Converting AND-control to OR-control by program transformation

M.H. van Emden
University of Waterloo

P. Szeredi
Comp. Research & Innovation Centre

Research Report CS-87-21
March 1987

To appear in

"Foundations of Deductive Databases and Logic Programming"
Jack Minker (editor)
Morgan, Kaufmann Publishers, 1987.

Converting AND-control to OR-control by program transformation

M.H. van Emden
University of Waterloo
Waterloo, Canada

P. Szeredi
Computer Research and Innovation Centre (SZKI)
Budapest, Hungary

Abstract

We show how AND-control of logic programs can be transformed to OR-control by the well-known program transformation of unfolding followed by folding. We demonstrate the technique by taking as starting point the logic specification of a dataflow network, which requires complex AND-control, so that it cannot be run by standard Prolog. After transformation we supply the required OR-control, resulting in a program that can be run by a standard Prolog interpreter.

Introduction

The success of logic programming depends on how often it happens that a definition in logic can be transformed easily to an executable Prolog program. How easy it is to transform (or whether transformation has to done at all), depends on the power of Prolog's control mechanisms.

The control mechanisms of logic programming belong to two main categories. Goal selection determines the derivation tree when the query and the program are given. To find a successful derivation within this tree is the problem of *OR-control.* To select goals in such a way that the derivation tree as a whole is favourable is the problem of *AND-control.*

Prolog's AND-control is characterized by the fixed goal ordering where the leftmost goal is always selected. Thus, when the goals are $G_1, G_2, \ldots, G_n$, none of $G_2, \ldots, G_n$ will get any attention until G_1 is completely solved. This is not effective for an important

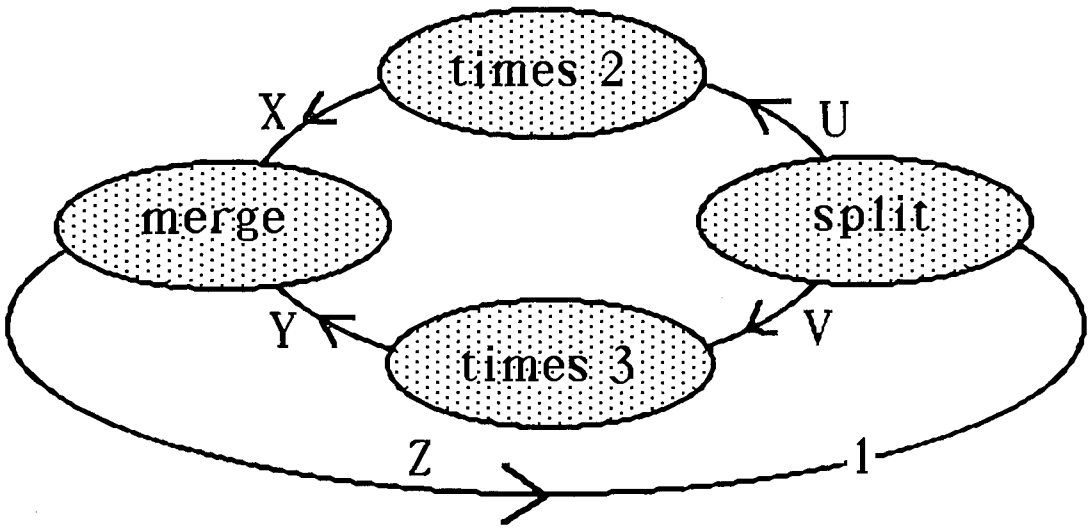

Figure 1: Dataflow network for a simplified version of Hamming's problem

class of logic definitions, where co-routining between goals is required: a goal typically is not solved to completion, its execution being interrupted by work on other goals. Several variants of Prolog provide AND-control which is adequate for co-routining [4,5,11].

Co-routining can be viewed as a way of running conceptually parallel cooperating processes on a single processor. Thus, it is not surprising that several variants of Prolog [2,3,9,12,13] designed for parallel execution of goals have more sophisticated AND-control than Prolog itself. In this paper we consider an alternative to providing such more sophisticated control, namely to transform a program requiring co-routining AND control to one that can be run by plain Prolog. The elimination of the need for coroutining is obtained at the cost of additional OR-control.

In this paper we demonstrate a transformation method applicable to logic definitions representing dataflow programs where each node in the dataflow network represents a perpetual cyclic process. This class is not executable by Prolog because co-routining AND-control is required.

Translation of network specifications

The type of network specification we translate here was proposed in [10]. Consider the dataflow network in Figure 1, which solves a version of Hamming's problem. This simplified version produces the sequence of integers containing no prime factors other than 2 and 3. Each node represents a processor, each arc a communication channel. Each processor executes an cyclical computation and is activated as soon as its input channels contain enough data for one cycle of its computation.

The computation of the `merge` node is as follows. In case the first numbers in each of

the input channels are equal, both are removed and one is written in the output channel. If the two first numbers are different, the smallest one is removed and output. `times2` outputs each input number multiplied by 2; analogously for `times3`. `split` outputs a copy of each input number. The diagram in Figure 1 shows the initial state of the network, where all channels are empty except for the one between `merge` and `split`, which contains a 1.

According to [6,7], a dataflow network is represented in logic in two parts. The logic program contains definitions of the individual nodes in the network as relations between lists of data items. Thus the program contains no information about how the nodes are connected; this is done in the goal statement.

Here follow the definitions of the nodes in Figure 1:

```
merge(A.X B.Y A.Z) if lt(A B) merge(X B.Y Z);
merge(A.X B.Y B.Z) if lt(B A) merge(A.X Y Z);
merge(A.X A.Y A.Z) if merge(X Y Z);

times2(A.U B.X) if prod(A 2 B) times2(U X);

times3(A.V B.Y) if prod(A 3 B) times3(V Y);

split(A.Z A.U A.V) if split(Z U V);
```

The network is specified by the goal statement:

```
? merge(X Y Z) times2(U X) times3(V Y) split(1.Z U V)
```

Note how the shared variables represent the communication channels between the processors of the network. In cases where two goals have as argument just the shared variable itself, the corresponding channel is empty in the initial state. This is the case everywhere except for the channel connecting `merge` (where the argument is `Z`) and `split` (where the argument is `1.Z`). The difference (in the sense of "difference lists") between these arguments is `1`, just the content of the channel represented by the arguments.

Of course, if we ask the question

```
? merge(X Y Z) times2(1.Z X) times3(1.Z Y)
```

obtained from the previous one by using the fact that, by the meaning of `split`, `U` and `V` both equal `1.Z`, then we don't need the definition of `split` at all *and* we get a shorter question. This shorter question, however, has no corresponding dataflow diagram. This simplification is a symptom of the greater power of logic programming compared to dataflow diagrams. In the sequel we continue with the dataflow diagram because we need an instructive example, rather than a most concise statement of Hamming's problem.

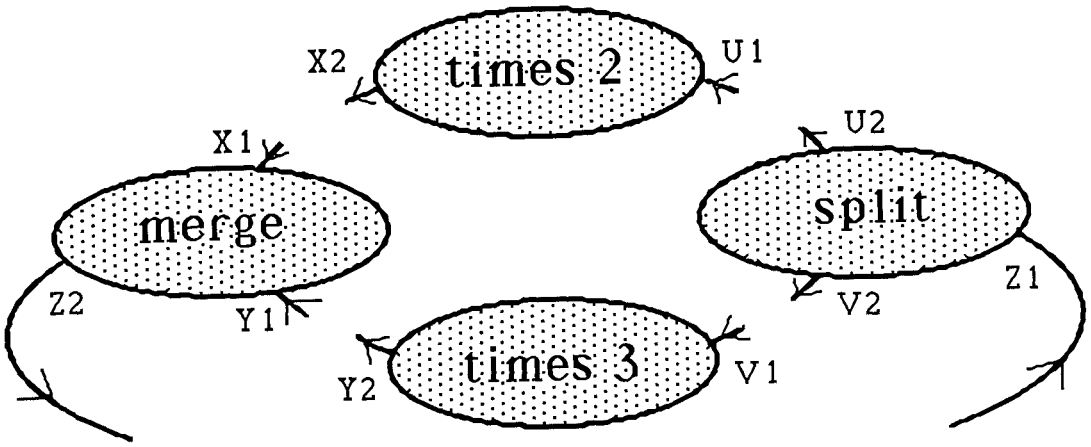

Figure 2: The unconnected conjunction of nodes

A variant of the dataflow network representation

For the purpose of the program transformation it is convenient to modify the dataflow network representation described in the previous section, where the network was represented directly by a question.

In this section we describe a network representation consisting of two parts. In the first part, consisting of an equivalence, we represent a conjunction of all nodes, without specifying any connections.

For our example this equivalence is:

```
conj(X1 Y1 Z2 U1 X2 V1 Y2 Z1 U2 V2) iff
merge(X1 Y1 Z2) & times2(U1 X2) & times3(V1 Y2) & split(Z1 U2 V2)
```

Although this is not a clause, we have followed here the clausal convention of implicitly universally quantifying all variables. In the right-hand side we have named the variables in a way indicating how they are going to be "connected": `X1` and `X2` are now respectively the input and output ports of the same channel that was represented by `X` before; similarly for the other variables and channels. See Figure 2.

More helpfully, `X1` and `X2` can be regarded as lists: `X1` is then the list of all items passing through the output port of X, while `X2` is the list of all items passing through its input port. It follows that `X2` must be a posterior sublist of `X1` and that the (possibly empty) prefix of items in `X1` which is not in `X2` consists of the contents of the channel X. In other words, `X1` and `X2` are the two components of the *difference list* representation of channel X.

In the second stage, we complete the network specification by a question. It is here that we use the difference list representation to specify the contents of each channel.

```
? conj(X  Y  Z  U  X  V  Y  1.Z  U  V)
```

Although this representation will support our program transformation, it has the disadvantage that `conj` has a long list of arguments that hides a useful structure: the arguments can be partitioned into groups representing ports of the same node. We will therefore continue with a modification of the representation used before.

This modification is obtained by changing the predicates for the node computations to become, formally at least, unary, by the use of a functor to package the input and output streams into a single term.

```
merge(m(A.X B.Y A.Z)) if lt(A B) merge(m(X B.Y Z));
merge(m(A.X B.Y B.Z)) if lt(B A) merge(m(A.X Y Z));
merge(m(A.X A.Y A.Z)) if merge(m(X Y Z));

times2(t2(A.U B.X)) if prod(A 2 B) times2(t2(U X));

times3(t3(A.V B.Y)) if prod(A 3 B) times3(t3(V Y));

split(s(A.Z A.U A.V)) if split(s(Z U V));
```

Now that each of the node relations has one argument, the definition of `conj` is simplified:

```
conj(M T2 T3 S) iff merge(M) & times2(T2) & times3(T3) & split(S);
```

Finally, the network connections are specified by the question

```
? conj(m(X Y Z) t2(U X) t3(V Y) s(1.Z U V))
```

Deriving the program

In this section we show how to apply the unfold/fold transformation [1] to obtain a logic program requiring OR-control where the original network specification required AND-control.

The key predicate in the derived program is `conj`. From the definition we use only the *if* part:

```
conj(M T2 T3 S) if merge(M) times2(T2) times3(T3) split(S);
```

Let us combine this definition with that of `merge`. As the first step, we obtain from the above implication the instance:

```
conj(m(A.X B.Y A.Z) T2 T3 S)
    if merge(m(A.X B.Y A.Z)) times2(T2) times3(T3) split(S);
```

We use the definition of merge to obtain by resolution ("unfolding", or "partial application"):

```
conj(m(A.X B.Y A.Z) T2 T3 S)
    if lt(A B) merge(m(X B.Y Z)) times2(T2) times3(T3) split(S);
```

When we now use the *only if* part of the definition of conj on the right-hand side, we obtain (by "folding"):

```
conj(m(A.X B.Y A.Z) T2 T3 S)
    if lt(A B) conj(m(X B.Y Z) T2 T3 S);
```

In a similar way, each of the clauses defining a node relation can be combined with the definition of conj to yield the following set of clauses:

```
conj(m(A.X B.Y A.Z) T2 T3 S)
    if lt(A B) conj(m(X B.Y Z) T2 T3 S);
conj(m(A.X B.Y B.Z) T2 T3 S)
    if gt(A B) conj(m(A.X Y Z) T2 T3 S);
conj(m(A.X A.Y A.Z) T2 T3 S)
    if conj(m(X Y Z) T2 T3 S);
conj(M t2(A.U B.X) T3 S)
    if prod(A 2 B) conj(M t2(U X) T3 S);
conj(M T2 t3(A.U B.X) S)
    if prod(A 3 B) conj(M T2 t3(U X) S);
conj(M T2 T3 s(A.Z A.U A.V))
    if conj(M T2 T3 s(Z U V));
```

OR-control of the derived program

The derived program is activated by the question:

```
? conj(m(X Y Z) t2(U X) t3(V Y) s(1.Z U V))
```

This call, and calls generated later by this call, are such that all clauses for conj match. The required control is to select a suitable subset of clauses; hence OR-control.

Just as with AND-control for the specification, the control is derived from the computation rule for dataflow networks:

Any node is allowed to execute one cycle whenever it has enough data in its input channels to execute this cycle.

In this example, the computation rule translates to the condition that a clause should only be selected when certain variables have become bound. A straightforward approach is the following:

```
conj(m(A.X B.Y A.Z) T2 T3 S)
    if not(var(A)) not(var(B)) lt(A B) conj(m(X B.Y Z) T2 T3 S);
conj(m(A.X B.Y B.Z) T2 T3 S)
    if not(var(A)) not(var(B)) gt(A B) conj(m(A.X Y Z) T2 T3 S);
conj(m(A.X A.Y A.Z) T2 T3 S)
    if not(var(A)) conj(m(X Y Z) T2 T3 S);
conj(M t2(A.U B.X) T3 S)
    if not(var(A)) prod(A 2 B) conj(M t2(U X) T3 S);
conj(M T2 t3(A.V B.Y) S)
    if not(var(A)) prod(A 3 B) conj(M T2 t3(V Y) S);
conj(M T2 T3 s(A.Z A.U A.V))
    if not(var(A)) conj(M T2 T3 s(Z U V));
```

Each condition of the derived program has received its OR-control in the form of "guards" on the conditions. In this way we intend to realize the requirement of the dataflow rule of computation that only those processes are allowed to be activated that have the required data in their input channels.

This example shows that just mechanically inserting guards of the form `not(var(...))` does not always have the desired effect. In fact, the above program does not solve Hamming's problem, but generates instead an infinite sequence of 2's. Why this happens may be seen as follows. Suppose the state of the dataflow network is as represented in the question

```
? conj(m(2.X Y Z) t2(U X) t3(1.V Y) s(Z U V))
```

Now the third clause

```
conj(m(A.X' A.Y' A.Z') T2 T3 S)
    if not(var(A)) conj(m(X' Y' Z') T2 T3 S);
```

will be used. But this clause represents an action by the merge-node, which should only be allowed when *both* input channels are non-empty. Use of the third clause results in the question

```
? conj(m(X Y' Z') t2(U X) t3(1.V 2.Y') s(2.Z' U V))
```

Not only has a 2 appeared in the output channel of merge, but the equivalent of pushing it into the empty input channel has also occurred.

The third clause should not be obtained by merely inserting the guard. Instead, it should be recognized that there is an implicit test for equality (the first two occurrences of A) and that this test should come *after* a guard ensuring that the input channels are nonempty, like this:

```
conj(m(A.X B.Y A.Z) T2 T3 S)
    if not(var(A)) not(var(B)) eq(A B) conj(m(X Y Z) T2 T3 S);
```

An alternative route to the transformed program

The transformed program can also be obtained by considering the *state* of the dataflow network and regarding a computation by the network as a sequence of state transitions. And of course, we have to formulate a logic program having as property that its execution by SLD-resolution somehow mimicks the desired sequence of state transitions.

We first explain the general principle and then apply it to the dataflow situation. Consider a binary relation between states such that `trans(State1 State2)` iff the transition from `State1` to `State2` is possible.

Let us call a *computation* a sequence of states such that successive elements in the sequence are in the transition relation. If there is a last state in the computation, then it has to be a specially designated *halt state*. Let `comp(State1 State2)` be true iff there is a computation with `State1` as first element and `State2` as last element. The relation `comp` is then defined by:

```
comp(State State) if halt(State);
comp(State1 State2) if trans(State1 State) comp(State State2);
```

As we are interested in our data-flow example in computations without a halt state, the first clause is irrelevant. Moreover, in the second clause the second argument of `comp` becomes irrelevant. Hence we arrive at the predicate `start` which asserts of a state that it is the start state of a possibly infinite computation.

```
start(State1) if trans(State1 State2) start(State2);
```

The state consists of the aggregation of the channel contents, represented as lists. We implement this aggregation by representing the state as a term with `state` as functor and with clusters of channels as arguments. Let us take as example one of the transitions possible when the `merge` node has nonempty input channels:

```
trans(state(m(A.X B.Y A.Z) T2 T3 S) state(m(X B.Y Z) T2 T3 S))
     if lt(A B);
```

When we do a resolution ("partial application") with the clause for `start`, we get:

```
start(state(m(A.X B.Y A.Z) T2 T3 S))
  if lt(A B) start(state(m(X B.Y Z) T2 T3 S));
```

Note that this is substantially the same as one of the clauses of the program obtained by the unfold/fold transformation and that the other clauses can be obtained in the same way from operational reasoning about state transitions.

Of course, we have to add the same OR-control as we did to the result of the unfold/fold transformation. We use the operational reasoning about state-transitions in the dataflow model to finish the incomplete control of the last version of the result of the unfold/fold transformation.

Opportunities for parallel computation arise in dataflow networks when more than one node has sufficient input. In our example it often happens that both multiplication nodes can operate in parallel. In the transformed program this shows by more than one clause having its guard succeed. From the dataflow rule of computation it is apparent that one can commit on a choice to any clause for which the guard succeeds. Thus, the result of our transformation of dataflow programs can be made to exhibit the committed-choice nondeterminism typical of variants of Prolog for parallelism.

Taking this consideration into account, we find that we can place "cut" operators, to obtain as our final program the one shown below. Of course, a Prolog hacker would have seen the opportunity for cuts right away; we prefer to conclude their permissibility from properties of the dataflow computation model.

```
conj(m(A.X B.Y A.Z) T2 T3 S)
    if not(var(A)) not(var(B)) lt(A B) cut conj(m(X B.Y Z) T2 T3 S);
conj(m(A.X B.Y B.Z) T2 T3 S)
    if not(var(A)) not(var(B)) gt(A B) cut conj(m(A.X Y Z) T2 T3 S);
conj(m(A.X B.Y A.Z) T2 T3 S)
    if not(var(A)) not(var(B)) eq(A B) cut conj(m(X Y Z) T2 T3 S);
conj(M t2(A.U B.X) T3 S)
    if not(var(A)) cut prod(A 2 B) conj(M t2(U X) T3 S);
conj(M T2 t3(A.V B.Y) S)
    if not(var(A)) cut prod(A 3 B) conj(M T2 t3(V Y) S);
conj(M T2 T3 s(A.Z A.U A.V))
    if not(var(A)) cut conj(M T2 T3 s(Z U V));
```

Conclusions

We presented two ways of deriving Prolog-executable programs from logic specifications of dataflow diagrams. The first used the well-known unfold/fold transformation; it depended

exclusively on declarative concepts. The second was based entirely on machine-oriented concepts: transitions allowed in the dataflow model of computation. It is surprising that a purely declarative approach can result in essentially the same Prolog program as a purely operational approach.

It is not surprising of course to find *some* connection with operational concepts. After all, the unfold transformation is a top-down computation step of a Prolog interpreter. But we found a close relation not directly to the action of the Prolog interpreter, but the dataflow model of computation.

We have shown an example of conversion of AND-control to OR-control. AND-control is needed to exploit the potential for parallelism in the evaluation of the conditions of a clause. Our work raises the question: have we also shown how to convert opportunities for AND-parallelism to opportunities for OR-parallelism?

The demonstration of this paper clearly applies to all static dataflow networks consisting of nodes executing nonterminating cyclic computations. An attractive area for further investigation is to expand our technique to a wider class of logic specifications.

Acknowledgments

We owe a debt of gratitude to Steve Gregory and two anonymous referees for their penetrating remarks, which made it possible for us to remedy some of the previous version's shortcomings. Our thanks also to Mantis H.M. Cheng for pointing out that the naïve OR-control of the derived program is wrong, and why. Last, but not least, we gratefully acknowledge the role of the Digital Equipment Corporation of Canada and the National Science and Engineering Research Council of Canada in contributing to the research facilities used in the work reported here.

References

[1] Burstall, R. and Darlington, J. [1977] A transformation system for developing recursive programs. *Journal of the ACM*, vol. 24, 44-67.

[2] Clark, K.L. and Gregory, S. [1981] A relational language for parallel programming. *Proceedings of the ACM conference on functional languages and computer architecture.*

[3] Clark, K.L. and Gregory, S. [1986] Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, vol. 8, 1-49.

[4] Clark, K.L. and McCabe, F.G. [1979] The control facilities of IC-Prolog. In *Expert Systems*, D. Michie (ed.), Edinburgh University Press.

[5] Colmerauer, A. [1982] Prolog II reference manual and theoretical model. *Internal report, Groupe Intelligence Artificielle, Université d'Aix Marseille II.*

[6] Emden, M.H. van, and Lucena, G.J. de [1979] Predicate logic as a language for parallel programming. *Research Report CS-79-15, Computer Science Department, University of Waterloo.*

[7] Emden, M.H. van, and Lucena, G.J. de [1982] Predicate logic as a language for parallel programming. In *Logic Programming*, K.L. Clark and S.A. Tärnlund (eds.), Academic Press, 189-198.

[8] Gregory, S. [1980] Towards the compilation of annotated logic programs. *Research Report DoC 80/16, Department of Computing, Imperial College*, London.

[9] Gregory, S. [1987] *Parallel Logic Programming in PARLOG.* Addison-Wesley.

[10] Kahn, G. and McQueen, D.B. [1977] Coroutines and networks of parallel processes. *Proceedings IFIP 1977.*

[11] Naish, L. [1983] The MU-Prolog 3.2 reference manual. *Technical Report 85/11, Department of Computer Science, University of Melbourne.*

[12] Shapiro, E.Y. [1983] A subset of concurrent Prolog and its interpreter. *ICOT Technical Report TR-003.*

[13] Ueda, K. [1985] Guarded Horn Clauses. *ICOT Technical report TR-103.*