S_x \times S_x \land \\
&\quad A = (S_{xx} \times S_y - S_x \times S_{xy})/\mathit{Delta} \land \\
&\quad B = (S \times S_{xy} - S_x \times S_y)/\mathit{Delta}.
\end{aligned}$$

It is obvious that this program can be run in $O(\log n)$ time using n processors, dominated by the reductions. The bounded quantification which computes the intermediate arrays Z, Z_x, Z_y, Z_{xx} and Z_{xy} runs in constant time using n processors.

11 LIST EXAMPLES

The following two examples are present simply to show that it is possible to express also list algorithms using bounded quantifications, although the recursive programs are usually more elegant.

11.1 Lessall

The *lessall* program for lists is of course very similar to the array program (this makes it easy to change the data structure).

$$\mathit{lessall}(A,L) \leftarrow \forall B \forall I\{\mathit{member}(I,L,B) \rightarrow A < B\}.$$

11.2 Partition

The *partition* program, finally, is an example of a program which is much clearer when expressed recursively. We intend that $\mathit{partition}(X,A,L,H)$ be true if and only if L contains exactly those of elements of X which are less than or equal to A, and H contains exactly those which are greater than A. The partition predicate is usually part of an implementation of Hoare's Quicksort algorithm. Here is the recursive program:

$$\begin{aligned}
&\mathit{partition}([\,],A,[\,],[\,]). \\
&\mathit{partition}([B|X],A,L,[B|H]) \leftarrow \\
&\quad A \le B \land \mathit{partition}(X,A,L,H). \\
&\mathit{partition}([B|X],A,[B|L],H) \leftarrow \\
&\quad A > B \land \mathit{partition}(X,A,L,H).
\end{aligned}$$

In the following program which uses bounded quantifications, we have tried to keep some of the structure of the recursive program.

$$\begin{aligned}
&\mathit{partition}(X,A,L,H) \leftarrow \\
&\quad \forall F_X \forall Z \forall I\{\mathit{suffix}(I,X,F_X) \rightarrow \\
&\qquad\qquad \mathit{member}(I,S_L,L) \land \\
&\qquad\qquad \mathit{member}(I,S_H,H) \land \\
&\qquad\qquad \mathit{part}(F_X,L,H,A,S_L,S_H)\} \land
\end{aligned}$$

```
    $member(1, S_L, L) \wedge member(1, S_H, H)$.
$part([], [], [], A, S_L, S_H)$.
$part([B|X], L, H, A, S_L, S_H) \leftarrow$
    $J = I + 1 \wedge$
    $member(J, S_L, L_1) \wedge member(J, S_H, H_1) \wedge$
    $(A \leq B \wedge L = L_1 \wedge H = [B|H_1] \vee$
    $A > B \wedge L = [B|L_1] \wedge H = H_1)$.
```

The program computes two lists of lists S_L and S_H which are scans of partitions on X, picking out those elements which are less than or greater than A, respectively.

12 NESTED BOUNDED QUANTIFICATIONS

Consider a bounded quantification whose body is another bounded quantification:

$$\forall x\{\Theta_1[x] \to \forall y\{\Theta_2[y] \to \Phi[x, y]\}\}.$$

Provided that $\Theta_1[x]$ is true for any x in $\{c_0, c_1, \ldots, c_{k-1}\}$, and that similarly $\Theta_2[y]$ is true for any y in $\{d_0, d_1, \ldots, d_{\ell-1}\}$, the nested bounded quantification is equivalent to the $k \times \ell$ element conjunction

$$\begin{array}{llll} \Phi[c_0, d_0] & \wedge\, \Phi[c_0, d_1] & \wedge \cdots & \wedge\, \Phi[c_0, d_{\ell-1}] \\ \wedge\, \Phi[c_1, d_0] & \wedge\, \Phi[c_1, d_1] & \wedge \cdots & \wedge\, \Phi[c_1, d_{\ell-1}] \\ \vdots & \vdots & \ddots & \vdots \\ \wedge\, \Phi[c_{k-1}, d_0] & \wedge\, \Phi[c_{k-1}, d_1] & \wedge \cdots & \wedge\, \Phi[c_{k-1}, d_{\ell-1}] \end{array}$$

As before, provided that all data dependencies are satisfied, all these conjuncts can be run simultaneously.

13 TOLERATING DEPENDENCIES

In all examples shown above the computations of the conjuncts obtained from a bounded quantification have been independent. Therefore the conjuncts could be computed in any order, for example in parallel.

There are interesting computations where the resulting conjuncts are dependent. Consider, for example, the following program (adapted from a program by Anderson & Hudak [1990]) which defines an $n \times n$ matrix A through a recurrence.

```
$rec(A) \leftarrow$
    $size(0, A, N) \wedge size(1, A, N) \wedge$
    $\forall I \forall J\{A[I, J] = X \to$
                $I = 1 \wedge X = 1 \vee$
                $I > 1 \wedge J = 1 \wedge X = 1 \vee$
                $I > 1 \wedge J > 1 \wedge$
                $X = A[I-1, J] +$
                    $A[I-1, J-1] +$
                    $A[I, J-1]\}$.
```

This program requires a co-routining implementation of bounded quantification to run on a sequential computer or synchronization to run on a parallel computer. We are currently investigating whether automatic generation of synchronization/co-routining code is sufficient or if the programmer should be allowed to annotate the program, for example, through read-only variables (Shapiro, 1983).

14 RELATED WORK

We noted above that M. Meier has suggested (1991) how to compile some tail recursive (conjunctive as well as disjunctive) programs to iterative programs on top of WAM.

Several authors, e.g., Lloyd & Topor (1984) and Sato & Tamaki (1989), have discussed methods for running logic programs with arbitrary formulas in bodies. Our method only covers a limited extension of Horn clauses.

14.1 Array Comprehensions

It is obvious that there are similarities between arrays and bounded quantifications on one side, and the array comprehensions proposed for the Haskell language (Hudak & Wadler, 1990) on the other. Both concepts aim to express the contents of an array, or the relationship between several arrays, declaratively.

It appears to us, as with most functional programming language concepts, that when they are at all appropriate they offer a more compact and occasionally more elegant notation. For example, the factorial program above could have been expressed more easily if an expression describing the temporary array T could have been written immediately.

However, when the relationship between the elements of more than one array are to be described, the bounded quantifications appear to be more comprehensive.

Array comprehensions are, in general, evaluated by lazy computation. This can be thought of as a degenerated form of concurrency which suspends part of a computation until it is known that it must be performed. We do not think lazy computation is necessary, provided unification with the "logical variable" and a more general form of concurrency.

Futures (Halstead, 1985) are yet another way of giving names for values which are yet to be fully computed.

14.2 Nova Prolog

The ideas presented above originated as a generalization of the language Nova Prolog (Barklund & Millroth, 1988).[4] Here, however, it is appropriate to present Nova Prolog as a language embodying a subset of bounded quantifications. The subset is chosen to obtain a language tailored specifically for massively parallel SIMD computers, such as the Connection Machine. More specifically, we assume that we can store some data structures in such a way that processor i has particularly efficient access to the ith element of each data structure. We say that those data structures are distributed.

[4]Nova Prolog relates to Prolog in much the same way as *LISP (by Thinking Machines Corp.) relates to Common LISP and C* (also by Thinking Machines Corp.) to C. That is, it is a sequential programming langauge extended with a distributed data structure and a control structure for expressing computations over each element on the data structure.

We currently limit the distributed data structures to be compound terms; in fact only those compund terms whose function symbol is *pterm* and whose arity is some fixed value. We shall call them 'pterms.' (This is to help a compiler distinguish distributed data structures from other compound terms.)

Since pterms are the only distributed data structures and they are compound terms, the only range formula we need is $arg(i, t, x)$.[5] We have chosen a syntax for bounded quantifications which makes it possible to combine the range formula with the quantification of variables. In Nova Prolog a formula

$$A_1 : T_1, A_2 : T_2, \ldots, A_n : T_n \,//\, \Phi[self],$$

where T is a pterm, is called a 'parall' and has the same meaning as the bounded quantification

$$\forall I \forall A_1 \forall A_2 \cdots \forall A_n (arg(I, T_1, A_1) \rightarrow arg(I, T_2, A_2) \wedge \cdots \wedge arg(I, T_n, A_n) \wedge \Phi[I]),$$

namely that Φ is true for every corresponding element A_i of T_i, $1 \leq i \leq n$. We can see that in Nova Prolog the 'index' I is implicit and is denoted by the constant symbol *self* in the body Φ.

All examples above for array computations can be translated into Nova Prolog. We have recently implemented parts of Nova Prolog in *LISP (Blanck, 1991).

15 CONCLUSION AND FUTURE WORK

We have defined bounded quantifications, a new construct for logic programming languages. We have discussed how they can be efficiently implemented on sequential and parallel computers. They offer clarity as well as efficiency and we propose that language designers and implementors consider including them in implementations of, e.g., Prolog, Gödel and KL1.

A natural continuation of this work is to verify experimentally that bounded quantifications can be implemented efficiently in sequential and concurrent languages, and on sequential and parallel computers. It is also important to investigate how data dependencies and other synchronization considerations can be handled, when bounded quantifications are interpreted concurrently.

REFERENCES

Anderson, S. & Hudak, P., 1990, Compilation of Haskell Array Comprehensions for Scientific Computing. In *Proc. SIGPLAN '90 Conf. on Programming Language Design and Implementation.* ACM Press, New York, N.Y.

Barklund, J. & Millroth, H., 1988, Nova Prolog. *UPMAIL Tech. Rep. 52.* Computing Science Dept., Uppsala University.

Blanck, J., 1991, Abstrakt maskin för Nova Prolog. Internal report. Computing Science Dept., Uppsala University.

DeGroot, D., 1984, Restricted And-Parallelism. In *Proc. Intl. Conf. on Fifth Generation Comp. Systems 1984*, pp. 471–8. North-Holland, Amsterdam.

Halstead, R., 1985, Multilisp—a Language for Concurrent Symbolic Computation. *ACM TOPLAS, 2*, 501–38.

Hill, P. M. & Lloyd, J. W., 1991, The Gödel Report. *Tech. Rep. 91-02.* Computer Science Dept., University of Bristol.

Hudak, P. & Wadler, P., 1990, Report on the Programming Language Haskell. *Tech. Rep. YALEU/DCS/RR-777.* Dept. of Computer Science, Yale Univ.

Knuth, D. E., 1968, *The Art of Computer Programming.* Volume 1 / Fundamental Algorithms. Reading, Mass.

Lloyd, J. W. & Topor, R. W., 1984, Making Prolog more Expressive. *J. Logic Programming, 1*, 225–40.

Meier, M., 1991, Recursion vs. Iteration in Prolog. In *Proc. 8th Intl. Conf. on Logic Programming* (ed. K. Furukawa), pp. 157–69. MIT Press, Cambridge, Mass.

Millroth, H., 1990, Reforming Compilation of Logic Programs. Ph.D. thesis. *Uppsala Theses in Computing Science 10.* Computing Science Dept., Uppsala University. (A summary will appear in the next item.)

Millroth, H., 1991, Reforming Compilation of Logic Programs. In *Proc. 1991 Intl. Logic Programming Symp.* (ed. V. Saraswat, K. Ueda). MIT Press, Cambridge, Mass.

Press, W. H. *et al.*, 1989, *Numerical Recipes.* The Art of Scientific Computing. Cambridge Univ. Press, Cambridge, U.K.

Sato, T. & Tamaki, H., 1989, First Order Compiler: a Deterministic Logic Program Synthesis Algorithm. *J. Symbolic Computation, 8*, 605–27.

Shapiro, E., 1983, A Subset of Concurrent Prolog and Its Interpreter. *Technical Report TR-003.* ICOT, Tokyo.

Tärnlund, S.-Å., 1992, Reform. In *Massively Parallel Reasoning Systems* (ed. J. A. Robinson). To be published by MIT Press, Cambridge, Mass.

Warren, D. H. D., 1983, An Abstract Prolog Instruction Set. *SRI Tech. Note 309.* SRI International, Menlo Park, Calif.

[5] The difference from the *elt* predicate we proposed earlier is that *arg* operates on compund terms, rather than arrays, and that indexing is one-based. This is of course related to the use of the arg predicate in Prolog.