PRESSING FOR PARALLELISM: A PROLOG PROGRAM MADE CONCURRENT

LEON STERLING AND MIKE CODISH*

▷ We describe the translation of a nontrivial program for solving equations from PROLOG to Concurrent PROLOG, and further to Flat Concurrent PROLOG. The translation from PROLOG to Concurrent PROLOG required understanding of the program but was straightforward. The translation from Concurrent PROLOG to Flat Concurrent PROLOG was more suitable to be the basis for automatic procedures. The different styles of translation used are illustrated with examples of code from the three programs. The gain in speed by performing computations in parallel is discussed. ◁

1. INTRODUCTION

An ideal of logic programming is to free the programmer from worrying about control. The task of the programmer according to this ideal is to compose the logical axioms which specify the relationship the program is to compute. How to use the axioms in a particular computation is left to the interpreter of the logic program.

The reality of logic programming using current languages is otherwise. Languages such as PROLOG and Concurrent PROLOG have well-defined operational semantics which must be understood and exploited to write practical programs. For example, clause and goal order are important in PROLOG programs. Furthermore, there are control-related primitives specific to the language, whose use must be mastered. The set is minimal, consisting of the cut in PROLOG, and the commit and the read-only annotation in Concurrent PROLOG. However, their correct use is crucial.

Address correspondence to Leon Sterling, Department of Computer Engineering and Science, and Centre for Automation and Intelligent Systems Research, Case Institute of Technology, Case Western Reserve University, Cleveland, Ohio 44106.

Received 5 September 1985; accepted 23 September 1985.

*Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

THE JOURNAL OF LOGIC PROGRAMMING

52 Vanderbilt Ave., New York, NY 10017 0743-1066/86/$03.50

Programming techniques continue to evolve based on the operational behavior of the languages. The techniques are collected and abstracted into programming paradigms. Examples are generate and test for problem solving in PROLOG, and message passing in Concurrent PROLOG. The techniques are passed into the folklore, are handed down, and determine the applications for which the languages are used.

It is an interesting, largely unstudied question how techniques in different logic programming languages correspond. More generally, one might ask how and whether programs written in the natural paradigms of one logic programming language can be translated to another. This paper concerns these questions.

We translated part of the PROLOG program PRESS [4, 13] to gain insight into the relationship between the languages. PRESS is a program developed at the University of Edinburgh for solving symbolic equations. It has evolved into a significant piece of software combining different styles of programming: deterministic algorithms, heuristic rules, and PROLOG utilities. Our Concurrent PROLOG version is CONPRESS.

The evolution from PRESS to CONPRESS required basic common sense. The translation is straightforward for procedures of the program using don't-care nondeterminism. Procedures which depend on the sequential nature of PROLOG use the synchronization capabilities of Concurrent PROLOG to achieve the same effect. The message passing features of Concurrent PROLOG were exploited in some places to save unnecessary computation.

Implementing the OR-parallel part of Concurrent PROLOG efficiently is difficult, due to the maintenance of separate guard environments. This has suggested the language Flat Concurrent PROLOG (FCP), which is essentially the AND-parallel subset of Concurrent PROLOG [12, 9]. Many Concurrent PROLOG programs are readily translated to Flat Concurrent PROLOG [1], but evaluations of the expressive power of the languages are still being made. We further successfully translated CONPRESS into FCPRESS, an equivalent version of the equation solver in Flat Concurrent PROLOG, to gain insight into the expressiveness of the language.

The translation from CONPRESS to FCPRESS was more systematic than from PRESS to CONPRESS. Three standard approaches were identified for removing the OR-parallel sections of CONPRESS. The first is running the guards in AND-parallelism and using a mutual exclusion variable to decide which guard would be chosen. The second is the compilation of the guards into if-then-else structures. The third approach involves writing specialized predicates. The approaches are connected. More details on translating Concurrent PROLOG code to FCP can be found in [5].

Our experience shows that all three languages are suitable for writing symbolic equation solvers. However the style of code is slightly different in each case. PRESS is the least explicit but the most general. It uses PROLOG's backtracking, and exploits unpredictable interactions to solve difficult equations. CONPRESS, and more especially FCPRESS, demand a more explicit elaboration of conditions.

There is an interesting underlying issue of composing parallel algorithms. Since PROLOG executes programs sequentially and Concurrent PROLOG (FCP) executes programs concurrently, a translation of a PROLOG program to Concurrent PROLOG (FCP) is the development of a parallel algorithm from a sequential one. This is potentially very powerful. Shapiro demonstrates in [11] that powerful systolic algorithms appear from naive translation to Concurrent PROLOG of pure logic

programs. Our translation of PRESS is not the ultimate in parallel equation solvers, and the form of CONPRESS is a direct result of the form of PRESS. Nonetheless there was considerable speedup in some examples due to performing part of the computation in parallel.

The outline of the paper is as follows. The next section gives a brief overview of PRESS to make the paper somewhat self-contained. The equation solving methods themselves are not described: the reader is referred to the papers on PRESS [4, 13, 3] for details. The research in this paper is based on a simplified version of PRESS. It has been partially reconstructed and cleaned up from the original to allow easier comparison with CONPRESS and FCPRESS. The complete code of the three programs used in this paper can be found in [14]. The authors are happy to send them on request.

Section 3 describes the translation process from PROLOG to Concurrent PROLOG. The process is illustrated with examples from PRESS and CONPRESS. Similarly Section 4 uses examples from CONPRESS and FCPRESS to discuss translation from Concurrent PROLOG to FCP. The next section presents statistics discussing the speedup achieved by using a parallel language, and finally some conclusions are given.

Knowledge is assumed of all three languages. Introductions to Concurrent PROLOG and to Flat Concurrent PROLOG can be found respectively in [10] and in [9].

2. MEET THE PRESSES

The top level of the equation solver is a collection of axioms defining methods. Abstractly a method can be split into two parts: a condition, called the *entrance condition*, determining whether the method is applicable, and the application of the method itself. An entrance condition is *binding* if its success guarantees that the equation will be solved correctly.

The clause below is an abstracted prototypical clause at the top level of PRESS. The basic predicate is *solve_equation(Equation,X,Solution)*. The predicate is true if *Solution* solves equation *Equation* in the unknown *X*. The predicate *condition(Equation,X)* determines whether the equation satisfies the entrance condition for the method, while *method(Equation,X,Solution)* applies the method to solve the equation:

```
solve_equation(Equation,X,Solution) :-
    condition(Equation,X), method(Equation,X,Solution).
```

Instances of this clause will be given as examples in the following sections. Four equation-solving methods were translated to provide the comparison: factorization, isolation for solving equations with a single occurrence of the unknown, a suite of polynomial methods, and a general form of change of unknown called homogenization.

All these methods, apart from homogenization, have binding entrance conditions. Since PRESS has backtracking, nonbinding entrance conditions are no problem. In CONPRESS and FCPRESS a condition sufficient to distinguish homogenization from the other methods was used.

Syntactically the top level of PRESS and CONPRESS look similar. Operationally they are different. PRESS attempts the methods in the following order: factorization, isolation, polynomial, and homogenization. CONPRESS attempts the entrance conditions of the methods in parallel, and the first to succeed commits the equation solver. The top level of FCPRESS explicitly shows the behavior of CONPRESS.

3. TRANSLATING PROLOG CODE TO CONCURRENT PROLOG

Syntactically any pure PROLOG program is a Concurrent PROLOG program. However, an arbitrary PROLOG program will usually behave incorrectly when run as a Concurrent PROLOG program. In this section we give examples of the changes necessary.

We concentrate our discussion on control issues. The goal and clause order of PROLOG programs convey implicit control information, particularly when cuts are present. This control information generally must be made explicit in the translation. The basic tools for expressing control information in Concurrent PROLOG are the commit operator and the read-only annotation.

Let us consider the task of placing a commit operator. For each clause we must define a condition, the guard, which will enable commitment to this clause from the other possible choices. This condition should be as concise as possible to reduce the amount of unnecessary computation and speed up execution.

The simplest guard is the empty guard, possible if unification successfully determines the correct choice of clause. Otherwise one must determine the appropriate test whose satisfaction establishes that the correct clause has been chosen. This is not difficult in general. The worst case is when the condition for commitment is the success of the whole body. As a guarded clause, the guard is then the whole body, and the body is empty. Examples of the two extremes can be seen in the programs in Figures 2 and 4 below.

Shapiro argues [10] that Concurrent PROLOG incorporates indeterminacy but not nondeterminism, whereas PROLOG supports both. In the terms of Kowalski [8], don't-care nondeterminism can be expressed easily, but not don't-know nondeterminism. This is reflected in our translation. Those parts of PRESS incorporating don't-care nondeterminism were the most easily and elegantly translated. More generally, code written "nondeterministically" where clause and goal orders are not important can be translated immediately.

Elegant translation is exemplified by the several parsing procedures in PRESS. Parsing procedures typically make the correct choice of clause, depending on the success of unification. Such code is almost identical when naively translated to Concurrent PROLOG. Generally the translated clauses have empty guards. A few of the clauses have simple guards easily defined as in the PROLOG code.

As an example we give the predicate *is_polynomial(X,Term)*, true if *Term* is a polynomial in *X*. Figure 1 is the version in PRESS, while Figure 2 is the equivalent code in CONPRESS.

The commit operator is concerned with the choice of the correct clause to use in a computation. In a much less clean way, cuts also indicate the correct choice of clause. During the translation process cuts are simply removed from PROLOG code. However, their placement in well-written code should not be ignored. It can suggest

```
is_polynomial(X,X) :- !.
is_polynomial(X,Term) :-
    free_of(X,Term), !.
is_polynomial(X,Term1 + Term2) :-
    !, is_polynomial(X,Term1), is_polynomial(X,Term2).
is_polynomial(X,Term1 - Term2) :-
    !, is_polynomial(X,Term1), is_polynomial(X,Term2).
is_polynomial(X,Term1* Term2) :-
    !, is_polynomial(X,Term1), is_polynomial(X,Term2).
is_polynomial(X,Term1/Term2) :-
    !, is_polynomial(X,Term1), free_of(X,Term2).
is_polynomial(X,Term^N) :-
    !, integer(N), N >= 0, is_polynomial(X,Term).
```

FIGURE 1. Recognizing polynomials in PRESS.

```
is_polynomial(X,X).
is_polynomial(X,Term) :-
    free_of(X,Term) | true.
is_polynomial(X,Term1 + Term2) :-
    is_polynomial(X,Term1), is_polynomial(X,Term2).
is_polynomial(X,Term1-Term2) :-
    is_polynomial(X,Term1), is_polynomial(X,Term2).
is_polynomial(X,Term1* Term2) :-
    is_polynomial(X,Term1), is_polynomial(X,Term2).
is_polynomial(X,Term1/Term2) :-
    is_polynomial(X,Term1), free_of(X,Term2).
is_polynomial(X,Term^N) :-
    integer(N), N >= 0, is_polynomial(X,Term).
```

FIGURE 2. Recognizing polynomials in CONPRESS.

an appropriate positioning of commit operators. Note this is true for Figures 1 and 2.

An example of a predicate less elegantly translated in *position(Sub,Term,P)*, which computes a position list *P* of a subterm *Sub* occurring in a term *Term*. The PRESS code appears as the program in Figure 3.

The translation of Figure 3 to CONPRESS is given as Figure 4. The two clauses of *position/3* are neatly translated. The clauses of *position/4* have an immediate naive translation, but are less pleasing as Concurrent PROLOG code due to the complex nature of the guards. Essentially all subterms of the term are searched in parallel.

```
position(Term,Term,[]).
position(Sub,Term,Path) :-
    compound(Term), functor(Term,_,N), position(N,Sub,Term,Path), !.
position(N,Sub,Term,[N|Path]) :-
    N > 0, arg(N,Term,Arg), position(Sub,Arg,Path).
position(N,Sub,Term,Path) :-
    N1 is N - 1, N1 > 0, position(N1,Sub,Term,Path).
```

FIGURE 3. *Position* in PRESS.

```
position(Term,Term,[ ]).
position(Sub,Term,Path) :-
    compound(Term) | functor(Term,_,N), position(N,Sub,Term,Path).
position(N,Sub,Term,[N|Path]) :-
    N > 0, arg(N,Term,Arg), position(Sub,Arg?,Path) | true.
position(N,Sub,Term,Path) :-
    M is N – 1, M > 0, position(M?,Sub,Term,Path) | true.
```

FIGURE 4. *Position* in CONPRESS.

We don't know the correct branch in advance, and this is reflected in the code. (Better versions of *position* can be written in Concurrent PROLOG. In fact *position* was rewritten in the context of FCPRESS.)

Cuts are often used in PROLOG to achieve implicit negation. Care is needed when translating clauses in a procedure appearing after a clause with a cut. Conditions that were omitted because of the cut may have to be explicitly written. This is true for the top-level equation-solving clauses.

A binding entrance condition for isolation is that there must be a single occurrence of the unknown in the equation. In PRESS, the entrance condition for homogenization, which was checked after isolation, assumed that there were multiple occurrences of the unknown in the equation. When writing the entrance condition for homogenization in CONPRESS, multiple occurrences of the unknown had to be specified.

Several parts of the equation solver implement deterministic algorithms. In contrast to the nondeterministic code, where the translation effort concerns mainly the placement of commit operators, algorithmic code must worry about synchronization.

To implement deterministic algorithms where the order of steps is important, the read-only annotation is necessary. There are two uses of read-only annotations. The first is *consumer protection*. In a clause whose body has goals with a shared variable, such as

```
a :- p(X), q(X).
```

The goal which instantiates the value of X, $p(X)$ say, is the *producer* of X, while the goal which then uses the value, $q(X)$, is the *consumer*. In PROLOG, one uses goal order in a rule to ensure that the producer is called before the consumer. In Concurrent PROLOG, one needs to synchronize the computation so that the consumer goal waits on values from the producer goal. This is done by annotating the occurrence of the variable in the consumer goal, i.e. *q(X?)* in the above clause. If q is a recursive procedure consuming a stream, then the recursive calls of q will have to protect the tail of the stream with a read-only annotation.

A typical example of deterministic code is the algorithm for isolation, the method for solving equations with a single occurrence of the unknown. The algorithm is described in [4]. Its top-level implementation in PRESS is given in Figure 5. The entrance condition, which is binding, is *single_occurrence/2*.

The necessary synchronization to translate the program in Figure 5 to CONPRESS is straightforward. The algorithm proceeds by calculating a position list locating the single occurrence of the unknown in the term in predicate *position/3*. The head of

```
solve_equation(Equation,X,Solution) :-
    single_occurrence(X,Equation), !,
    position(X,Equation,[Side|Position]),
    maneuver_sides(Side,Equation,Equation1),
    isolate(Position,Equation1,Solution).
```

FIGURE 5. Solving equations by *Isolation* in PRESS.

```
solve_equation(Equation,X,Solution) :-
    single_occurrence(X,Equation) |
    position(X,Equation,[Side|Position]),
    maneuver_sides(Side?,Equation,Equation1),
    isolate(Position?,Equation1?,Solution).
```

FIGURE 6. Solving equations by *Isolation* in CONPRESS.

the position list is used by *maneuver_sides/3* to ensure that the unknown occurs in the left-hand side of the equation, reversing the equation if necessary. The tail of the position list is used by *isolate/3* to know how to apply the appropriate rewrite rules. Also *isolate/3* must wait for the equation produced by *maneuver_sides/3*. All the necessary read-only annotations are of the consumer-protection type. The transformed code is given in Figure 6.

The second use of read-only annotations is *producer protection*, where the producer goal is responsible for protecting the incomplete part of the structure under construction [6]. Occurrences of a variable in the head of the clause are annotated, and thus passed on with protection. Producer protection is necessary when a structure in the head of a clause is being partially built by a goal G in the clause body. Commitment to that clause may occur before G has finished executing, and the partially built structure communicated to the rest of the computation. To avoid other goals incorrectly instantiating the value, the value is marked as read only in the head of the clause. An example clause is:

```
a(B,f(X?)) :- b(X).
```

Consumer protection of the second argument of the goal a is not sufficient, since the partial structure $f(X)$ is not a variable.

An example of the necessity of producer protection read-only annotation comes in the implementation of an algorithm for adding polynomials. Polynomials are converted to a *polynomial normal form* which is a list of tuples of the form (A_i, N_i), where A_i is the coefficient of X^{N_i}. In this normal form terms with zero coefficients are eliminated. The polynomial manipulation routines assume polynomials in normal form.

The predicate *add_polynomials(Xs,Ys,Zs)* adds the polynomials Xs and Ys in normal form to give a polynomial Zs in normal form. The PRESS code is given in Figure 7.

The adaptation to CONPRESS is given in Figure 8. The interesting point is the read-only annotation of A in the fourth clause of the program. This is necessary to avoid A being instantiated by other goals in the context of a computation. This was a bug discovered by experience.

```
add_polynomials([ ],Poly,Poly).
add_polynomials(Poly,[ ],Poly).
add_polynomials([(Ai,Ni)|Ps],[(Aj,Nj)|Qs],[(Ai,Ni)|Rs]) :-
    Ni > Nj, !, add_polynomials(Ps,[(Aj,Nj)|Qs],Rs).
add_polynomials([(Ai,N)|Ps],[(Aj,N)|Qs],[(A,N)|Rs]) :-
    !, A is Ai + Aj, add_polynomials(Ps,Qs,Rs).
add_polynomials([(Ai,Ni)|Ps],[(Aj,Nj)|Qs], [(Aj,Nj)|Rs]) :-
    Ni < Nj, !, add_polynomials([(Ai,Ni)|Ps],Qs,Rs).
```

FIGURE 7. Adding polynomials in PRESS.

```
add_polynomials([ ],Poly,Poly).
add_polynomials(Poly,[ ],Poly).
add_polynomials([(Ai,Ni)|Ps], [(Aj,Nj)|Qs],[(Ai,Ni)|Rs]) :-
    Ni > Nj | add_polynomials(Ps,[(Aj,Nj)|Qs],Rs).
add_polynomials([(Ai,N)|Ps],[(Aj,N)|Qs],[A?,N)|Rs]) :-
    A is Ai + Aj, add_polynomials(Ps,Qs,Rs).
add_polynomials([(Ai,Ni)|Ps],[(Aj,Nj)|Qs], [(Aj,Nj)|Rs]) :-
    Ni < Nj | add_polynomials([(Ai,Ni)|Ps],Qs.Rs).
```

FIGURE 8. Adding polynomials in CONPRESS.

The examples of code so far in this section have been essentially literal translations. All the translation involved was the correct modification of the control. Better translations are sometimes possible if features of the new language are taken into account. We demonstrate two changes here which were possible due respectively to the use of streams and the message-passing capabilities of Concurrent PROLOG.

Several of the equation-solving methods perform problem reduction. The equation to be solved is reduced to simpler equations. It seems sensible to solve the subproblems in parallel. There are several different ways of handling separate solutions or multiple solutions in the original PRESS. For example a disjunction operator, treated specially in the various methods, indicates alternative solutions. When translating from PRESS to CONPRESS we decided to make the handling of multiple equations uniform. The natural technique in Concurrent PROLOG is to use streams.

If the input to the equation solver is a stream of equations, it is necessary to associate with each equation a specific unknown. For this reason we regard an equation in CONPRESS as a tuple *(Equation,Unknown)*. The top level of CONPRESS is changed to be a predicate *solve_equation(Equations,Solution—Ss)* where *Equations* is a stream of tuples of equations and the unknown the equation is solved for, and *Solution–Ss* is a queue, represented as a difference list of the solutions. A queue is used so that the order of solutions corresponds to the order of the equations. Using streams rather than a single equation means that *solve_equations* becomes a tail-recursive process.

Streams in Concurrent PROLOG correspond to lists in PROLOG. It transpired that the extension to PRESS to solve lists of equations was natural. Our modified version of PRESS is actually written this way. Other predicates where streams were useful were filtering processes, for example Program 17 in the next section.

```
solve_equations([(Equation,X)|Eqns],Solution-Solns) :-
    single_occurrence(X,Equation), !,
    position(X,Equation,[Side|Position]),
    maneuver_sides(Side,Equation,Equation1),
    isolate(Position,Equation1,Solution-Ss),
    solve_equations(Eqns,Ss-Solns).
```

FIGURE 9. The modified top-level rule for isolation.

Figure 9 gives the appropriate modification of Figure 5 for solving an equation with a single occurrence of the unknown, which uses streams. Since the isolation procedure can give multiple solutions, it must update the queue.

Our next example of translation exploits the message-passing features of Concurrent PROLOG. A useful relationship when solving equations is the number of times one term appears as a subterm of another. Examples are determining how many times an unknown appears in an equation, whether a term is free of appearances of an unknown, and whether there is more than one occurrence of an unknown in an equation.

Relations of this kind are called *occurrence relations*. PRESS and CONPRESS use predicates *occurrences(Subterm,Term,Occs)* defining the relation that there are *Occs* occurrences of *Term* in *Subterm*, *free_of(Subterm,Term)* defining the relation that *Term* is free of occurrences of *Subterm*, and predicates *single_occurrence(Subterm,Term)* and *multiple_occurrence(Subterm,Term)* with similar definitions.

The code for occurrence predicates was rewritten for CONPRESS to take advantage of the message passing capabilities of Concurrent PROLOG. Neater code than the immediate 'naive' translation from PRESS resulted.

We could use *occurrences(Subterm,Term,Occs)* to define all occurrence relations as in the three clauses in Figure 10. This is a clean declarative definition which uniformly expresses occurrence relations in terms of *occurrences*. Procedurally these are not the best definitions in PROLOG. Instead of counting all the occurrences of the subterm in the term, *free_of* could fail upon encountering the first occurrence. Similarly, in *multiple_occurrences* it is sufficient to locate just two occurrences of the subterm in the term.

In PROLOG there is no obvious way to define these predicates capturing both efficiency and conciseness. To achieve efficient PROLOG code, each of these predicates must be rewritten. Indeed, this happened in PRESS. The original version of *free_of* had the clean declarative definition, which was later replaced by a more efficient version written by Richard O'Keefe with a structure similar to that of

```
free_of(Subterm,Term) :-
    occurrences(Subterm,Term,N), N = 0.
single_occurrence(Subterm,Term) :-
    occurrences(Subterm,Term,N), N = 1.
multiple_occurrence(Subterm,Term) :-
    occurrences(Subterm,Term,N), N > 1.
```

FIGURE 10. Defining occurrence relations.

```
free_of(Kernel,Kernel) :-
    !, fail.
free_of(Kernel,Expression) :-
    atomic(Expression), !.
free_of(Kernel,Expression) :-
    functor(Expression,_,Arity), !, free_of(Arity,Kernel,Expression).

free_of(0,Kernel,Expression) :- !.
free_of(N,Kernel,Expression) :-
    arg(N,Expression,Argument),
    free_of(Kernel,Argument),
    !, N1 is N - 1,
    free_of(N1,Kernel,Expression).
```

FIGURE 11. An efficient version of *free_of*.

occurrences. They differ in their response to occurrences of the *Subterm*: *occurrences* counts these; *free_of* will fail if one is encountered. The code is illustrated in Figure 11.

In Concurrent PROLOG the situation is different and definitions similar to those in Figure 10 can be given capturing both clarity and efficiency. All the occurrence relations can be defined in terms of the *occurrences* predicate. Correct communication between the goals will enable success or failure once there is sufficient information, eliminating excess computations.

In CONPRESS the *occurrences* predicate has been modified and is defined as a relation *occurrences(Subterm,Term, Pulses,H)*. *Pulses* is an output stream which will be used to pass on information about occurrences of *Subterm*. Each time an occurrence of *Subterm* is encountered a signal is sent along this communication channel. The fourth argument is used by *multiple_occurrences* to signal *occurrences* that it already has enough information to succeed so that *occurrences* can terminate.

The definitions of the occurrence relations in CONPRESS using communication are given by the program in Figure 12. The predicate *no_pulse* will fail as soon as one pulse is sent, causing the failure of *free_of*. Similarly, *single_occurrence* fails as soon as two pulses have been sent.

A halt message is signaled by the instantiation of the halting variable *H* by *multiple_pulse* as soon as two pulses have been sent. The signal is transmitted to *occurrences* through this variable.

```
free_of(Sub,Term) :-
    occurrences(Sub,Term,Pulses,_), no_pulse(Pulses?).
single_occurrence(Sub,Term) :-
    occurrences(Sub,Term,Pulses,_), single_pulse(Pulses?).
multiple_occurrence(Sub,Term) :-
    occurrences(Sub,Term,Pulses,H), multiple_pulse(Pulses?,H).

no_pulse([ ]).

single_pulse([pulse]).

multiple_pulse([pulse,pulse|Ps],halt).
```

FIGURE 12. Occurrence relations in CONPRESS.

4. PRESSED FLAT

The task of translating Concurrent PROLOG into Flat Concurrent PROLOG is different in essence from that of translating PROLOG into Concurrent PROLOG. Flat Concurrent PROLOG, as a powerful subset of Concurrent PROLOG, inherits most of the latter's programming techniques and styles. The problem then is the technical issue of reexpressing the logic of a program in terms of AND-parallelism. The resulting code simulates explicitly the operational behavior of Concurrent PROLOG when executing the original program. The techniques used for flattening Concurrent PROLOG are automated more easily (see [1] and [5]) than those for making PROLOG concurrent.

We present three different approaches used to flatten CONPRESS. The first is illustrated by the top level of FCPRESS. An extra level of reasoning has been added with the clauses.

```
solve_equations([EqVar|EqVars],Ss-Ss1) :-
  choose(EqVar,Method),
  solve_by_method(EqVar,Method?,Ss-Ss2?),
  solve_equations(EqVars?,Ss2-Ss1?).
solve_equations([],Ss-Ss).
```

The predicate *choose* calls a conjunction of goals derived from the predicates which appeared as guards in the top level of CONPRESS. It uses a *mutual exclusion variable* to convey to the *solve_by_method* predicate the name of a method which can be applied to the given equation. A *mutual exclusion variable* is a variable that several processes are concurrently trying to instantiate. Successful instantiation by one of the processes renders the continuation of the others irrelevant. The definition of *choose* is given in Figure 13. Note the use of the control primitive *otherwise* to achieve the effect of if-then-else.

Predicates called by *choose* of necessity must be altered. Each will contain a copy of the mutual exclusion variable, *Method*. Guards must be treated differently. Those that previously would succeed, causing the computation to commit, must now succeed and instantiate the mutual exclusion variable, resulting in a simulation of commitment. Guards that previously would fail now *quit*, that is, do not fail, but rather succeed without affecting the rest of the computation. The code is also altered to accept abort messages. When one of the goals called by *choose* succeeds in instantiating the mutual-exclusion variable, computation of all the others is aborted.

In Program 13 *choose* calls a condition for each of the four methods implemented. The first condition to succeed instantiates *Method* with the name of the method which is to be applied. An exception is homogenization, the condition for which is that a multiple-offenders set, *MOS*, has been identified in the equation. Application of the method itself involves a computation based on *MOS*. When this is the first condition to succeed, *Method* is instantiated to the multiple-offenders set.

In flattening CONPRESS, more structure has been added to the program. The computation of a goal is now clearly divided into two stages. In terms of equation solving, a clear distinction is made between choosing a method and applying it. This is a major change from the original style of PRESS code.

The first style of translation could be generalized into a method for flattening Concurrent PROLOG programs. For general applications there are several problems

```
solve_equations([EqVar|EqVars],Ss-Ss1) :-
    choose(EqVar,Method),
    solve_by_method(EqVar,Method?,Ss-Ss2?),
    solve_equations(EqVars?,Ss2-Ss1?).
solve_equations([ ],Ss-Ss).
choose(EqVar,Method) :-
    condition_factorize(EqVar,Method),
    condition_isolation(EqVar,Method),
    condition_polynomial(EqVar,Method),
    condition_homogenization(EqVar,Method).

condition_factorize((Lhs = 0,_),Method) :-
    mulbag(Lhs,Method).
condition_factorize(_,Method) :-
    otherwise | true.

condition_isolation((Eq,Var),Method) :-
    single_occ(Var,Eq,Method).

condition_polynomial((Lhs = Rhs,Var),Method) :-
    is_polynomial(Var,Lhr-Rhs,Method).

condition_homogenization(EqVar,Method) :-
    multiple_offender_set(EqVar,Method).
```

FIGURE 13. Choosing a method.

to be solved. The guards of a Concurrent PROLOG procedure are executed in separate environments, and different guards could instantiate a common variable to different values. Only when a guard commits are the local values of the variables broadcast to the rest of the computation. A generalization of this method would have to make separate copies of any common structures instantiated by the guards themselves to simulate this aspect of Concurrent PROLOG's execution of OR-parallelism. Copying structures is an expensive operation. Another problem to be solved is ensuring consistency between a copied variable and an instance of the original variable [1]. In our application the guards being flattened did not instantiate variables, and so these problems were avoided. A general solution to both these problems can be found in [5].

How (or even whether) to simulate commitment by aborting the computation of other guards is a question of efficiency, not of correctness. At one extreme, the program would be correct even if unnecessary computations were not aborted, but rather continued uselessly. At the other extreme, too many additional halting variables and too much insistence on handling abort messages could result in efforts being diverted from the subject of solving equations to that of simulating commitment. There is a tradeoff between clarity and efficiency. We must decide which parts of the code are to be haltable and which are not.

Two versions of the predicate *multiple_offender_set* are given as Figures 14 and 15. The predicate parses an equation to determine if it contains at least two offenders. This predicate is abortable. It will abort when some other goal instantiates *Method*. Figure 14 shows an example of more efficient code. When an abort message is received it is propagated to all the processes spawned by the top level process. This introduces an additional halting variable in each of these processes as well as code to

```
multiple_offender_set((Eq,Var),Method) :-
    parse(Eq,Var,Offs,Method),
    remove_duplicates(Offs?,Foffs,Method),
    mult_members(Foffs?,Method).
```

FIGURE 14. More abortable code.

```
multiple_offender_set((Eq,Var),Method) :-
    parse(Eq,Var,Offs),
    remove_duplicates(Offs?,Foffs),
    mult_members(Foffs?,Method).
```

FIGURE 15. Less abortable code.

handle abort messages (not shown here). In Figure 15 only the top level will respond to an abort message, leaving its spawned processes computing undisturbed without affecting the main computation.

The second approach to flattening concurrent PROLOG code is to compile the guarded commands into if-then-else structures. This is done by adding an extra argument to each guard predicate. This argument returns the result of the guard's execution—whether it succeeded or failed. This approach is a special case of the previous one when the procedure being flattened contains a single guard, which is not already a Flat Concurrent PROLOG kernel guard predicate.

The program in Figure 16 shows an example of a parsing predicate in FCPRESS. A term *Term* is parsed to check if it is a polynomial in *Var*. In CONPRESS *free_of* appeared in the guard of a clause (Figure 2). In FCPRESS *free_of* is called in conjunction with a continuation predicate which waits on the result of *free_of*, giving the effect of an if-then-else structure.

In this method, as in the previous, we have ensured that none of the predicates fail. They either succeed, returning a negative answer, or just quit. Goals in the bodies of these predicates have also been altered so as not to fail. One of these predicates could give a negative answer (or quit) as soon as one of the goals spawned by it did so. To gain efficiency we could add communication between brother goals to enable one to abort the others when possible. As before, this involves a tradeoff between clarity and efficiency. The price of increased efficiency is performing the explicit communication.

Two extra variables are used for communication in the programs in Figure 16. There is an instance of the mutual-exclusion variable *Method*, for communication with other predicates for choosing a method, and an additional halting variable *H* to halt all processes spawned by *is_polynomial* as soon as one of these discovers a nonpolynomial subterm of *Term*. An additional predicate is required by *is_polynomial* to merge the results of its subgoals. The predicate *poly_merge_results(M1,M2,Method,H)* defines *Method* to be *polynomial* if and only if both *M1* and *M2* are *polynomial*.

The third approach to translation is writing specialized definitions for those predicates which appear in the guards. A typical example is specializing *member*. The programs in Figures 17 and 18 respectively show the Flat Concurrent PROLOG

```
is_polynomial(Var,Term,Method) :-
    is_polynomial(Var,Term,Method,H).

is_polynomial(_,_,_,H) :-
    H = halt | true.
is_polynomial(_,_,M,halt) :-
    nonvar(M) | true.
is_polynomial(Var,Term1 + Term2,Method,H) :-
    is_polynomial(Var,Term1,M1,H),
    is_polynomial(Var,Term2,M2,H),
    poly_merge_results(M1,M2,Method,H).
is_polynomial(Var,Term1-Term2,Method,H) :-
    is_polynomial(Var,Term1,M1,H),
    is_polynomial(Var,Term2,M2,H),
    poly_merge_results(M1,M2,Method,H).
is_polynomial(Var,Term1* Term2,Method,H) :-
    is_polynomial(Var,Term1,M1,H),
    is_polynomial(Var,Term2,M2,H),
    poly_merge_results(M1,M2,Method,H).
is_polynomial(Var,Term^N,Method,H) :-
    N > = 0 | is_polynomial(Var,Term,Method,H).
is_polynomial(Var,Var,polynomial,H).
is_polynomial(Var,Term,M,H) :-
    otherwise | free_of(Var,Term,Ans), base_is_polynomial(Ans?,M).

base_is_polynomial(yes,polynomial).
base_is_polynomial(_,_) :-
    otherwise | true.

poly_merge_results(_,_,_,H) :-
    H = halt | true.
poly_merge_results(M1,M2,polynomial,H) :-
    M1 = polynomial, M2 = polynomial | true.
poly_merge_results(_,_,_,halt) :-
    otherwise | true.
```

FIGURE 16. Recognizing polynomials in FCPRESS.

and the Concurrent PROLOG versions of the predicate *remove_duplicates*, which filters duplicates from a stream. In the Concurrent PROLOG version *member* appears in the guard. In Figure 18 the predicate *remove_duplicates*_1 is a specialized version of *member*, which has been optimized to gain parallelism.

The third approach is a generalization of the second. Instead of modifying the guard to return *yes* or *no*, and writing code to handle each answer, the guard is

```
remove_duplicates(In,Out) :- remove_duplicates(In,[],Out).

remove_duplicates([],_,[]).
remove_duplicates([X|Xs],Acc,Out) :-
    member(X,Acc) | remove_duplicates(Xs?,Acc,Out).
remove_duplicates([X|Xs],Acc,[X|Out]):-
    otherwise | remove_duplicates(Xs?,[X|Acc],Out).
```

FIGURE 17. *remove_duplicates* in CONPRESS.

```
remove_duplicates(In,Out) :- remove_duplicates(In,[ ],Out-[ ]).
remove_duplicates([X|Xs],Acc,Out-Out1) :-
      remove_duplicates_1(X,Acc,Acc1,Out-Out2),
      remove_duplicates(Xs?,Acc1?,Out2-Out1).
remove_duplicates([ ],_,Out-Out).
remove_duplicates_1(A,[ ],[A],[A|Out]-Out).
remove_duplicates_1(A,[S|Ss],[S|Ss1],Out) :-
      S\ = A | remove_duplicates_1(A,Ss?,Ss1,Out).
remove_duplicates_1(A,[S|Ss],[S|Ss],Out-Out) :-
      S = A | true.
```

FIGURE 18. *remove_duplicates* with specialized member in FCPRESS.

modified to return a more general result, which reflects the behavior of the original program.

The question of using these and related approaches to translate Concurrent PROLOG programs to FCP are discussed in [5]. A general automated method based on partial evaluation is described there. The three approaches described in this section are handled in a uniform way.

5. PARALLEL SPEEDUP

It is a controversial issue in AI whether a machine which allows true parallel programming will make a significant difference in our ability to write intelligent programs [2]. This section describes the (theoretical) gain in speed obtained in the domain of equation solving by being allowed to perform calculations in parallel. We compare the performance of PRESS, CONPRESS, and FCPRESS in solving equations.

There are two levels where latent parallelism has been exploited in our translation. At the top level, the various entrance conditions can be checked in parallel in both CONPRESS and FCPRESS. Also independent equations, arising for example from factorization, can be solved in parallel. At a lower level some of the algorithms used in solving equations have been made parallel. Of particular note are the parsing and filtering algorithms.

We collected statistics on the number of logical inferences performed by the three programs in solving equations. Appropriate meta-interpreters were written. The process of collection favors the parallel languages Concurrent PROLOG and FCP. The communication costs of the parallelism are ignored, for example.

The equation we use as the basis for comparison is

$$2^{2x} - 5 \times 2^{x+1} + 16 = 0.$$

There are three stages to solving this equation. The first is homogenizing the equation to produce the equation

$$y^2 - 10y + 16 = 0, \qquad \text{where} \quad y = 2^x.$$

The second stage is solving the quadratic equation for y, producing two solutions. Solving the two resultant equations for x is the third stage.

The performance of the three programs during the three stages is summarized in Table 1. The column for PRESS is the total number of reductions (or logical inferences) needed to solve the equation. This number includes the "unnecessary

TABLE 1. Comparing logical inferences

Stage	PRESS	CONPRESS	FCPRESS
1	500	243/144/21	238/120/26
2	460	114/61/15	138/100/21
3	140	207/150/24	215/143/26
Total	900	564/355/56	591/365/67

calculations", for example testing the entrance conditions of methods which are not in fact applicable. The columns for CONPRESS and FCPRESS have the form $r_t/r_s/c$, where r_t is the total number of reductions performed, r_s is the total number of reductions in the successful computation path, and c is the number of cycles in the whole reduction, where a cycle is one reduction on all active processors.

The different figures for CONPRESS and FCPRESS represent different measures of potential speedup. The total number of reductions indicates how the program performs on a single processor. Parallelism is simulated by sharing among the processes equally. This type of speedup has been discussed in the context of cryptarithmetic puzzles by Kornfeld [7].

In trying to estimate how fast programs would run, the comparative speeds of the interpreters for the languages should be taken into consideration. C-PROLOG on a VAX, which was used for this research, runs at approximately 1000 Lips for PROLOG, 300 Lips for Concurrent PROLOG, and 500 Lips for FCP. When considering compilers, trying to come up with meaningful comparisons is even harder. A compiler for FCP now exists which runs at 7000 Lips, in comparison with PROLOG compilers which run in excess of 20,000 Lips.

The number of cycles is a measure of how fast a parallel program can perform with as many processors as needed, and ignoring communication time. An equation requires at least the number of cycles to be solved. In contrast to the number of reductions, the number of cycles in the total is less than the sum of the numbers of cycles in the three stages. This is because some of the calculation can be interleaved. For this example the interleaving is between 5 and 10%.

The number of cycles is less for CONPRESS than for FCPRESS. This is due to the difference between AND-parallelism and OR-parallelism in the way statistics are collected. In FCPRESS, which only has AND-parallelism, all reductions are counted with the exception of guard kernel predicates, which are considered part of unification. In CONPRESS, however, all guards are counted as one cycle, even if they are complex. To avoid the same problem with the reductions, we estimated the number of reductions separately.

The number of successful reductions is the basis of the estimate of the amount of parallelism possible at the lower level of equation solving. The measure is provided by the quotient of the number of reductions by the number of cycles. The table shows a factor of approximately 5 for each stage. This factor is also a lower bound on the number of processors needed to achieve maximum speedup.

The quotient of the total number of reductions by the number of cycles is another estimate of the number of processors needed for maximum speedup. The figure, approximately 10 from the table, is an average over the whole computation. The maximum number of processes actively reducing at one time was over 20.

TABLE 2. Speedup for different methods

Method	Speedup
Isolation	3.5
Polynomial	20
Homogenization	15

An interesting quotient is the number of reductions in PRESS by the number of cycles in FCPRESS. This estimates the actual gain in speed under best circumstances. The speedup for homogenization is 900/67, approximately 14. Table 2 compares the speedup for the different methods. These can be calculated from our example, since polynomial methods are used to solve the second stage, and isolation is used to solve the two equations from the third stage.

The different methods reflect different features. The principal reason why isolation has the lowest speedup is that it is the first method tested for in PRESS. Testing whether an equation can be factorized is trivial.

6. CONCLUSIONS

We were successful in translating a PROLOG program to Concurrent PROLOG, and further to Flat Concurrent PROLOG. The cleaner parts of PRESS corresponding to pure logic programs were the easiest to translate. Both PROLOG and Concurrent PROLOG are good for don't-care nondeterminism, with Concurrent PROLOG being more correct logically, insisting that all conditions be made explicit. PRESS code which relies heavily on don't-know nondeterminism is not translated efficiently to concurrent PROLOG, resorting occasionally to the simulation of OR-parallel PROLOG in the guard system. In such cases a new algorithm is desirable. When this was overlooked while translating PROLOG to Concurrent PROLOG, it had to be solved later when translating to Flat Concurrent PROLOG.

General translation of PROLOG programs into Concurrent PROLOG seems to require basic common sense and understanding of the semantics of the program. No technique emerged which was suitable as the basis for automatic translation. In contrast, the techniques used for translating Concurrent PROLOG programs into Flat Concurrent PROLOG represent a promising step towards general translating methods, which have been successfully developed in [5].

This work was supported by a Dov Biegun postdoctoral fellowship to the first author and a Wiezmann studentship to the second author. Helpful suggestions were made by Ehud Shapiro and anonymous referees. The use of computing facilities at the Wiezmann Institute, supported by a Digital Equipment External research grant, and at Case Institute of Technology, are gratefully acknowledged.

REFERENCES

1. Bloch, C. A., Source to Source Transformations of Logic Programs, M.Sc. Thesis, Tech. Report CS84-22, Dept. of Applied Mathematics, Weizmann Insittitue of Science, 1984.
2. Bobrow, D. G. and Hayes, P. J., Artificial Intelligence: Where Are We? *Artificial Intelligence* 25:375–415 (1985).

3. Bundy, A. and Silver, B., Homogenization: Preparing Equations for Change of Unknown, in: *Proceedings of IJCAI-81*, 1981, pp. 551–553.
4. Bundy, A. and Welham, B., Using Meta-Level Inference for Selective Application of Multiple Rewrite Rules in Algebraic Manipulation, *Artificial Intelligence* 16:189–212 (1981).
5. Codish, M., Automatic Translation of Concurrent Prolog to FCP, M.Sc. Thesis, Weizmann Institute of Science, 1985; to appear.
6. Hellerstein, L. and Shapiro, E. Y., Implementing Parallel Algorithms in Concurrent Prolog: The MAXFLOW Experience, in: *Proceedings of the International Symposium on Logic Programming*, Atlantic City, N.J., Feb. 1984.
7. Kronfeld, W., The Use of Parallelism to Implement a Heuristic Search, in: *Proceedings of IJCAI-81*, 1981, pp. 575–580.
8. Kowalski, R., *Logic for Problem Solving*, North Holland, 1979.
9. Mierowsky, C., Design and Implementation of Flat Concurrent Prolog, M.Sc. Thesis, Tech. Report CS84-21, Dept. of Applied Mathematics, Weizmann Institute of Science, 1984.
10. Shapiro, E. Y., A Subset of Concurrent Prolog and its Interpreter, Tech. Report CS83-06, Dept. of Applied Mathematics, Weizmann Institute of Science, 1983.
11. Shapiro, E. Y., Systolic Programming: A Paradigm of Parallel Processing, In: *Proceedings Fifth Generation Computing*, Japan, Nov. 1984.
12. Shapiro, E Y., Mierowsky, C., Taylor, S., Levy, J., et al., A Sequential Implementation of Flat Concurrent Prolog: The Full Description, Tech. Report, in preparation.
13. Sterling, L., Bundy, A., Byrd, L., O'Keefe, R., and Silver, B., Solving Symbolic Equations with PRESS, *Comput. Algebra LNCS* **144**:109–116 (1982).
14. Sterling, L. and Codish, M., PRESSing for Parallelism: A Prolog Program Made Concurrent, Tech. Report CS-85-12, Weizmann Institute of Science.