Reforming Compilation of Logic Programs

Håkan Millroth

Computing Science Dept., Uppsala University
Box 520, S-751 20 Uppsala, Sweden
Electronic mail: hakanm@csd.uu.se

Abstract. We present a new method for parallel logic programming which is based on compilation of Tärnlund's inference system Reform. The idea is to compile recursively defined programs to parallel iterative code. Beside earlier parallel concepts, such as OR-parallelism and AND-parallelism, we have new forms of Reform parallelism: unification parallelism and recursion parallelism. These are implemented in our method by applying standard loop parallelization techniques to the iterative code obtained by compilation.

1. INTRODUCTION

Almost any logic program using a significant amount of computer time spends most of that time executing one or more recursive procedures. It is therefore unfortunate that parallel systems based on SLD-resolution cannot fully exploit the parallelism inherent in recursive programs. As an example, consider the program for checking whether a number z is smaller than each element of a list:

```
LessAll(∅,z).
LessAll(x.y,z) ← z < x ∧ LessAll(y,z).
```

Suppose we invoke this program with a goal containing a list of n elements. An AND-parallel system based on SLD-resolution will take n steps to do the n comparisons, since the n recursive calls to the program must be made in sequence. Yet this problem is parallel in nature. It should, in principle, be possible to make all comparisons simultaneously since they are not interdependent. The sequentiality is imposed by the way recursion is handled in the top-down computation schemes derived from Kowalski's (1974) procedural interpretation of logic programs.

Tärnlund (1991) has proposed a solution to this problem: the Reform inference system. The idea is to specialize the program at runtime with respect to a particular goal. For example, specialization of the recursive clause of the `LessAll` program with respect to the goal

```
← LessAll(1.2...n.∅,0)
```

adds the following clause (called the nth *reforment*) to the program:

$$\texttt{LessAll}(\texttt{x}_1 \ldots \texttt{x}_n \,.\, \texttt{y}, \texttt{z}) \leftarrow \texttt{z} < \texttt{x}_1 \wedge \cdots \wedge \texttt{z} < \texttt{x}_n \wedge \texttt{LessAll}(\texttt{y}, \texttt{z}).$$

The program now represents a parallel algorithm and the n comparisons can be made simultaneously. Furthermore, there is an additional source of parallelism in the specialized program: unification of the list $\texttt{1.2}\ldots\texttt{n.}\emptyset$ in the goal with the list $\texttt{x}_1 \ldots \texttt{x}_n \,.\, \texttt{y}$ in the clause head reduces to n independent subunifications that can proceed in parallel.

We shall take up this idea and consider the question: how do we uncover the parallel algorithm *at compile-time*? Our solution is to obtain a *parameterized* (by n, the recursion depth) representation of the specialized program. The head of a recursive clause is compiled to iterative code (WHILE- and FOR-loops) for a single large unification that replaces the n smaller head unifications of an SLD-resolution system. The body of the clause is compiled to iterative code that runs the left-body (the calls to the left of the recursive call) and right-body in separate FOR-loops.

The significance of compiling recursion to FOR-loops in this way is that the compiled code can be parallelized by standard techniques developed for parallelizing Fortran loops. This is a form of parallel processing that is very efficient, since the computation can be directly mapped to an array of parallel processors without overheads for run-time scheduling and load balancing.

Our compilation technique apply also to nonlinear recursion on, for example, binary trees (Millroth, 1990). However, we only discuss the linear case in this paper.

2. REFORM

Reform (Tärnlund, 1991) is a new inference system for logic programming that employs program transformation as computation. The idea is to specialize recursive programs at runtime with respect to a particular size of the input data. Reform can be formulated as a Resolution (Robinson, 1965) proof procedure. Due to space limitations we present Reform here by means of an example. Suppose that we invoke the `LessAll` program with a call:

$$\leftarrow \texttt{LessAll}(\texttt{1.2.3.4.}\emptyset, \texttt{0}) \qquad (1)$$

The Reform computation amounts to construction of the 4th reforment of the recursive clause with which the call is then resolved. In

the first step, two variants (2) and (3) of the recursive clause are resolved (we adapt the convention to underline the literals selected for unification):

$$\texttt{LessAll}(x_1 \,.\, y_1 \,,\, z_1) \leftarrow z_1 < x_1 \wedge \underline{\texttt{LessAll}(y_1 \,,\, z_1)}\,. \qquad (2)$$

$$\underline{\texttt{LessAll}(x_2 \,.\, y_2 \,,\, z_2)} \leftarrow z_2 < x_2 \wedge \texttt{LessAll}(y_2 \,,\, z_2)\,. \qquad (3)$$

The reforment derived is (4). In the second step, (4) and a new variant (5) of the recursive program clause are resolved:

$$\texttt{LessAll}(x_1 \,.\, x_2 \,.\, y_2 \,,\, z_2) \leftarrow z_2 < x_1 \wedge z_2 < x_2 \wedge \underline{\texttt{LessAll}(y_2 \,,\, z_2)}\,. \qquad (4)$$

$$\underline{\texttt{LessAll}(x_3 \,.\, y_3 \,,\, z_3)} \leftarrow z_3 < x_3 \wedge \texttt{LessAll}(y_3 \,,\, z_3)\,. \qquad (5)$$

The reforment derived is (6). In the third step, we resolve (6) and a new variant clause (7):

$$\texttt{LessAll}(x_1 \,.\, x_2 \,.\, x_3 \,.\, y_3 \,,\, z_3) \leftarrow \qquad (6)$$

$$\quad z_3 < x_1 \wedge z_3 < x_2 \wedge z_3 < x_3 \wedge \underline{\texttt{LessAll}(y_3 \,,\, z_3)}\,.$$

$$\underline{\texttt{LessAll}(x_4 \,.\, y_4 \,,\, z_4)} \leftarrow z_4 < x_4 \wedge \texttt{LessAll}(y_4 \,,\, z_4)\,. \qquad (7)$$

In this step the 4th reforment (8) is obtained:

$$\texttt{LessAll}(x_1 \,.\, x_2 \,.\, x_3 \,.\, x_4 \,.\, y_4 \,,\, z_4) \leftarrow \qquad (8)$$

$$\quad z_4 < x_1 \wedge z_4 < x_2 \wedge z_4 < x_3 \wedge z_4 < x_4 \wedge \texttt{LessAll}(y_4 \,,\, z_4)\,.$$

Finally, the call (1) is resolved with (8). The new goal obtained is:

$$\leftarrow 0 < 1 \wedge 0 < 2 \wedge 0 < 3 \wedge 0 < 4 \wedge \texttt{LessAll}(\emptyset, 0) \qquad (9)$$

Tärnlund (1991) gives a more efficient algorithm for Reform computation that requires only $\lceil \log n \rceil$ steps to compute the nth reforment. The merit of the linear algorithm used in our example is simplicity and uniformity—this is helpful in the analysis of Reform transformations from which we derive a parameterized representation of the nth reforment that can be obtained at compile-time.

3. REFORM PARALLELISM

Beside earlier parallel concepts, such as OR-parallelism and AND-parallelism, we have new forms of Reform parallelism: unification parallelism and recursion parallelism.

Recursion parallelism. Let us consider a recursive clause with head H and k goals $B_1, \ldots, B_k$, in addition to the recursive call T, in the body:

$$H \leftarrow B_1 \wedge \cdots \wedge B_k \wedge T.$$

The nth reforment of this clause is on the form:

$$H \leftarrow B_{11} \wedge \cdots \wedge B_{k1} \wedge \cdots \wedge B_{1n} \wedge \cdots \wedge B_{kn} \wedge T.$$

It is instructive to depict the body of this reforment as a matrix of n rows and k columns:

$$\begin{pmatrix} B_{11} & \ldots & B_{k1} \\ \vdots & \ddots & \vdots \\ B_{1n} & \ldots & B_{kn} \end{pmatrix}$$

How can the calls in this matrix be computed in parallel? First, the k calls within each row may be computed in parallel. This is traditional AND-parallelism. Second, the n rows may be computed in parallel. This generalization of AND-parallelism may be called *recursion parallelism* since it corresponds to computing all recursion levels in parallel.

It should be noticed that typically $k \ll n$. This suggests that the potential for parallel speedup is greater with recursion parallelism than with traditional AND-parallelism. Note that recursion parallelism cannot be exploited in an AND-parallel system based on SLD-resolution since the recursion levels must be computed in sequence with such a system. A restricted form, where the recursion levels overlap to some extent, can sometimes be exploited however; this may be called *recursion pipelining.*

Example. Consider the recursive clause of the `Lessall` program:

$$\texttt{Lessall(x.y,z)} \leftarrow \texttt{z < x} \wedge \texttt{Lessall(y,z)}.$$

The matrix has only one column, since there is only one call in the residual body:

$$\begin{pmatrix} \texttt{z < x}_1 \\ \texttt{z < x}_2 \\ \vdots \\ \texttt{z < x}_\texttt{n} \end{pmatrix}$$

There are, of course, no opportunities for parallelism within each row of this matrix. The n rows, however, can be computed in parallel.

Unification parallelism. A feature of logic programming is that composition and decomposition of data structures is done by unification. Hence a large share of the execution time of a clause is typically spent in head unification. Consider, for example, the `Append` program:

$$\texttt{Append}(\emptyset, \texttt{y}, \texttt{y}).$$
$$\texttt{Append}(\texttt{u}.\texttt{x}, \texttt{y}, \texttt{u}.\texttt{z}) \leftarrow \texttt{Append}(\texttt{x}, \texttt{y}, \texttt{z}).$$

There are no opportunities for AND-parallelism in this program since all productive work is done by head unifications. Let us now consider the unification parallelism of a Reform computation with this program. The nth reforment obtained by Reform is:

$$\texttt{Append}(\texttt{u}_1 \ldots \texttt{u}_n.\texttt{x}, \texttt{y}, \texttt{u}_1 \ldots \texttt{u}_n.\texttt{z}) \leftarrow \texttt{Append}(\texttt{x}, \texttt{y}, \texttt{z}).$$

Suppose that the transformed `Append` program is invoked with a call containing a list of n elements in the first argument position:

$$\leftarrow \texttt{Append}(\texttt{C}_1 \ldots \texttt{C}_n.\emptyset, \texttt{D}.\emptyset, \texttt{w})$$

Unification of the lists $\texttt{C}_1 \ldots \texttt{C}_n.\emptyset$ and $\texttt{u}_1 \ldots \texttt{u}_n.\texttt{x}$ reduces to n subunifications that might be computed in parallel. Furthermore, there is an additional source of parallelism in the construction of the output list $\texttt{u}_1 \ldots \texttt{u}_n.\texttt{z}$.

Unification parallelism, in the context of Reform computations, is a special case of recursion parallelism since it amounts to computing recursion levels in parallel. It might seem that the problem of unifying, say, two lists of n elements exhibits very little parallelism when lists are implemented as linked structures. This is not the case, however. There are algorithms that carry out such problems in $\log n$ steps for typical lists (Barklund, 1990).

4. REFORM SERIES

Program variables and variant variables (terms). A new variant of the recursive program clause is created for each step in a Reform transformation. Each variant contains a new set of variables. We shall need to make a distinction between the variables that occur in the original program clause, and the corresponding variables occurring in the

variant clauses used in the transformation. A variable occurring in a program clause C is called a *program variable*. A variable occurring in a variant C' of C is called a *variant variable*. The notions of *program term* and *variant term* are straight-forward extensions.

It is evident that n variants of a recursive clause are needed in the derivation of the nth reforment of that clause. We number these variant clauses from 1 to n according to the order in which they are used in the Reform transformation. We adapt the convention to rename a program variable v to v_i of the ith variant clause.

The n-solution of a Reform transformation. Consider a recursive clause S of the form:

$$P(q, \ldots, r) \leftarrow \Psi \wedge P(s, \ldots, t) \wedge \Phi$$

The substitutions resulting from a derivation of the nth reforment of this clause are identical to the solution of the following system E_n of equations ($E_1 = \emptyset$):

$$E_n = \left\{ \begin{array}{l} s_1 = q_2, \ldots, t_1 = r_2, \\ s_2 = q_3, \ldots, t_2 = r_3, \\ \vdots \\ s_{n-1} = q_n, \ldots, t_{n-1} = r_n \end{array} \right\}$$

The solution of E_n is a system of equations $\mu_n = \{x = u, \ldots, y = w\}$ where $x, \ldots, y$ are distinct variables not occurring in any of the terms $u, \ldots, w$. We shall refer to μ_n as the *n-solution* of the clause S.

We can associate a substitution $\sigma_n = \{u/x, \ldots, w/y\}$ with the n-solution $\mu_n = \{x = u, \ldots, y = w\}$ (we also include in σ_n a substitution z/z for each variable z that does not occur on the left-hand side of any equation in μ_n). We can therefore use $t\mu_n$, where t is any term, as a convenient notation for $t\sigma_n$.

Reform series. Consider a recursive clause S. A program variable x of S is represented by n variant variables $x_1, x_2, \ldots, x_n$ in the nth reforment of S. Let μ_n be the n-solution of S. The *Reform series* $\bar{x}$ of x is then the sequence

$$\bar{x} = \langle x_1\mu_n, \ldots, x_n\mu_n \rangle.$$

We shall often write $\bar{x}$ as $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$. The notion of Reform series is easily extended to compound terms. The Reform series of a compound term t is the sequence

$$\bar{t} = \langle t_1\mu_n, \ldots, t_n\mu_n \rangle.$$

Classification of variables. The main objective of the analysis below is to characterize the relationship between program variables and their associated Reform series for different classes of program variables. The classification of a variable depends on the substitutions in the *first* step of the Reform transformation (corresponding to the 2-solution μ_2). Since this step can be performed by the compiler, the variables in a clause can be classified at compile-time. Proofs of the theorems in this section can be found elsewhere (Millroth 1991).

NONE-variables. A variable x is a *NONE-variable* if it is not bound in the first Reform step. That is,

$$x_1\mu_2 = x_1 \quad \text{and} \quad x_2\mu_2 = x_2.$$

Examples where x is a NONE-variable:

`P(F(42,x))` $\leftarrow$ `P(y)` and `P(y)` $\leftarrow$ `Q(y,x)` $\wedge$ `R(x,z)` $\wedge$ `P(z)`.

The Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of a NONE-variable x is given by:

Theorem 4.1. *Let x be program variable in a clause S and μ_n the n-solution of S. If $x_1\mu_2 = x_1$ and $x_2\mu_2 = x_2$ then,*

$$\bar{x}_i = x_i\mu_n = x_i, \qquad \textit{for} \quad 1 \leq i \leq n.$$

For example, if $n = 4$ then $\bar{\mathtt{x}} = \langle \mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4 \rangle$.

Remark: When μ_2 contains an equation $x = y$, where both x and y are variables, we arbitrarily consider one of the variables, but not the other, bound by μ_2.

POS-variables. A variable x is a *POS-variable* if it occurs in the recursive call and there is a term t in the corresponding position of the clause head. That is,

$$x_1\mu_2 = t_2\mu_2.$$

Examples where x is a POS-variable:

`P(F(y))` $\leftarrow$ `P(x)` and `P(F(F(x)))` $\leftarrow$ `P(F(x))`.

The Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of a POS-variable x is given by:

Theorem 4.2. *Let x and t be program variable and term, respectively, in a clause S and μ_n the n-solution of S. If $x_1\mu_2 = t_2\mu_2$ then,*

$$\bar{x}_i = x_i\mu_n = \begin{cases} x_n, & i = n; \\ t_{i+1}\mu_n, & 1 \leq i < n. \end{cases}$$

For example, $\bar{\mathtt{x}} = \langle \mathtt{F(y_2)}, \mathtt{F(y_3)}, \mathtt{F(y_4)}, \mathtt{x_4} \rangle$ if $\mu_2 = \{\mathtt{x_1 = F(y_2)}\}$ and $n = 4$.

Example. Consider the recursive clause

$$\texttt{P(F(y),z)} \leftarrow \texttt{P(x,y)}.$$

The 2-solution of this clause is $\mu_2 = \{\texttt{x}_1 = \texttt{F(y}_2\texttt{)},\ \texttt{y}_1 = \texttt{z}_2\}$. We have the following classification and Reform series ($n = 4$) of the variables.

Variable	Type	Reform series
z	NONE	$\langle \texttt{z}_1, \texttt{z}_2, \texttt{z}_3, \texttt{z}_4 \rangle$
y	POS	$\langle \texttt{z}_2, \texttt{z}_3, \texttt{z}_4, \texttt{y}_4 \rangle$
x	POS	$\langle \texttt{F(z}_3\texttt{)}, \texttt{F(z}_4\texttt{)}, \texttt{F(y}_4\texttt{)}, \texttt{x}_4 \rangle$

NEG-variables. A variable x is a *NEG-variable* if it occurs in the clause head and there is a non-variable term t in the corresponding position of the recursive call. That is,

$$x_2\mu_2 = t_1\mu_2.$$

Examples where x is a NEG-variable:

$$\texttt{P(x)} \leftarrow \texttt{P(F(y))} \qquad \text{and} \qquad \texttt{P(F(x))} \leftarrow \texttt{P(F(F(x)))}.$$

The Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of a NEG-variable x is given by:

Theorem 4.3. *Let x and t be program variable and non-variable term, respectively, in a clause S and μ_n the n-solution of S. If $x_2\mu_2 = t_1\mu_2$ then,*

$$\bar{x}_i = x_i\mu_n = \begin{cases} x_1, & i = 1; \\ t_{i-1}\mu_n, & 1 < i \leq n. \end{cases}$$

For example, $\bar{\texttt{x}} = \langle \texttt{x}_1, \texttt{F(y}_1\texttt{)}, \texttt{F(y}_2\texttt{)}, \texttt{F(y}_3\texttt{)} \rangle$ if $\mu_2 = \{\texttt{x}_2 = \texttt{F(y}_1\texttt{)}\}$ and $n = 4$.

Example. Consider the recursive clause

$$\texttt{P(x,z)} \leftarrow \texttt{P(F(z),G(y))}.$$

The 2-solution of the clause is: $\mu_2 = \{\texttt{x}_2 = \texttt{F(z}_1\texttt{)},\ \texttt{z}_2 = \texttt{G(y}_1\texttt{)}\}$. We have the following classification and Reform series ($n = 4$) of the variables.

Variable	Type	Reform series
y	NONE	$\langle \texttt{y}_1, \texttt{y}_2, \texttt{y}_3, \texttt{y}_4 \rangle$
z	NEG	$\langle \texttt{z}_1, \texttt{G(y}_1\texttt{)}, \texttt{G(y}_2\texttt{)}, \texttt{G(y}_3\texttt{)} \rangle$
x	NEG	$\langle \texttt{x}_1, \texttt{F(z}_1\texttt{)}, \texttt{F(G(y}_1\texttt{))}, \texttt{F(G(y}_2\texttt{))} \rangle$

Special cases. We can identify certain special cases of POS and NEG when more direct and efficient expressions can be found for the Reform series. These special cases involve a recurrence in that a variable occurs in the term that we substitute for (a renaming of) the same variable. Let us, for example, consider the case of a POS-variable where the recursion has a 'step' function ψ:

$$P(t) \leftarrow P(x), \quad \text{where} \quad \psi(t) = x$$

We can identify a special case for each sufficiently uncomplicated function ψ. We shall only discuss two particulary simple, but common, cases here: when ψ is the 'tail' function and when it is the identity function.

INV-variables. A POS-variable x is an *INV-variable* if it occurs in corresponding positions in the head and recursive call of the clause. That is,

$$x_1\mu_2 = x_2 \quad \text{or} \quad x_2\mu_2 = x_1.$$

Examples where x is an INV-variable:

$$\texttt{P(x)} \leftarrow \texttt{P(x)} \qquad \text{and} \qquad \texttt{P(F(42,x))} \leftarrow \texttt{P(F(y,x))}.$$

The Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of an INV-variable x is given by:

Theorem 4.4. *Let x and t be program variable and term, respectively, in a clause S and μ_n the n-solution of S. If $x_1\mu_2 = x_2$ or $x_2\mu_2 = x_1$ then,*

$$\bar{x}_i = x_i\mu_n = x_n, \qquad \textit{for} \quad 1 \leq i \leq n.$$

For example, if $\mu_2 = \{\texttt{x}_1 = \texttt{x}_2\}$ and $n = 4$ then $\bar{\texttt{x}} = \langle \texttt{x}_4\,,\ \texttt{x}_4\,,\ \texttt{x}_4\,,\ \texttt{x}_4 \rangle$.

POS-LIST-variables. Let u be any term. A POS-variable x is a *POS-LIST-variable* if it occurs in the recursive call and there is a list $u.x$ in the corresponding position of the clause head. That is,

$$x_1\mu_2 = (u_2.x_2)\mu_2.$$

Examples where x is a POS-LIST-variable:

$$\texttt{P(u.x)} \leftarrow \texttt{P(x)} \qquad \text{and} \qquad \texttt{P(v.u.x)} \leftarrow \texttt{P(w.x)}.$$

The Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of a POS-LIST-variable x is given by:

Theorem 4.5. *Let x and u be program variable and term, respectively, in a clause S and μ_n the n-solution of S. If $x_1\mu_2 = (u_2.x_2)\mu_2$ then,*

$$\bar{x}_i = x_i\mu_n = \begin{cases} x_n, & i = n; \\ (u_{i+1}...u_n.x_n)\mu_n, & 1 \leq i < n. \end{cases}$$

For example, $\bar{\mathtt{x}} = \langle \mathtt{u_2 \,.\, u_3 \,.\, u_4 \,.\, x_4}\,,\ \mathtt{u_3 \,.\, u_4 \,.\, x_4}\,,\ \mathtt{u_4 \,.\, x_4}\,,\ \mathtt{x_4} \rangle$ if $n = 4$ and $\mu_2 = \{\mathtt{x_1 = u_2 \,.\, x_2}\}$.

Remarks. We conclude this section with a few remarks concerning the classification of variables.

1. A consequence of Theorems 4.1–5 is that we can compute the Reform series of any variable directly, without Reform transformation of the clause. Hence we need not carry out the Reform transformation explicitly in order to construct the transformed program. This observation forms the basis of our compilation method.
2. A variable may belong to more than one class. Suppose, for example, that a variable x belongs to both POS and NEG. We have then two expressions for the Reforms series $\bar{x}$ of x: $\bar{x}_+$ and $\bar{x}_-$, say. The fact that x belongs to both POS and NEG implies that neither $\bar{x}_+$ nor $\bar{x}_-$ gives a complete description of $\bar{x}$. The complete description is obtained by unifying $\bar{x}_+$ and $\bar{x}_-$.
3. The classification procedure can readily be extended to linear integer recursion. We may then consider the case $\psi(y) = y \oplus k$, for some integer function $\oplus$ (addition, multiplication, etc.) and integer k:
$$P(y) \leftarrow x = y \oplus k \wedge P(x);$$
$$P(x) \leftarrow x = y \oplus k \wedge P(y).$$

5. COMPILATION

The computation of a recursive program is made up of two kinds of activities: head unification and body call computation. Let H denote an activity of the former kind and B an activity of the latter kind. We can then describe, in an informal way, the structure of the computation using SLD-resolution as:

$$H\ B\ H\ B\ H\ B\ \ldots$$

That is, head unification and body call computation proceed alternatingly. We shall, in contrast, let the program compute according to the following structure:

$$H^*\ B^*$$

where H^* and B^* indicate iteration of the activities H and B.

H^* corresponds to the unification resulting from invoking the nth reforment. (This is equivalent to iteration of the unification work H performed at each recursion level in an SLD-resolution system.) We divide H^* into two phases. In the first phase we take in all input data from the invoking call. This is done by a modified version of Warren's (1977, 1983) compilation schemes for head unification. In the second phase we compute the variable bindings made in Reform transformation. This is done by code for computing Reform series according to Theorems 4.1–5.

B^* corresponds to computation of the body of the nth reforment. (This is equivalent to iteration of the clause body computation B performed at each recursion level in an SLD-resolution system.) B^* is divided into three phases: iteration of the left-body calls, a single recursive call to the program, and iteration of the right-body calls.

We arrive at H^* and B^* by compiling the recursive clause of a program to a parameterized encoding of its nth reforment. We can describe the computation of a recursive clause $H \leftarrow \Psi \wedge T \wedge \Phi$ and an invoking call G as follows.

1. Unify the head H with the call G, using the 1st element of the Reform series of each variable in H.
2. Compute Ψ n times, using the ith element of the Reform series of each variable in Ψ in the ith computation of Ψ $(1 \leq i \leq n)$.
3. Compute the recursive call T, using the nth element of the Reform series of each variable in T.
4. Compute Φ n times, using the ith element of the Reform series of each variable in Φ in the ith computation of Φ $(1 \leq i \leq n)$.

Let $H(i)$, $\Psi(i)$, $T(i)$, and $\Phi(i)$ denote H, Ψ, T, and Φ, respectively, with all variables replaced by the ith elements of the corresponding Reform series. We can then, finally, give the parameterized encoding of the nth reforment schematically as:

```
unify H(1) with G
for i=1 to n do call Ψ(i)
call T(n)
for i=n to 1 do call Φ(i)
```

We turn now to the question of how to represent variables in the compiled program. The idea is that a variable is represented at run-time by its Reform series implemented as a vector. On a parallel

machine we assume that the ith element of the Reform series of a variable is stored on processor i, for $1 \leq i \leq n$. However, we need not always represent all elements of a variable's Reform series explicitly. Suppose, for example, that the Reform series of a POS-variable x is $\langle \bar{t}_2, \ldots, \bar{t}_n, x_n \rangle$, where $\bar{t}_i$ is the ith element of the Reform series of some term t. We need not explicitly represent the first $n-1$ elements of the Reform series of x. References to these elements can be replaced by references to $\bar{t}_2, \ldots, \bar{t}_n$.

Now let us consider a simple example, the recursive clause of the `LessAll` program:

$$\texttt{LessAll(x.y,z)} \leftarrow \texttt{z < x} \wedge \texttt{LessAll(y,z)}.$$

The variable classification and representation is:

Variable	Variable class	Representation
`x`	NONE	vector
`y`	POS-LIST	scalar
`z`	INV	scalar

The compiled clause is:

```
<n,x,y> := traverse(arg1);
z := arg2;
for i=1 to n do z<x[i];
call lessall(y,z);
```

A few remarks might help the reader understand this code. The operation `traverse` traverses the input list while collecting (in `x`) and counting (in `n`) the list elements. The last tail of the input list is assigned to `y`. In the FOR-loop, all comparisons can be made simultaneously. The single recursive call to the base case is needed to ensure correct behaviour if the input list ends with an unbound variable, rather than the empty list; in this case both the base clause and the recursive clause of `Lessall` applies. In Prolog, this call would then match the base clause and create a choice point. For integer programs, where this situation does not occur since integers are ground, we can unfold the recursive call.

Note that this compilation technique is not restricted to tail-recursive programs. The correspondance between tail-recursion and unbounded iteration (that is, WHILE-loops) is trivial. The correspondance between structural recursion and bounded iteration, which we

exploit here, is less direct. Our next example program, which is not tail-recursive, uses list concatenation to reverse a list.

$$\texttt{Nrev(u.x, y)} \leftarrow \texttt{Nrev(x, z)} \wedge \texttt{Append(z, u.}\emptyset\texttt{, y)}.$$

The variable classification and representation is:

Variable	Variable class	Representation
u	NONE	vector
x	POS-LIST	scalar
y	NONE	vector
z	POS	scalar

The compiled code for this clause is:

```
<n,u,x> := traverse(arg1);
y := arg2;
for i=2 to n do y[i] := undef;
z := undef;
call nrev(x,z);
call append(z,cons(u[n],nil),y[n]);
for i=n-1 to 1 do
   call append(y[i+1],cons(u[i],nil),y[i]);
```

Our compilation method is applicable for *structural* recursion: the 'step' function (how the recursion argument is decomposed or composed) of the recursion must be derivable from the program text and not dependent on input data. Moreover, this step function must be the same in all recursive clauses of the predicate.

6. LOOP PARALLELIZATION

A consequence of compiling structural recursion to FOR-loops as discussed above is that standard techniques developed for parallelizing Fortran loops can be applied to recursive logic programs. In this section we briefly review these techniques.

A loop of n iterations can be computed in parallel on n processors by letting each processor perform one iteration of the loop. The loop can then get a n-fold speedup if the different iterations are independent (the loop program is assumed available on each processor). This technique, which has been successful for running Fortran-like programs in parallel, is known as *loop parallelization* (Kuck *et al.*, 1972).

Loop parallelization works only for bounded iteration (that is, FOR-loops) since the number of iterations to be performed must be known when the loop is entered. Hence unbounded iteration (WHILE-loops) cannot be parallelized in this way.

Consider a loop of n iterations computed in parallel on k processors. If $n = k$ then each processor is assigned one iteration of the loop. If $n > k$ then every kth iteration is assigned to the same processor. For example, the loop

```
for i=1 to 5 do
   a[i] := b[i]/j
   d[i] := b[i]*2
end
```

can be computed as follows on three processors:

Time	Processor 1:	Processor 2:	Processor 3:
↓	`a[1] := b[1]/j`	`a[2] := b[2]/j`	
	`d[1] := b[1]*2`	`d[2] := b[2]*2`	`a[3] := b[3]/j`
	`a[4] := b[4]/j`	`a[5] := b[5]/j`	`d[3] := b[3]*2`
	`d[4] := b[4]*2`	`d[5] := b[5]*2`	

Other scheduling schemes are certainly conceivable.

7. CONCLUSION

We have described a parallelizing compilation method, based on the Reform inference system, for recursive logic programs. The idea is to compile structural recursion to parallel iteration. This approach has two important advantages:

1. There is a natural mapping of the program onto a parallel machine whose processors are organized in a linear array or in a ring. The inter-processor communication on such a machine will mostly be between neighboring processors, since it is unusual that data is passed between nonadjacent recursion levels in a logic program. Furthermore, the compiler, rather than the run-time system, can perform task scheduling and load balancing.
2. It gives a natural partitioning of the computation and its data. In many logic programs, only a fraction of the data used on each recursion level is accessible from other levels. Moreover, nondeterminism and data dependencies are often local to the individual recursion levels.

ACKNOWLEDGEMENT

Thanks to Sten-Åke Tärnlund, Jonas Barklund and Åke Hansson for valuable help and advice, and to Thomas Lindgren, Margus Veanes and Johan Bevemyr for interesting discussions.

REFERENCES

BARKLUND, J. (1990) *Parallel Unification*, Ph.D. Thesis, Computing Science Dept., Uppsala University.

KUCK, D. J., Y. MURAOKA & S. CHEN. (1972) On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans. Computers* C-21, no. 12, 1293–1310.

KOWALSKI, R. A. (1974) Predicate logic as a computer language. In *Information Processing 74*, pp. 569–574. North-Holland, Amsterdam.

MILLROTH, H. (1990) *Reforming Compilation of Logic Programs*, Ph.D. Thesis, Computing Science Dept., Uppsala University.

MILLROTH, H. (1991) Compiling Reform, (to appear in) *Massively Parallel Reasoning Systems* (eds. J. A. Robinson & E. E. Sibert), MIT Press.

ROBINSON, J. A. (1965) A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12, 23–41.

TÄRNLUND, S.-Å. (1991) Reform, (to appear in) *Massively Parallel Reasoning Systems* (eds. J. A. Robinson & E. Sibert), MIT Press.

WARREN, D. H. D. (1977) Implementing Prolog—compiling predicate logic programs. Research Reports 39 and 40, Dept. of AI, Univ. of Edinburgh, Edinburgh

WARREN, D. H. D. (1983) An abstract Prolog instruction set. Report 309, SRI International, Menlo Park, Calif.