SLDR-Resolution: Parallelizing Structural Recursion in Logic Programs *

Håkan Millroth
Uppsala University

November 25, 1994

Abstract

We introduce a new operational semantics, SLDR-resolution, for a class of recursive logic programs. We establish the soundness and completeness of SLDR-resolution by showing that one SLDR-resolution inference is equivalent to \( n \geq 1 \) SLD-resolution inferences.

SLDR-resolution facilitates parallel processing of recursive programs and can exploit parallelism that is not exploitable in SLD-resolution. In contrast to SLD-resolution, where each recursive invocation of a program typically results in a small unification, SLDR-resolution results in large unification computations amenable to optimization and parallelization. Moreover, SLDR-resolution allows a form of AND-parallelism in which the recursive invocations of a procedure start computing simultaneously.

Unification in SLDR-resolution can be compiled in a novel way: the unifications correspond to solving systems of recurrence equations, and the compilation corresponds to obtaining closed-form expressions for these systems. Another new idea in compilation of SLDR-resolution is to exploit the technique of loop parallelization which was developed in the context of imperative programming in the early 1970s.

---

*To appear in *Journal of Logic Programming.*
1 Introduction

Unrolling a loop in an imperative program reduces the overheads of running the loop. A similar reduction of overheads can be achieved by unfolding the recursion in a logic program. As an example, consider the program for checking whether a number \( B \) is smaller than each element of a list:

\[
\text{lessall}([],_). \\
\text{lessall}([A|X],B) :- A > B, \text{lessall}(X,B).
\]

Suppose we invoke the program with the following call:

\[
| ?- \text{lessall}([1,2,\ldots,1000],0).
\]

A complete unfolding of the recursion in the program to match this call gives:

\[
\text{lessall}([A1,A2,\ldots,A1000],B) :- \\
A1 > B, \\
A2 > B, \\
\ldots \\
A1000 > B.
\]

This program is clearly much more efficient than the original program, since it contains no procedure calls. But we can also see that unfolding the recursion has an additional advantage: it makes the program more parallel. The 1000 comparisons can be done in parallel, given enough processors. This is a significant improvement, since, in the original program, the last comparison cannot be made until the program has been recursively invoked 1000 times.

We can rewrite the unfolded program as follows, using a low-level notation.

\[
n \leftarrow \text{number_of_elements}(Arg1); \\
\text{unify } Arg1 \text{ with } [A_1, A_2, \ldots, A_n]; \\
\text{unify } Arg2 \text{ with } B; \\
\text{for } i \leftarrow 1 \text{ to } n \text{ do } A_i > B;
\]

The important point to notice here is that the comparisons are done in a for-loop. It is possible to run all iterations of a for-loop simultaneously on a
parallel computer, a fact that parallelizing Fortran compilers have exploited since the early 1970s [10]. Hence this iterative program is as parallel as the unfolded one.

We present in this paper a compilation technique for compiling a special case of recursion, where the recursion bound can easily be computed, to for-loops. The main technical problem involved is to provide the input data to the loops. As indicated in the lessall example, this is done by carrying out all head unifications before entering the loop.

Our compilation technique does not result in SLD-resolution computations. Therefore, we define a new operational semantics, SLDR-resolution, that captures the control-flow of compiled programs. We start by identifying the programs for which SLDR-resolution is applicable: those defined by structural recursion. Informally, a recursive definition is structural if the recursion bound depends only on the size (according to some suitable norm) of the input arguments. We then define SLDR-resolution for linear recursion. We establish the soundness and completeness of SLDR-resolution by showing that one SLDR-resolution inference is equivalent to $n \geq 1$ SLD-resolution inferences.

Next we discuss how SLDR-resolution aids parallel processing of recursive programs. SLDR-resolution can exploit parallelism that is not exploitable in SLD-resolution. First, it can exploit fine-grained parallelism in the big head unification. Second, the recursive invocations of the program can be computed in parallel.

We then show how a large share of the unification computation of SLDR-resolution can be precompiled. These unification computations correspond to solving systems of recurrence equations, and the precompilation corresponds to obtaining closed-form expressions for these systems. Different (syntactically distinguishable) types of variables have different types of closed forms. In the last part of the paper we discuss compilation of SLDR-resolution.
2 Preliminary definitions

2.1 Language

Our language is that of first-order predicate logic with disjoint sets of variables, function symbols and predicate symbols. Each function symbol and predicate symbol has an associated arity (a natural number).

A term is either a variable, a function symbol with arity 0 (a constant), or a function symbol with arity \( k > 0 \) applied to \( k \) terms.

An atom is either a predicate symbol with arity 0, or a predicate symbol with arity \( k > 0 \) applied to \( k \) terms. A literal is either an atom (a positive literal) or the negation of an atom (a negative literal).

A clause is a finite set of literals. A program clause is a clause with exactly one positive literal. A goal is a clause with only negative literals (calls). The empty clause (goal) is denoted by \( \Diamond \). A predicate (procedure) is a collection of program clauses whose positive literals all have the same predicate symbol and arity.

Two expressions (terms, literals, clauses) that are equal up to variable renaming are called variants of each other. We indicate variants by subscripts. For example, if \( C \) is a clause then \( C_2 \) and \( C_7 \) are variants of \( C \) (and of each other).

2.2 Notation for logic programs

We use a meta language to discuss programs, and an object language to define programs. In the meta language, expressions (terms, literals, clauses) are written using italics and roman fonts. We use the notation

\[ H \leftarrow B_1 \land \cdots \land B_m \]  
\[ \leftarrow B_1 \land \cdots \land B_m \]

for program clauses and goals, respectively. Upper-case greek letters (\( \Psi, \Phi \), etc.) denote conjunctions of literals.

In the object language, expressions are written in type-writer font. Standard Prolog syntax is used.
2.3 Equations and substitutions

A set of equations \( \{ x_1 = t_1, \ldots, x_n = t_n \} \) is on solved form if \( x_1, \ldots, x_n \) are distinct variables not occurring in any of the terms \( t_1, \ldots, t_n \). The solution of a set of equations \( E \), denoted by \( \text{soln}(E) \), with respect to an equality theory \( T \), is a set of equations on solved form such that

\[
T \models E \quad \text{if and only if} \quad T \models \text{soln}(E).
\]

We assume that \( \text{false} \in E \) if and only if \( E \) has no solution. We fix \( T \) to be Clark's syntactic equality theory [6].

Let \( \mu = \{ x_1 = t_1, \ldots, x_n = t_n \} \) be the solution of a set of equations and let \( w \) be a term. The application of \( \mu \) to \( w \), written \( w\mu \), is the term obtained by replacing each variable \( x_i \) in \( w \) by \( t_i \) if and only if \( (x_i = t_i) \in \mu \).

The following result will be used later.

**Proposition 1** Let \( E_1 \) and \( E_2 \) be systems of equations. Then

\[
\text{soln}(\text{soln}(E_1) \cup E_2) = \text{soln}(E_1 \cup E_2)
\]

in Clark's equality theory.

3 Structural recursion

We will assume that recursive predicates are defined by a single base clause and a single recursive clause (a simple program transformation can preprocess predicates to this form). We will also assume that the input argument that control the recursion is in the first argument position.

To say that a predicate is defined by structural recursion means, intuitively, that the recursion bound depends only on the size—according to some suitable norm—of the input arguments.

A norm \( |.| \) is a mapping from terms to natural numbers. We consider two norms for linear recursion here. For list recursion we use the length of the list, and for integer recursion we use the absolute value of the integer.

Consider a predicate \( p/n \) defined by
\[ p(R_1, \ldots, R_n) \leftarrow \Delta \]
\[ p(S_1, \ldots, S_n) \leftarrow \Psi \land p(T_1, \ldots, T_n) \land \Phi \]

The predicate is defined by \textit{structural recursion} if there exists a norm \(| \cdot |\) and integers \(k, m\), such that, for all \(\theta\) and \(1 \leq i \leq n\),

\[ |S_i\theta| - |T_i\theta| = k \quad \text{and} \quad |R_i\theta| = m. \]

The terms \(R_i, S_i, T_i\) are called \textit{recursion terms} (recursion list, recursion integer, etc.).

\textbf{Example.} We consider first the following program for reversing a list using list concatenation (‘naive reverse’).

\[
\text{nrev([], []).} \\
\text{nrev([|X|], Y) :- nrev(X, Z), append(Z, [A], Y).}
\]

It is easy to see that this is a case of structural recursion since

\[
[A \times X]\theta - |X\theta| = 1 \quad \text{and} \quad ||\theta|| = 0
\]

for all \(\theta\). \hfill \Box

\textbf{Example.} Next we consider a program for list permutation.

\[
\text{perm([], []).} \\
\text{perm([|X|], [B|Y]) :- select([|X|], B, Z), perm(Z, Y).}
\]

\[
\text{select([|X|], A, X).} \\
\text{select([|X|], B, [A|Y]) :- select(X, B, Y).}
\]

This program is also defined by structural recursion since

\[
[A \times X]\theta - |Z\theta| = 1 \quad \text{and} \quad ||\theta|| = 0
\]

for all \(\theta\). However, this is not as easy to see as it requires inspection of \textit{select/3}. \hfill \Box

\textbf{Example.} Finally we shall see example of non-structural recursion. The following program sorts a list using the Quicksort algorithm.
qsort([], U-W).
qsort([A|X], U-W) :-
    part(X,A,Y,Z), qsort(Y,U-[A|V]), qsort(Z,V-W).

Here the values of

\[ [A | X]_\theta = [Y \theta] \quad \text{and} \quad [A | X]_\theta = [Z \theta] \]

depends on \( \theta \), which determines how \( X \) is partitioned into \( Y \) and \( Z \) by part/4.

\[ \square \]

4 SLDR-resolution

We define the operational semantics of logic programs as transition systems. A transition system is a pair \( \langle , \rightarrow \rangle \) where

- \( \rightarrow \) is a set of states;
- \( \rightarrow \subseteq \times \), \( \times \), is a state transition relation.

A state is a pair \( \langle G, \sigma \rangle \), where \( G \) is a goal and \( \sigma \) is a system of equations on solved form. The set of initial states contains states of the form \( \langle G, \emptyset \rangle \). The set of final states contains states of the form \( \langle \phi, \sigma \rangle \), where \( \sigma \) is the computed answer.

We write \( S \xrightarrow{n} S' \) for \( n \geq 1 \) state transitions \( S \rightarrow \cdots \rightarrow S' \).

A computation is any sequence of transitions \( \langle G, \emptyset \rangle \xrightarrow{n} \langle G', \sigma \rangle \) (\( n \geq 1 \)). A successful computation is of the form \( \langle G, \emptyset \rangle \xrightarrow{n} \langle \phi, \sigma \rangle \). A failed computation is of the form \( \langle G, \emptyset \rangle \xrightarrow{n} \langle G', \sigma \rangle \) where \( \text{false} \in \sigma \).

4.1 SLD-resolution

Let \( H \leftarrow \Phi \) be a clause in the program. The following rule defines a state transition (\( \rightarrow \)) in SLD-resolution.

\[
\begin{align*}
\theta = \text{soln}(\sigma \cup \{ H_1 = A \}) \\
\langle A \land G, \sigma \rangle \rightarrow \langle \Phi_1 \land G, \theta \rangle
\end{align*}
\]
This rule is valid even if $A$ is not the ‘leftmost’ atom in the goal, since the ordering of atoms in a conjunction is logically irrelevant.

An SLD-computation is a sequence of transitions where the transition relation is the one defined above.

**4.2 SLD-resolution**

Let $H \leftarrow \Psi \land H' \land \Phi$ be a structurally recursive clause where $H'$ is the recursive call. Let $n$ be the recursion bound if invoking this clause with a call $A$. The following rule defines a state transition ($\Rightarrow$) in SLD-resolution.

$$
\theta = \text{soln}(\sigma \cup \{ H_1 = A \} \cup \{ H'_i = H_{i+1} \mid 1 \leq i < n \})
$$

$$
\langle A \land G, \sigma \rangle \Rightarrow \langle \Psi_1 \land \cdots \land \Psi_n \land H'_1 \land \Phi_n \land \cdots \land \Phi_1 \land G, \theta \rangle
$$

SLD-resolution is applicable to a state $\langle A \land G, \sigma \rangle$ if $A$ is a call to a structurally recursive predicate.

We describe in Section 6 how $\{ H'_i = H_{i+1} \mid 1 \leq i < n \}$ can be precompiled to allow efficient solution.

An SLD-computation is any sequence of transitions $S_0 \triangleright_1 S_1 \triangleright_2 \cdots \triangleright_n S_n$ ($n \geq 1$) where the state transition operators $\triangleright_i$ ($1 \leq i \leq n$) are defined by

$$
\triangleright_i = \begin{cases} 
\Rightarrow, & \text{if SLD-resolution is applicable to state } S_{i-1}; \\
\rightarrow, & \text{otherwise}.
\end{cases}
$$

**4.3 Relating SLD-resolution to SLD-resolution**

We show that one SLD-resolution transition is equivalent to $n \geq 1$ SLD-resolution transitions where the recursive call is selected for reduction in each transition.

**Theorem 1** Consider a structurally recursive clause $H \leftarrow \Psi \land H' \land \Phi$ (where $H'$ is the recursive call), and let $S$ and $S'$ be states. Assume that the recursive call $H'_i$ is selected for reduction in each SLD-transition. Then $S \Rightarrow S'$ if and only if $S \Rightarrow^n S'$, for some $n \geq 1$. 

8
Proof. We consider a state \( \langle A \wedge G, \sigma \rangle \). Without loss of generality, we may assume that the recursive clause is of the form \( H \leftarrow H' \wedge \Phi \) (i.e. \( \Psi \) is empty). We prove the theorem by induction on the recursion bound \( n \).

Base \( n = 1 \). Both for SLD and SLDR we have \( \theta = \text{soln}(\sigma \cup \{ H_1 = A \}) \).
Hence \( \langle A \wedge G, \sigma \rangle \Rightarrow \langle \Phi_1 \wedge G, \theta \rangle \) and \( \langle A \wedge G, \sigma \rangle \Rightarrow \langle \Phi_1 \wedge G, \theta \rangle \).

For the induction, assume that
\[
\langle A \wedge G, \sigma \rangle \Rightarrow \langle H'_n \wedge \Phi_1 \wedge \cdots \wedge \Phi_n \wedge G, \theta \rangle
\]
and
\[
\langle A \wedge G, \sigma \rangle \xrightarrow{n} \langle H'_n \wedge \Phi_1 \wedge \cdots \wedge \Phi_n \wedge G, \theta \rangle,
\]
where \( \theta = \text{soln}(\sigma \cup \{ H_1 = A \} \cup \{ H'_i = H_{i+1} \mid 1 \leq i < n \}) \). By definition, the SLDR state transition for \( n + 1 \) is
\[
\langle A \wedge G, \sigma \rangle \Rightarrow \langle H'_{n+1} \wedge \Phi_1 \wedge \cdots \wedge \Phi_{n+1} \wedge G, \theta' \rangle,
\]
where \( \theta' = \text{soln}(\sigma \cup \{ H_1 = A \} \cup \{ H'_i = H_{i+1} \mid 1 \leq i < n + 1 \}) \). We turn now to the SLD transition from state \( \langle H'_n \wedge \Phi_1 \wedge \cdots \wedge \Phi_n \wedge G, \theta \rangle \). Selecting \( H'_n \) for reduction gives the transition
\[
\langle H'_n \wedge \Phi_1 \wedge \cdots \wedge \Phi_n \wedge G, \theta \rangle \Rightarrow \\
\langle H'_{n+1} \wedge \Phi_1 \wedge \cdots \wedge \Phi_{n+1} \wedge G, \text{soln}(\theta \cup \{ H'_n = H_{n+1} \}) \rangle.
\]
Using Proposition 1 we have \( \text{soln}(\theta \cup \{ H'_n = H_{n+1} \}) = \theta' \), and hence
\[
\langle A \wedge G, \sigma \rangle \xrightarrow{n+1} \langle H'_{n+1} \wedge \Phi_1 \wedge \cdots \wedge \Phi_{n+1} \wedge G, \theta' \rangle.
\]

\( \square \)

Corollary 1. \textit{SLDR-resolution is sound and complete.}

Let us now compare the computations of two small programs with SLD- and SLDR-resolution.

Example. We first consider the following program for list concatenation.
\[
\text{append}([], Y, Y).
\text{append}([A | X], Y, [A | Z]) :- \text{append}(X, Y, Z).
\]
For the goal append([a, b, c, d], [e], Answ) we have the following successful computations for SLD-resolution and SLDR-resolution.

\[\langle \text{append}([a, b, c, d], [e], Answ), \ldots \rangle\]
\[\quad \rightarrow \langle \text{append}([b, c, d], [e], Z_1), \ldots \rangle\]
\[\quad \rightarrow \langle \text{append}([c, d], [e], Z_2), \ldots \rangle\]
\[\quad \rightarrow \langle \text{append}([d], [e], Z_3), \ldots \rangle\]
\[\quad \rightarrow \langle \text{append}([], [e], Z_4), \ldots \rangle\]
\[\quad \rightarrow \langle \phi, \ldots \rangle\]

\[\langle \text{append}([a, b, c, d], [e], Answ), \ldots \rangle\]
\[\quad \Rightarrow \langle \text{append}([], [e], Z_4), \ldots \rangle\]
\[\quad \rightarrow \langle \phi, \ldots \rangle\]

\[\Box\]

**Example.** We next consider the lessall program again.

lessall([],_).
lessall([A|X],B) :- A > B, lessall(X,B).

For the goal lessall([1, 2, 3, 4], 0) we have the following successful computations for SLD-resolution and SLDR-resolution.

\[\langle \text{lessall}([1, 2, 3, 4], 0), \ldots \rangle\]
\[\quad \rightarrow \langle 1 > 0 \land \text{lessall}([2, 3, 4], 0), \ldots \rangle\]
\[\quad \rightarrow \langle 1 > 0 \land 2 > 0 \land \text{lessall}([3, 4], 0), \ldots \rangle\]
\[\quad \rightarrow \langle 1 > 0 \land 2 > 0 \land 3 > 0 \land \text{lessall}([4], 0), \ldots \rangle\]
\[\quad \rightarrow \langle 1 > 0 \land 2 > 0 \land 3 > 0 \land 4 > 0 \land \text{lessall}([], 0), \ldots \rangle\]
\[\quad \rightarrow \langle \phi, \ldots \rangle\]

\[\langle \text{lessall}([1, 2, 3, 4], 0), \ldots \rangle\]
These examples illustrate that SLD-resolution computes in one large step what SLD-resolution computes in several smaller steps.

5 Computing with SLD-resolution

With SLD-resolution, the invocation of a recursive procedure first matches the call with the procedure head and then computes the procedure body. A computation with recursion bound $n$ thus interleaves $n$ matchings with $n$ body computations. With SLD-resolution, head matching and body computation are separated—the $n$ matchings are completed before the $n$ body computations are initiated.

As discussed in the Introduction, implementing recursion by iteration in this way can avoid redundant type checking and other overheads associated with the recursive procedure calls. However, this effect can also be obtained by the call forwarding optimization technique, described by De Bosschere et al [5], where different entry points of a procedure are derived using information at the call site.

The distinguishing feature of SLD-resolution is rather that it allows new, and potentially very efficient, ways of executing recursive logic programs in parallel, since both the head unification phase and the body computation phases can be parallelized by using traditional loop parallelization techniques.

5.1 Recursion-parallelism

Consider a recursive clause with head $H$, $k$ calls $B_1, \ldots, B_k$ and a recursive call $H'$ in the body:

$$H \leftarrow B_1 \land \cdots \land B_k \land H'.$$
The goal obtained after a SLDR-resolution step with this clause is on the form:

\[ \leftarrow B_{11} \land \cdots \land B_{k1} \land \cdots \land B_{1n} \land \cdots \land B_{kn} \land H'. \]

We can depict this goal 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 is a form of AND-parallelism that we call recursion-parallelism, since it corresponds to computing all recursive invocations of the procedure in parallel.

We refer to the rows of the matrix as recursion levels. Typically \( k \ll n \), that is, there are typically many more recursion levels in a computation than calls within each recursion level. This suggests that the potential for parallel speedup is greater with recursion-parallelism than with traditional AND-parallelism. Of course, it is possible to combine recursion-parallelism and AND-parallelism—this would result in \( n \times k \) parallel processes.

Recursion-parallelism cannot be exploited with SLD-resolution, since the recursive calls must be sequenced. However, a restricted form, which we call recursion pipelining, can be obtained. The calls at one level can then start computing once the recursive call at the preceding level has been reduced; it is not necessary that the other calls at the preceding level have terminated.

**Example.** Consider the recursive clause of the lessall program:

\[
\text{lessall}(\text{[A|X]}, \text{B}) \leftarrow \text{A} \succ \text{B}, \text{lessall(X,B)}. 
\]

The matrix has only one column, since there is only one call, besides the recursive call, in the clause body:

\[
\begin{pmatrix}
A_1 > B \\
A_2 > B \\
\vdots \\
A_n > B
\end{pmatrix}
\]
There are, of course, no opportunities for parallelism within each row of this matrix. But the $n$ rows can be computed in parallel.

Recursion pipelining in SLD-resolution will not speed up this program, since the comparison $A > B$ is cheaper than a procedure call.

**Example.** The following program compares a sequence $B$ with a list of sequences. Each comparison, carried out by `match/3`, computes a similarity value $V$ that is stored in a sorted tree $T$ for later access (the tree is implemented as an incomplete data structure).

```prolog
match_segs([], _, _).
match_segs([A|X], B, T) :-
    match(A, B, V),
    put_in_tree(T, V),
    match_segs(X, B, T).
```

Here the matrix corresponding to a computation with recursion bound $n$ has two columns:

\[
\begin{pmatrix}
\text{match}(A_1, B, V_1) & \text{put}_{\text{in-tree}}(T, V_1) \\
\text{match}(A_2, B, V_2) & \text{put}_{\text{in-tree}}(T, V_2) \\
\vdots & \vdots \\
\text{match}(A_n, B, V_n) & \text{put}_{\text{in-tree}}(T, V_n)
\end{pmatrix}
\]

The $n$ calls to `put_{in-tree}/2` will be sequenced by synchronization on the shared variable $T$ (but a clever implementation will try to lock only a part of the tree). Nevertheless, execution of the rows can to quite a large extent proceed in parallel. In contrast, there is almost no scope for parallelism within each row.

### 5.2 Unification parallelism

SLD-resolution can hardly ever benefit from parallel unification, since the unifications spawned are too small. In contrast, SLDR-resolution spawns much larger unification problems that are amenable to parallel solution. Consider, for example, the append program again:
append([], Y, Y).
append([A | X], Y, [A | Z]) :- append(X, Y, Z).

This program involves list traversal and construction that cannot be parallelized in SLD-resolution. But it is well-known that, using $p$ processors, a linked list structure with $n$ elements can be traversed in $\lceil n \log n \rceil / p$ steps given only a pointer to the first list cell [8]. Furthermore, a list of $n$ elements can be constructed in $\lceil n/p \rceil$ steps using $p$ processors. Barklund has designed a parallel unification algorithm using these techniques [1] that can be exploited with SLDR-resolution.

In this paper we restrict the discussion to unification-based logic programming languages. However, SLDR-resolution can also be applied to constraint logic programming. The analog of parallel unification is then parallel constraint solving. SLDR-resolution with constraints has the effect of adding large chunks of new constraints to the constraint system when a recursive program is invoked. This can be important in case the constraint solver has an overhead when new components are added. The overhead is multiplied when executing recursive programs with SLD-resolution, since new constraints are then added much more frequently.

5.3 Breaking some data dependencies by program transformation

Consider the naive reverse program:

nrev([], []).
nrev([A | X], Y) :- nrev(X, Z), append(Z, A, Y).

The calls to append/3 depend on each other—the output of one call is the input of another—so the available parallelism in this program is limited. We will now consider a method for breaking the data dependencies of this program.

Assume that we can compute the relative sizes of certain terms in the program at compile-time (several algorithms for this task are known [7, 15]). Hence we can infer that if $\text{size}([A | X]) = n + 1$, then $\text{size}(Z) = \text{size}(X) = n$. We can, therefore, rewrite the program as follows, where the goal length($X, N$) computes the number of elements $N$ in the list $X$. 

14
nrev(X,Y) :- length(X,N), nrev(N,X,Y).

nrev(0,[] []).nrev(N+1,[A|X],Y) :- nrev(N,X,Z), append(N,Z,[A],Y).

(We allow simple arithmetic expressions in clause heads—this can be viewed as syntactic sugar.) A similar transformation of append/3 gives:

append(0,[],Y,Y).
append(N+1,[A|X],[Y],[A|Z]) :- append(N,X,Y,Z).

Now, suppose the program is called with a list of four elements:

| ?- nrev([e,f,g,h],Answ).

Reducing the subsequent calls to nrev/3 yields the goal

:- nrev(0,X,Z),
   append(0,Z,[h],Y4),
   append(1,Y4,[g],Y3),
   append(2,Y3,[f],Y2),
   append(3,Y2,[e],Answ).

Suppose now that the four calls to append/4 start computing in parallel. The following events will then take place more or less simultaneously (we assume that write accesses to shared variables are locked):

- the call append(0,Z,[h],Y4) binds Y4 to [h];
- the call append(1,Y4,[g],Y3) binds Y3 to [g];
- the call append(2,Y3,[f],Y2) binds Y2 to [f];
- the call append(3,Y2,[e],Answ) binds Answ to [e].

Depending on the exact timing of these events, the lists Y3, Y2 and Answ might have further elements instantiated when they are created. Otherwise, the list elements are instantiated after the lists are created.
The advantage of the transformed program is that each call to append can produce a list although the input list argument is not yet available. This is possible since the length of the not-yet-available list is known.

The result of the transformation is thus that each consumer of a list produces its own input list; the (former) producer then merely instantiates the individual elements of the list. This demonstrates the power of logical variables—they allow us to refer to the result of a computation before the result is actually computed.

For the transformation to work, the first argument to nrev must be instantiated to a list. This is important, since calling the transformed program with an unbound variable in the first argument position might cause non-termination.

6 Compiling SLDR-resolution

In this section we give an overview of the compilation scheme. In the subsequent sections we go into detail on precompilation of recurrence equations that occur in the compilation scheme and on code generation.

Recall that for a recursive clause

\[ H \leftarrow \Psi \land H' \land \Phi \]

a state transition in SLDR-resolution is defined as

\[
\theta = \text{soln}(\sigma \cup \{ H_1 = A \} \cup \{ H'_i = H_{i+1} \mid 1 \leq i < n \})
\]

\[
\langle A \land G, \sigma \rangle \Rightarrow \langle \Psi_1 \land \cdots \land \Psi_n \land H'_1 \land \Phi_n \land \cdots \land \Phi_1 \land G, \theta \rangle
\]

where \( A \) is the call to the clause. The compiled code that we generate is:

\[
\begin{align*}
n & \leftarrow \text{sizeof}(A); \\
\theta & \leftarrow \text{soln}(\sigma \cup \{ H_1 = A \} \cup \{ H'_i = H_{i+1} \mid 1 \leq i < n \}); \\
\text{for } i & \leftarrow 1 \text{ to } n \text{ do } \Psi_i \theta; \\
\text{call } & H'_n \theta; \\
\text{for } i & \leftarrow n \text{ down to } 1 \text{ do } \Phi_i \theta;
\end{align*}
\]

Parallelism can be exploited in computing \( \theta \) and in the for-loops executing the clause body. Thus, running the compiled program results in a four-phase computation:
1. A big head unification, corresponding to the $n$ small head unifications with SLD-resolution, is performed.

2. All $n$ instances of the calls to the left of the recursive call are computed in a parallel loop.

3. The program is called recursively. This call is known to match the base clause, and is thus trivially cheap.

4. All $n$ instances of the calls to the right of the recursive call are computed in a parallel loop.

**Example.** Consider the following program for checking whether all elements of a list are different. We assume that \texttt{notin}(A, X) holds if $A$ is not a member of the list $X$.

\[
\text{diffall}([]).
\]
\[
\text{diffall}([A|X]) :- \text{notin}(A,X), \text{diffall}(X).
\]

Suppose that the program is invoked by the call

\[
| ?- \text{diffall}([1,2,3,4]).
\]

Figure 1 shows how the computation proceeds.

\[
\square
\]

7 Precompiling recurrence equations

Recall again the definition of state transition in SLDR-resolution:

\[
\theta = \text{soln}(\sigma \cup \{ H_1 = A \} \cup \{ H'_i = H_{i+1} \mid 1 \leq i < n \})
\]
\[
\langle A \land G, \sigma \rangle \Rightarrow \langle \Psi_1 \land \cdots \land \Psi_n \land H'_n \land \Phi_n \land \cdots \land \Phi_1 \land G, \theta \rangle
\]

We shall discuss how to precompile \{ $H'_i = H_{i+1} \mid 1 \leq i < n$ \} so that it can be solved efficiently. The compilation corresponds to obtaining closed-form expressions for this system of equations.
<table>
<thead>
<tr>
<th>Phase</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$[1, 2, 3, 4] = [A1, A2, A3, A4]$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{notin}(1, [2, 3, 4])$ &lt;br&gt;$\text{notin}(2, [3, 4])$ &lt;br&gt;$\text{notin}(3, [4])$ &lt;br&gt;$\text{notin}(4, [])$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{diffall}([])$</td>
</tr>
<tr>
<td>4</td>
<td>$-$</td>
</tr>
</tbody>
</table>

Figure 1: Execution of the call $\text{diffall}([1, 2, 3, 4])$. The calls to $\text{notin}/2$ in phase 2 are computed in parallel.
7.1 Variable expansions

Consider a recursive clause \( H \leftarrow \Psi \land H' \land \Phi \), where \( H' \) is the recursive call, and its \( n \) variants \( H_i \leftarrow \Psi_i \land H'_i \land \Phi_i \). We define, for \( n \geq 1 \),

\[
\mathcal{E}_n = \begin{cases} 
0, & n = 1; \\
\{ H'_i = H_{i+1} \mid 1 \leq i < n \}, & n > 1.
\end{cases}
\]

We are interested in \( \mu_n = \text{soln}(\mathcal{E}_n) \) for \( n > 1 \) (by definition \( \mu_1 = \emptyset \)).

A variable \( X \) in a recursive clause is represented by \( n \) variant variables \( X_1, X_2, \ldots, X_n \) in a computation of recursion bound \( n \). The expansion \( \overline{X} \) of \( X \) is then the sequence

\[
X_1\mu_n, \ldots, X_n\mu_n.
\]

The expansion of a variable captures the bindings of the variable during a computation, considering only the unifications in the recursive calls to the procedure in which the variable occurs.

The notion of expansions is easily extended to compound terms. The expansion of a compound term \( T \) is the sequence

\[
T_1\mu_n, \ldots, T_n\mu_n.
\]

7.2 Classification of variables

We show in this section how the expansion

\[
X_1\mu_n, \ldots, X_n\mu_n
\]

are derived for different (syntactically distinguishable) classes of variables. The expansions are derived using only information available in the solution \( \mu_2 \) of \( \mathcal{E}_2 \), which can be obtained at compile-time.

The reasoning behind the classification is as follows. Consider a variable \( X \). Either \( \mu_2 \) does not bind \( X \) or it binds \( X \) to a term \( T \). In the latter case, either \( X \) occurs in the recursive call and \( T \) occurs in the clause head, or the other way around. We thus have three cases—we refer to these as none, pos, and neg.
What is the point of this classification? It is simply this: if we know how a variable is bound in one recursive invocation, then we know how it will be bound in all other invocations.

We use the following clause as a running example in the discussion.

\[
p([A|X], Y, f(A), W, U) :- p(X, [A|Y], Z, g(A), U).
\]  

(*)

7.2.1 none-variables

A variable \(X\) is a none-variable if it is not bound in \(\mu_2\):

\[
X_1\mu_2 = X_1 \quad \text{and} \quad X_2\mu_2 = X_2.
\]

Hence \(A\) is a none-variable in the clause (*).

Theorem 2 If \(X_1\mu_2 = X_1\) and \(X_2\mu_2 = X_2\) then \(X_i\mu_n = X_i\), for \(1 \leq i \leq n\).

Proof. Immediate.

Example. If \(\mu_2 = \emptyset\) and \(n = 4\) then:

\[
\begin{array}{c|cccc}
  i & 1 & 2 & 3 & 4 \\
  \hline
  X_i\mu_i & X_1 & X_2 & X_3 & X_4 \\
\end{array}
\]

\(\square\)

7.2.2 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 in the clause head:

\[
X_1\mu_2 = T_2\mu_2.
\]

Hence \(X\) and \(Z\) are pos-variables in the clause (*).

Theorem 3 If \(X_1\mu_2 = T_2\mu_2\) then \(X_i\mu_n = T_{i+1}\mu_n\), for \(1 \leq i < n\).

20
Proof. An easy induction on $n$, using $X\mu_{n+1} = X\mu_n\mu_{n+1}$, for any $X$ (which follows from $\mu_n \subseteq \mu_{n+1}$). □

The proofs of Theorems 4–7 below are similar and therefore omitted.

Example. If $\mu_2 = \{ X_1 = f(Y_2) \}$ and $n = 4$ then:

$$
\begin{array}{c|cccc}
  i & 1 & 2 & 3 & 4 \\
  X_i\mu_i & f(Y_2) & f(Y_3) & f(Y_4) & X_4
\end{array}
$$

□

7.2.3 neg-variables

A variable $X$ is a neg-variable if it occurs in the clause head, and there is a term $T$ in the corresponding position in the recursive call:

$$X_2\mu_2 = T_1\mu_2.$$

Hence $Y$ and $W$ are neg-variables in the clause ($*$$*$).

Theorem 4 If $X_2\mu_2 = T_1\mu_2$ then $X_i\mu_n = T_{i-1}\mu_n$, for $1 < i \leq n$.

Example. If $\mu_2 = \{ X_2 = f(Y_1) \}$ and $n = 4$ then:

$$
\begin{array}{c|cccc}
  i & 1 & 2 & 3 & 4 \\
  X_i\mu_i & X_1 & f(Y_1) & f(Y_2) & f(Y_3)
\end{array}
$$

□

7.3 Special cases of pos and neg

More direct and efficient expressions for the expansions can be inferred for certain special cases of pos- and neg-variables. We shall discuss three particularly simple, but common, special cases here.
7.3.1 inv-variables

A pos- or neg-variable $X$ is an *inv-variable* if it occurs both in the clause head and in the corresponding position in the recursive call:

$$X_1\mu_2 = X_2\mu_2.$$  

Hence $U$ is an inv-variable in the clause $(*).$

**Theorem 5** If $X_1\mu_2 = X_2\mu_2$ then $X_i\mu_n = X_1\mu_n$, for $1 < i \leq n.$

**Example.** If $\mu_2 = \{X_1 = X_2\}$ and $n = 4$ then:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_i\mu_i$</td>
<td>$X_1$</td>
<td>$X_1$</td>
<td>$X_1$</td>
<td>$X_1$</td>
</tr>
</tbody>
</table>

7.3.2 poslist-variables

A pos-variable $X$ is a *poslist-variable* if it occurs in the recursive call, and there is a list $[A \mid X]$ in the corresponding position in the clause head:

$$X_1\mu_2 = [A_2 \mid X_2]\mu_2.$$  

Hence $X$ is a poslist-variable in the clause $(*).$

**Theorem 6** If $X_1\mu_2 = [A_2 \mid X_2]\mu_2$ then $X_i\mu_n = [A_{i+1}, \ldots, A_n \mid X_n]\mu_n$, for $1 \leq i < n.$

**Example.** If $\mu_2 = \{X_1 = [A_2 \mid X_2]\}$ and $n = 4$ then:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_i\mu_i$</td>
<td>$[A_2, A_3, A_4 \mid X_4]$</td>
<td>$[A_3, A_4 \mid X_4]$</td>
<td>$[A_4 \mid X_4]$</td>
<td>$X_4$</td>
</tr>
</tbody>
</table>
7.3.3 neglist-variables

A neg-variable $X$ is a neglist-variable if it occurs in the clause head, and there is a list $[A\mid X]$ in the corresponding position in the recursive call:

$$X_{2\mu_2} = [A_1\mid X_1]\mu_2.$$ 

Hence $Y$ is a neglist-variable in the clause $(*)$.

**Theorem 7** If $X_{2\mu_2} = [A_1\mid X_1]\mu_2$ then $X_{i\mu_n} = [A_{i-1},\ldots,A_1\mid X_1]\mu_n$, for $1 < i \leq n$.

**Example.** If $\mu_2 = \{X_2 = [A_1\mid X_1]\}$ and $n = 4$ then:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_{i\mu_i}$</td>
<td>$X_1$</td>
<td>$[A_1\mid X_1]$</td>
<td>$[A_2,A_1\mid X_1]$</td>
<td>$[A_3,A_2,A_1\mid X_1]$</td>
</tr>
</tbody>
</table>

7.4 Variables belonging to several classes

A variable may belong to more than one class. Suppose, for example, that a variable $X$ is classified both as a pos-variable and as a neg-variable. We have then two expressions for the expansion $\overline{X}$ of $X$: $\overline{X}_+$ and $\overline{X}_-$, say. The fact that $X$ is both a pos-variable and a neg-variable implies that neither $\overline{X}_+$ nor $\overline{X}_-$ gives a complete description of $\overline{X}$. The complete description is obtained by solving the equation $\overline{X}_+ = \overline{X}_-$.

**Example.** Consider the following recursive clause.

$$p(X,f(Y)) :- p(f(Z),X).$$

We have $\mu_2 = \{X_2 = f(Z_1),X_1 = f(Y_2)\}$. Hence $X$ is both a pos-variable and a neg-variable. We have the following expansions for pos and neg.

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>$\cdots$</th>
<th>$n-1$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{X}_+$</td>
<td>$f(Y_2)$</td>
<td>$f(Y_3)$</td>
<td>$\cdots$</td>
<td>$f(Y_n)$</td>
<td>$X_n$</td>
</tr>
<tr>
<td>$\overline{X}_-$</td>
<td>$X_1$</td>
<td>$f(Z_1)$</td>
<td>$\cdots$</td>
<td>$f(Z_{n-2})$</td>
<td>$f(Z_{n-1})$</td>
</tr>
</tbody>
</table>
The expansion of $X$ is obtained by solving the equation $\overline{X}_+ = \overline{X}_-$:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>$\cdots$</th>
<th>$n-1$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{X}$</td>
<td>$f(Y_2)$</td>
<td>$f(Y_3)$</td>
<td>$f(Y_4)$</td>
<td>$\cdots$</td>
<td>$f(Y_n)$</td>
<td>$f(Z_{n-1})$</td>
</tr>
<tr>
<td>$\overline{Y}$</td>
<td>$Y_1$</td>
<td>$Y_2$</td>
<td>$Z_1$</td>
<td>$\cdots$</td>
<td>$Z_{n-3}$</td>
<td>$Z_{n-2}$</td>
</tr>
</tbody>
</table>

\[ \square \]

**Example.** Consider a slight variation of the preceding example:

$p(X,f(Y)) :- p(g(Z),X).$

As before, $X$ is both a pos-variable and a neg-variable. The expansions for pos and neg are now:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>$\cdots$</th>
<th>$n-1$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{X}_+$</td>
<td>$f(Y_2)$</td>
<td>$f(Y_3)$</td>
<td>$\cdots$</td>
<td>$f(Y_n)$</td>
<td>$X_n$</td>
</tr>
<tr>
<td>$\overline{X}_-$</td>
<td>$X_1$</td>
<td>$g(Z_1)$</td>
<td>$\cdots$</td>
<td>$g(Z_{n-2})$</td>
<td>$g(Z_{n-1})$</td>
</tr>
</tbody>
</table>

We note that the equation $\overline{X}_+ = \overline{X}_-$ can be solved only if $n = 1$ or $n = 2$. This corresponds to the easily verified fact that for this clause the equation system

\[ \mathcal{E}_n = \begin{cases} \emptyset, & n = 1; \\ \{ H_i' = H_{i+1} \mid 1 \leq i < n \}, & n > 1. \end{cases} \]

has a solution only for $n = 1$ and $n = 2.$

\[ \square \]

## 8 Code generation

In this section we discuss SLDR-specific compilation of recursive clauses. An important issue in AND-parallel logic programming is how to combine non-determinism with AND-parallelism. We will not consider this issue here, since a solution that works for SLD-resolution will also work for SLDR-resolution. To see this, note that computing with SLDR-resolution is operationally equivalent to computing with SLD-resolution after a two-step program transformation:

1. If there are several recursive clauses, then combine them into one, using disjunctions and explicit unifications in the clause body.
2. Unfold the recursive call in the clause.

Both steps preserve the nondeterministic behaviour of the procedure. Hence there is no need for different nondeterminism-handling techniques in SLDR-resolution than in SLD-resolution.

Other issues that we will not consider, since they are handled as in compilation of SLD-resolution, include clause selection and compilation of non-recursive clauses. We will describe compiled code at a fairly high level of abstraction, ignoring, e.g., dereferencing and trailing of variables, and synchronization of processes.

We describe compiled code for recursive clauses only for the case when the recursion argument is instantiated. We assume that a sequential version of the code (e.g., standard WAM [17] code) is invoked when the recursion argument is an unbound variable.

### 8.1 Notation

The pseudo-code notation used below is fairly standard. However, there are a few exceptions:

- Unification of $x$ and $y$ is denoted by $x = y$.
- We use the construct `$\forall i \leftarrow k \text{ to } m \text{ do } E`$ to describe parallel iteration. It means that $m - k + 1$ instances of the statement $E$ are computed in parallel. (We also allow the variant `$\forall i \leftarrow m \text{ downto } k \text{ do } E`$; the difference matters only if the loops are executed sequentially.)

### 8.2 Representation of variables

How do we represent variables in the compiled program? The idea is that a variable is represented at run-time by its expansion implemented as a vector. (The reader might notice the analogy with scalar expansion [13] in compilation of Fortran.)

The expansion of a pos-variable $X$ is $<T_2, \ldots, T_n, X_n>$, where $T_i$ is the $i$th element of the expansion of some term $T$. Hence we need not explicitly
represent the first $n-1$ elements of the expansion of $X$. References to these elements can be replaced by references to $T_2, \ldots, T_n$.

Similarly, the expansion of a neg-variable $X$ is $\langle x_1, T_1, \ldots, T_{n-1} \rangle$. References to the $n-1$ last elements of the expansion of $X$ can be replaced by references to $T_1, \ldots, T_{n-1}$. Thus we represent the expansion of a pos- or neg-variable $X$ as a scalar variable in the compiled program.

An inv-variable $X$ has the expansion $\langle X_1, \ldots, X_1 \rangle$. It can be represented by a scalar variable.

The expansion of a poslist- or neglist-variable $X$ is represented by a vector if $X$ occurs in the clause body outside the recursive call; otherwise it is represented by a scalar holding the $n$th element of $X$’s expansion. The expansion of a none-variable is represented by a vector.

**Example.** Consider the recursive clause of the list reversal program:

$$\text{nrev}(\text{[A} \text{|X} \text{]}, Y) \leftarrow \text{nrev}(\text{X}, \text{Z}), \text{append}(\text{Z}, \text{[A]}, \text{Y})\,.$$ 

The classification of the variables in this program, and the representation of their expansions, is shown in Figure 2. Note that we have classified $Y$ as a neg-variable and $Z$ as a none-variable; we could instead have classified $Y$ as a none-variable and $Z$ as a pos-variable (the choice is arbitrary). The expansions for $n = 4$ are:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\overline{A}$</td>
<td>$A_1$</td>
<td>$A_2$</td>
<td>$A_3$</td>
<td>$A_4$</td>
</tr>
<tr>
<td>$\overline{X}$</td>
<td>$[A_2, A_3, A_4 \mid X_4]$</td>
<td>$[A_3, A_4 \mid X_4]$</td>
<td>$[A_4 \mid X_4]$</td>
<td>$X_4$</td>
</tr>
<tr>
<td>$\overline{Z}$</td>
<td>$Z_1$</td>
<td>$Z_2$</td>
<td>$Z_3$</td>
<td>$Z_4$</td>
</tr>
<tr>
<td>$\overline{Y}$</td>
<td>$Y_1$</td>
<td>$Z_2$</td>
<td>$Z_3$</td>
<td>$Z_4$</td>
</tr>
</tbody>
</table>

8.3 **Head Unification**

Head unification is divided 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 traditional head unification that will
compute the expansions of some, but not necessarily all, variables in the clause.

- In the second phase we compute the remaining variable expansions.

In this section we discuss the first phase. We refer to the arguments of the invoking call as A1, A2, etc.

**Warren’s unification scheme**

We start by reviewing Warren’s scheme for unification of terms in the invoking call with terms in the clause head [16, 17]. Let us first consider unification of a call argument Ai with a variable X in the clause head. We distinguish between the first and subsequent occurrences of X in encoding this unification:

First occurrence of X: \( X \leftarrow Ai \)

Subsequent occurrences of X: \( X = Ai \)

Next we consider unification of a call argument Ai with a compound term in the clause head. We restrict the discussion to unification of lists, since other compound terms are compiled analogously. So, assume that the term in the clause head is a list \([X \mid Y]\) where Y is not a poslist-variable (we will treat the case where Y is a poslist-variable later). If Ai is a non-empty list, then we unify its head with X and its tail with Y. If Ai is an unbound variable we create a list \([X \mid Y]\) and unify it with the variable in the call. Following Warren we say that the unification proceeds in *read mode* in the former case, and in *write mode* in the latter case. We assume that these are the
first occurrences of $X$ and $Y$ (otherwise the code is modified in accordance with the discussion above):

Read mode: \[ X \leftarrow \text{head}(Ai) \]
\[ Y \leftarrow \text{tail}(Ai) \]

Write mode: \[ X \leftarrow \text{new\_variable}() \]
\[ Y \leftarrow \text{new\_variable}() \]
\[ Ai = \text{cons}(X,Y) \]

List unification

We diverge from Warren’s scheme in the case of lists whose tails are poslist-variables. The recursion list in the clause head is always such a list, but other lists can also have tails that are poslist-variables. For example, the first and third arguments in the head of `append/3` are such lists:

\[
\text{append}([A|X], Y, [A|Z]) := \text{append}(X,Y,Z).
\]

We discuss these two kinds of lists in turn.

The recursion list

Assume that $Ai$ is the recursion list in the call and that $[X \mid Y]$ is the recursion list in the head. We compile the unification of $Ai$ with $[X \mid Y]$ to the operation

\[
\langle n, x, y \rangle \leftarrow \text{traverse}(Ai)
\]

which has the following effects when $Ai$ is a list:

- $n$ is assigned the number of elements in the list $Ai$;
- the elements of $Ai$ are assigned to the vector $x[1: n]$;
- the successive tails of $Ai$ are assigned to the vector $y[1: n]$ in the case that $Y$ occurs in the clause body outside the recursive call; otherwise the last tail of $Ai$ is assigned to the scalar variable $y$. 

A straight-forward sequential implementation of this operation (for the case when \( y \) is a vector) is:

\[
\begin{align*}
n &\leftarrow 0; \\
\textbf{while} \ \text{is\_list}(Ai) \ \textbf{do} \\
& n \leftarrow n + 1; \\
& x[n] \leftarrow \text{head}(Ai); \\
& y[n] \leftarrow \text{tail}(Ai); \\
& Ai \leftarrow \text{tail}(Ai);
\end{align*}
\]

\textbf{od}

Barklund [1] defines a data-parallel algorithm for this operation. The algorithm runs in \( O(\log n) \) time for linear lists (a list is linear if no variable occurs more than once).

We assume the following behaviour of the operation \( \langle n, x, y \rangle \leftarrow \text{traverse}(Ai) \) when \( Ai \) is an unbound variable: \( Ai \) is bound to a new list cell \( \text{cons}(x[1], y[1]) \) and \( n \) is set to 1.

\textit{Other lists with poslist tails}

Two additional issues that must be dealt with for other lists with poslist tails, such as the list \([A | Z]\) in the append program above.

First, the list in the call might be only partly constructed. Say, for example, that the list has five elements and ends with a variable, while the recursion list in the call has 20 elements so that \( n = 20 \). Then the head unification code for this list must match (‘read mode’) the five elements and then complete (‘write mode’) the list with 15 more elements.

Second, it is possible that the list does not represent the first occurrence of the list-head variable. For example, the list \([A | Z]\) does not represent the first occurrence of the variable \( A \) in the append program. In this case, the traverse operation is modified to \texttt{unify} \ the list elements. Although probably less common, the same modification might be required for the list-tail variable.

Let \( Y \) be a poslist-variable and consider unification of \([X | Y]\) with a list or variable \( Ai \) in the call. We assume that this is not the first occurrence of \( X \) and that the operation traverse\* is a modification of traverse that captures this difference. We also assume that \( Y \) is represented as a vector. The
Variable | Class | Representation
---|---|---
A | none | vector
X | poslist | scalar
Y | inv | scalar
Z | poslist | scalar

Figure 3: Variable classification and representation in `append/3`.

code for the unification is (we assume that a traverse operation has already computed `n`)

\[
\langle k, x, y \rangle \leftarrow \text{traverse}^{*}(Ai);
\]

\[\text{if } k < n \text{ then} \]

\[y[n] \leftarrow \text{new\_variable}();\]

\[\text{for } i \leftarrow n \text{ down to } k + 1 \text{ do } y[i] \leftarrow \text{cons}(x[i], y[i]);\]

\[\text{fi}\]

A parallel version of this code is:

\[
\langle k, x, y \rangle \leftarrow \text{traverse}^{*}(Ai);
\]

\[\text{if } k < n \text{ then} \]

\[y[n] \leftarrow \text{new\_variable}();\]

\[\text{forall } i \leftarrow k \text{ to } n - 1 \text{ do } y[i] \leftarrow \text{cons}(x[i + 1], \text{nil});\]

\[\text{forall } i \leftarrow k \text{ to } n - 1 \text{ do } \text{tail}(y[i]) \leftarrow y[i + 1];\]

\[\text{fi}\]

(A remark on notation: pointer dereferencing is here denoted by `*` so that if `x` holds the integer 42, then `*x \leftarrow 17` stores 17 in memory location 42.)

Both loops in this code are parallel. In case `k = 1`, a list of `n - 1` elements is constructed with a very high degree of parallelism.

**Example.** Consider again the recursive clause of the `append` program:

\[
\text{append}([A | X], Y, [A | Z]) \leftarrow \text{append}(X, Y, Z).
\]

The variable classification and representation is given in Figure 3. We assume that the first argument is the recursion list. The compiled clause head of `append/3` is then:
\begin{verbatim}
\langle n, a, x \rangle \leftarrow \text{traverse}(A1);
y \leftarrow A2;
\langle k, a, z \rangle \leftarrow \text{traverse}(A3);
\text{if } k < n \text{ then }
   \begin{align*}
   &\text{tmp} \leftarrow \text{new\_variable}();
   &\text{for } i \leftarrow n \text{ downto } k + 1 \text{ do }
   \begin{align*}
   &\text{tmp} \leftarrow \text{cons}(a[i], \text{tmp});
   \end{align*}
   \end{align*}
   z \leftarrow \text{tmp}
\end{verbatim}

Here traverse* is the modified version of traverse mentioned earlier.

\section*{8.4 Computing remaining variable expansions}

In order to implement our control scheme of first computing all head unifications, and then computing the clause body in bounded loops, we must have access to the variable expansions when entering the loops.

No extra computation, in addition to head unification as described above, is required in order to obtain the expansions of neg-, inv-, and poslist-variables.

Additional code is required for computing the expansions of none-, pos-, and neglist-variables:

- A pos-variable \( X \) is created by the assignment \( x \leftarrow \text{new\_variable}() \).

- The code for computing the expansion of a none-variable \( X \) depends on whether or not the variable occurs in the clause head. In the former case, it also depends on where in the clause it occurs.
  
  - If \( X \) occurs in the head in a list whose tail is a poslist-variable (the recursion list, for example), then its expansion is constructed in head unification by the traverse operation.
  
  - If \( X \) occurs in the head, but not in a list whose tail is a poslist-variable, then the first element of its expansion is constructed by head unification. The remaining parts are constructed by:

    \begin{verbatim}
    \text{if } n > 1 \text{ then forall } i \leftarrow 2 \text{ to } n \text{ do } x[i] \leftarrow \text{new\_variable()} \text{ fi}
    \end{verbatim}

- If \( X \) does not occur in the head, then its expansion is constructed by:
\[ \text{forall } i \leftarrow 1 \text{ to } n \text{ do } x[i] \leftarrow \text{new_variable()} \]

- Let \( [X \mid Y] \) be a list where \( Y \) is a neglist-variable. Then \( y[1] \) is created by head unification and we can also assume that the expansion \( x[1 : n] \) of \( X \) is available. The remaining parts of the expansions of \( Y \) is constructed by:

\[ \text{for } i \leftarrow 2 \text{ to } n + 1 \text{ do } y[i] \leftarrow \text{cons}(x[i - 1], y[i - 1]) \]

(Note that the vector \( y \) has \( n + 1 \) elements, where \( y[n + 1] \) is a pointer to the entire list; this is convenient when setting up the arguments of the recursive call. This trick is due to Thomas Lindgren.)

The loops for constructing the expansion of a none-variable are parallel. A parallel implementation of the code for constructing the expansion of a neg-list variable is:

\[ \begin{align*}
\text{forall } i & \leftarrow 2 \text{ to } n + 1 \text{ do } y[i] \leftarrow \text{cons}(x[i - 1], \text{nil}); \\
\text{forall } i & \leftarrow 2 \text{ to } n + 1 \text{ do } *\text{tail}(y[i]) \leftarrow y[i - 1];
\end{align*} \]

It might enhance efficiency to create the expansions of none- and neglist-variables as close to their use (as arguments to body calls) as possible. In this way we minimize unnecessary work in the case when some body call fail. However, in the case when recursion-parallelism cannot be exploited for the body calls, we might increase parallelism by creating the expansions outside the sequential body loop, since these operations can be done in parallel. The single pos-variable should be created close to the single recursive call.

### 8.5 Clause body computation

The single recursive call of the recursive clause is known to match the base clause. However, it may also match the recursive clause, for example if the recursion list ends with an unbound variable. The appropriate arguments of the recursive call are found by replacing each program variable in the call by the \( n \)th element of its expansion.

**Example.** Consider the recursive clause of `diffall/1`:

\[ \text{diffall}(\text{X} \mid \text{Y}) :- \text{notin}(\text{A}, \text{X}), \text{diffall}(	ext{X}). \]

The code for the recursive call is: `call diffall(x[n]);`
The rest of the clause body

The calls to the left and right of the recursive call are compiled to two `forall`-loops with loop index $i = 1, \ldots, n$ as described in Section 6. Recursion-parallelism is implemented by parallelization of these loops. The loop index serves as an index to the expansions of the variables occurring in the clause body. When the loops cannot be parallelized, we employ the standard left-to-right control rule used by, for example, Prolog, when computing the $n$ instances of the clause body.

**Example.** Recall the recursive clause of the naïve reverse program:

\[
\text{nrev}([A|X], Y) :- \text{nrev}(X, Z), \text{append}(Z, [A], Y).
\]

The compiled code for the clause body is:

```
call nrev(x, y[n + 1]);
forall i ← n downto 1 do call append(y[i + 1], cons(u[i], nil), y[i]);
```

Here we store the single needed value of Z in $y[n + 1]$, in this way we need not compute `append(z, cons(u[n], nil), y[n])` outside the loop.

\[\square\]

### 8.6 Examples

**Example.** Our first example is from an insertion sort program.

\[
\text{sort}([A|X], Y) :- \text{sort}(X, Z), \text{insert}(Z, A, Y).
\]

Compiled code:

```
\langle n, a, x \rangle ← \text{traverse}(A1);
y[1] ← A2;
forall i ← 2 to n + 1 do y[i] ← \text{new_variable}();
call \text{sort}(x, y[n + 1]);
forall i ← n downto 1 do call \text{insert}(y[i + 1], a[i], y[i]);
```

As in the naïve reverse example, we store the single needed value of Z in $y[n + 1]$.

\[\square\]

**Example.** The next example is from the lessall program.
lessall([A|X],B) :- A>B, lessall(X,B).

Compiled code:
\[
\langle n, a, x \rangle \leftarrow \text{traverse}(A1);
\]
\[
b \leftarrow A2;
\]
\[
\text{forall } i \leftarrow 1 \text{ to } n \text{ do } a[i] > b;
\]
\[
\text{call } \text{lessall}(x, b);
\]

**Example.** Our final example is from a program that reverses a list using an accumulating parameter.

\[
\text{rev}([A|X],Y,Z) :- \text{rev}(X,[A|Y],Z).
\]

Compiled code:
\[
\langle n, a, x \rangle \leftarrow \text{traverse}(A1);
\]
\[
y[1] \leftarrow A2;
\]
\[
\text{for } i \leftarrow 2 \text{ to } n + 1 \text{ do } y[i] \leftarrow \text{cons}(x[i - 1], y[i - 1]);
\]
\[
z \leftarrow A3;
\]
\[
\text{call } \text{rev}(x, y[n + 1], z);
\]
This code—although sequential—is quite efficient. 

## 9 Related work

Tärnlund first noted that complete unfolding of the recursion in a logic program vastly increases the potential for AND-parallel execution [14]. In later (unpublished) work, Tärnlund devised an algorithm for carrying out the unfolding at runtime, using \(O(\log n)\) unfolding steps for recursion bound \(n\).

This approach was the initial motivation for the work reported here. The idea to use iteration to represent the completely unfolded recursion at compile-time, as well as a similar technique for nonlinear recursion on e.g. binary trees, first appeared in the author’s thesis [12].

SLDR-resolution, and the compilation method described in the present paper, is implemented in the Reform Prolog system [2, 3, 4, 11]. The system
has been implemented on Sun and Sequent shared-memory multiprocessors. High parallel efficiency (89–95% on 24 processors) and low overhead for parallelization (2–12%) have been obtained on a set of small- and medium-sized Prolog programs.

10 Conclusion

The problem we address in this paper is: How can we make parallel execution of recursive logic programs more efficient? Our solution consists of:

- A compilation technique that translates a regular form of recursion to a parallelizable form of iteration.
- An operational semantics for programs compiled with this technique.

The solution has been implemented on shared-memory multiprocessors with good results.

Acknowledgements

Thanks to Jonas Barklund, Johan Bevemyr, Björn Carlson, Keith Clark, Åke Hansson, Thomas Lindgren, and Sten-Åke Tärnlund for valuable feedback during various parts of this research. Thanks also to two referees of an earlier version of the paper—their comments improved the presentation significantly.

References


