Printing Requisition/Graphic Services

14141

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number, to extension 3451.**

TITLE OR DESCRIPTION: CS-87-64 Complete Sets of Frontiers in Logic-based Program Transformation

DATE REQUISITIONED: April 18/88

DATE REQUIRED: ASAP

ACCOUNT NO.: 1,2,6,6,6,1,4,4,1

REQUISITIONER – PRINT: M. van Emden

PHONE: 2192

SIGNING AUTHORITY: M.H. van Emden

MAILING INFO – NAME: S. DeAngelis

DEPT.: C.S.

BLDG. & ROOM NO.: DC 2314

☑ DELIVER ☐ PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 15

NUMBER OF COPIES: 50

TYPE OF PAPER STOCK: ☑ BOND ☐ NCR ____ PT. ☑ COVER ☐ BRISTOL ☑ SUPPLIED ☐ ____

PAPER SIZE: ☑ 8½ x 11 ☐ 8½ x 14 ☐ 11 x 17 ☐ ____

PAPER COLOUR: ☑ WHITE ☐ ____

INK: ☑ BLACK ☐ ____

PRINTING: ☐ 1 SIDE ____ PGS. ☑ 2 SIDES ____ PGS.

NUMBERING: FROM ____ TO ____

BINDING/FINISHING: ☑ COLLATING ☑ STAPLING ☐ ____ HOLE PUNCHED ☐ PLASTIC RING

3 down left side

FOLDING/PADDING

CUTTING SIZE

Special Instructions

With fronts + backs enclosed

NEGATIVES	QUANTITY	OPER. NO.	TIME	LABOUR CODE
FLM				C01
FLM				C01
FLM				C01
FLM				C01
FLM				C01

PMT				
PMT				C01
PMT				C01
PMT				C01

PLATES				
PLT				P01
PLT				P01
PLT				P01

STOCK				
				001
				001
				001
				001

BINDERY				
RNG				B01
RNG				B01
RNG				B01
MIS00000				B01

COPY CENTRE	OPER. NO.	BLDG.	MACH. NO.

DESIGN & PASTE-UP	OPER. NO.	TIME	LABOUR CODE
			D01
			D01
			D01

TYPESETTING	QUANTITY			
PAP00000				T01
PAP00000				T01
PAP00000				T01

PROOF				
PRF				
PRF				
PRF				

OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL ☐ FEDERAL ☐

GRAPHIC SERV. OCT. 85 482-2

Complete Sets of Frontiers

in

Logic-based Program

Transformation

Research Report CS-87-64

December 1987

Complete Sets of Frontiers in Logic-based Program Transformation

Mantis H.M. Cheng* Maarten H. van Emden† Paul A. Strooper†

Abstract

Logic-based program transformation allows us to convert logic programs that are obviously correct (and as such can be considered as specifications), but often inefficient, into more efficient ones in such a way that correctness of answers is preserved. However, existing methods either do not guarantee completeness, or else introduce redundancy. We present a method for using sets of "partial derivation trees" to obtain a specialized version of a program for a particular query, guaranteeing completeness and avoiding redundancy. An example of specializing a meta interpreter for a particular program is given.

1 Introduction

The "fold" and "unfold" transformations were introduced by Burstall and Darlington [1] to make functional programs more efficient. What they had in mind was a system of "computer-aided programming" where a specification in functional form is transformed by means of a computer program into a functional program of acceptable efficiency. Independently of each other, Clark [2] and Hogger [5] have shown that these transformations also apply to logic programs. They pointed out that in this context they have an additional advantage: the transformed version is a logical consequence of the original. As a result, any answer obtainable from the transformed program is also an answer obtainable from the specification. The properties of logical consequence guarantee correctness with respect to the specification. But this correctness is only what is usually called *partial* correctness: it is not always clear whether all answers obtainable from the specification can also be obtained from the result of the transformation. When this is the case, we say that the transformed program is *complete* with respect to the specification.

In this paper we are only concerned with the "unfold" transformation, also called "partial evaluation" or "symbolic execution". As such, it appeals more directly to programming intuition: the transformed program is the result of replacing procedure calls by the appropriately instantiated bodies of their definitions. This sometimes allows more efficient execution of the transformed program.

*Dept. of Computer Science, University of Waterloo, Waterloo Ont., Canada N2L 3G1
†Dept. of Computer Science, University of Victoria, Victoria B.C., Canada V8W 2Y2

But partial evaluation is only useful if we know it yields a complete result, otherwise we cannot discard the inefficient specification. Tamaki and Sato [12] show that certain transformations on clauses, including unfolding, preserve completeness. To make the results of Tamaki and Sato applicable to our purpose, it is necessary to eliminate from the program those clauses which are redundant with respect to the desired query. Although this can be achieved by means of a dependency analysis done after the transformations, we found it possible to combine the unfold transformation with a dependency analysis to achieve the desired result in a single operation, which is to build what we call a "complete set of frontiers".

In doing so, we take as starting point the idea of Vasey [15], who shows that the result of unfolding can be regarded as a "qualified answer". We obtain the desired completeness by considering frontiers of "conditional answers" (our preferred terminology for Vasey's qualified answers) in a suitable derivation tree. As a first approximation, our completeness is based on the fact that in the derivation tree no computation can "escape" the frontier. This observation is only an approximation, because in all but a few trivial cases one has to consider more than a single derivation tree with frontier. We characterize when enough frontiers have been found. These are then a set of clauses obtained by unfolding which is complete with respect to the specification.

Lloyd and Shepherdson [8] independently found a result similar to our theorem on complete sets of frontiers.

2 The frontier theorem

From now on, we assume without loss of generality that each query posed to a logic program consists of a single goal. If this were not the case, and we would have a query of the form $?G_1, \ldots, G_n$ for $n > 1$, we can add the clause $G(X_1, \ldots, X_m) \leftarrow G_1, \ldots, G_n$ to the program and replace the query by $?G(X_1, \ldots, X_m)$. Here G stands for any predicate symbol not appearing elsewhere in the program, and $X_1, \ldots, X_m$ are the variables occurring in $G_1, \ldots, G_n$.

A *derivation tree* for a query Q is a tree with Q at the root. Each node in the tree is a query consisting of a conjunction of atomic formulae, called the *goals* of the query. If the query is not empty, it has a *selected goal.* A node N, consisting of the query $?G_1, \ldots, G_n$ with selected goal G_i has a child for each clause whose head unifies with G_i. If G_i unifies with the head of the clause $A \leftarrow B_1, \ldots, B_m$ with most general unifier θ, then the corresponding child consists of the query $?(G_1, \ldots, G_{i-1}, B_1, \ldots, B_m, G_{i+1}, \ldots, G_n)\theta$.

A *derivation* is a path starting from the root in the derivation tree which is either infinite or ends in a leaf node of the derivation tree. A *successful* derivation is a derivation ending in an empty query, denoted by $\Box$. A *failed* derivation is a derivation ending in a non-empty query with no children. It follows from the definition that this only happens if the selected goal does not unify with the head of any clause of the program. A *partial derivation* is a path starting from the root in the derivation tree. It can end in a non-leaf node of the derivation tree.

It should be noted that Prolog finds its answers by constructing a particular type of derivation tree, the *Prolog derivation tree.* This derivation tree is obtained by always

selecting the leftmost goal in each query, and by ordering the children of a node in the same way as the corresponding clauses in the program. A successful response by Prolog corresponds to a successful derivation in the Prolog derivation tree, the answer substitution being the composition of all substitutions made in that derivation.

Starting with a program P, if there exists a successful derivation of the query $?G$, with θ being the composition of all substitutions in the derivation, then we say that $G\theta$ is an *(unconditional) answer* to the query. In this case the answer is a logical consequence of the program P, which we write as:

$$P \models \forall G\theta$$

where the universal quantification is over all variables in $G\theta$.

Suppose, that starting from the query $?G$, we have derived a non-empty query $?G_1, \ldots, G_n$, with θ being the composition of all substitutions so far. Then the clause $(G \leftarrow G_1, \ldots, G_n)\theta$ is a *conditional answer* to the query. Again, as can easily be shown from the soundness of resolution, we have:

$$P \models \forall (G \leftarrow G_1, \ldots, G_n)\theta$$

where the universal quantification is over all variables occurring in the clause $(G \leftarrow G_1, \ldots, G_n)\theta$.

Definition 1 (Partial derivation tree) *A partial derivation tree is a finite initial subtree of a derivation tree in which it is possible for non-empty leaf queries not to have a selected goal.*

In a partial derivation tree there are three types of leaves: empty queries, failed queries (these have a selected goal, but no children nodes) and non-empty queries with no selected goal. For each empty query, there is a corresponding unconditional answer. Each non-empty query with no selected goal has a corresponding conditional answer.

For example, consider the following program, defining the membership relation using the append relation:

```
% member( E, L ): element E is a member of list L.
% example: member( 3, [1,2,3,4,5] )
member( E, L ) <- append( U, [E|V], L );

% append( U, V, W ): list W is list V appended to list U.
% example: append( [1,2], [3,4,5], [1,2,3,4,5] )
append( [], V, V );
append( [X|U], V, [X|W] ) <- append( U, V, W );
```

A partial derivation tree for the query `?member(X,Y)` is shown in Figure 1 (the selected goals are underlined).

Definition 2 (Frontier) *The frontier of a partial derivation tree is the set of all unconditional and conditional answers corresponding to the leaf nodes.*

In the above example, the frontier consists of the clauses

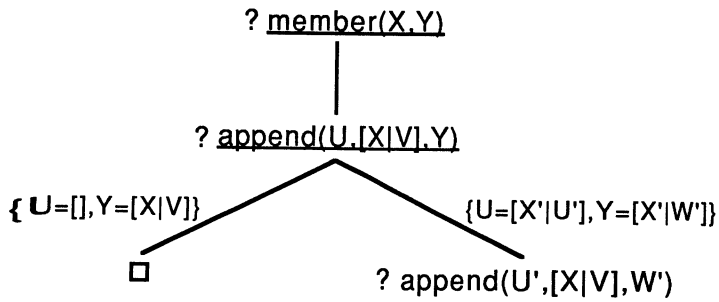

Figure 1: Partial derivation tree for member.

```
member( X,  [X|V] );
member( X,  [X'|W'] ) <- append( U', [X|V], W' );
```

where the first clause is the unconditional answer corresponding to the left branch of the partial derivation tree. The second clause is the conditional answer corresponding to the right branch.

The frontier of a partial derivation tree consists of clauses, and as such, we can regard it as a logic program. We would like to replace our original program by a frontier, but if we do that, the resulting program will not be complete. To obtain completeness, we have to consider a suitable set of frontiers, giving rise to a logic program by including the clauses in all of the frontiers. We formalize this idea by defining a *complete set of frontiers.*

In the following discussion, $L(F)$ denotes the set of goals appearing in the bodies of the clauses of a frontier F. Similarly, for a set of frontiers S, $L(S)$ denotes the union of the $L(F)$ for every frontier F in S. For a frontier F, $R(F)$ denotes the root of the partial derivation tree having F as a frontier. Finally, the goal G_1 is an *instance* of the goal G_2, if there exists a substitution θ such that $G_1 = G_2\theta$.

Definition 3 (Complete set of frontiers) *A set of frontiers S for a query Q is complete iff the following three conditions hold:*

1. *each frontier F in S is non-trivial, that is, at least the root itself has a selected goal.*
2. *there is a frontier F in S such that Q is an instance of $R(F)$.*
3. *for each goal G in $L(S)$, there is a frontier F in S such that G is an instance of $R(F)$.*

The above frontier by itself is not complete. The goal `append(U',[X|V],W')` is not an instance of the root of the partial derivation tree. If we add the frontier obtained by unfolding this goal once, which is

```
append( [], [X|V], [X|V] );
append( [X'|U'], [X|V], [X'|W'] ) <- append( U', [X|V], W' );
```

then we obtain a complete set of frontiers. Note that this is a version of the **append** program specialized for the purpose of the `member` program.

Any program P has at least one complete set of frontiers. For example, if we construct a frontier for the most general form of every predicate symbol appearing in P, then we have a complete set of frontiers (note that a frontier can be empty, which takes care of undefined calls in P). In fact, if we consider the frontiers in which we only select the root and do not select goals for any of the queries appearing at depth 1 in the partial derivation tree, then we obtain our original program as a complete set of frontiers.

The complete sets of frontiers given above are not very useful. The following algorithm suggests a more general way to obtain a complete set of frontiers S for a query Q and a program P.

1. Start with the frontier of any partial derivation tree for the query Q as the only element in S.

2. Select any goal, G, occurring in $L(S)$, but which is not an instance of $R(F)$ for any frontier F in S. Add the frontier of any partial derivation tree for G to the set, after removing any frontier F for which $R(F)$ is an instance of G (since the answers in such a frontier will also be included in the frontier of G).

3. If no such goal exists, then S is a complete set of frontiers for Q in P.

Although we can select any goal which is not an instance of a derivation tree already constructed, it is often more efficient to choose the *most general form* of all the occurrences of any such goals with the same predicate symbol. The intuition behind a complete set of frontiers is that they are expanded versions of the original program specialized to the query. The following result formalizes this idea.

Lemma 1 *Let S be a complete set of frontiers for a query Q and a program P. If the goal G is an instance of $R(F)$ for a frontier F in S, and if there is a successful derivation D of G in P, then there is a successful derivation D' of G in S. Moreover, if θ_1 and θ_2 are the compositions of all substitutions in D and D' respectively, then $G\theta_1$ is an instance of $G\theta_2$.*

A proof of this lemma is presented in the appendix.

Theorem 1 (Frontier Theorem) *If P is a program, Q a query, and S a complete set of frontiers for Q, then*

$$[Q] \cap \textit{success set of } P = [Q] \cap \textit{success set of } S$$

where $[Q]$ is the set of all ground instances of Q.

Proof By the correctness of SLD-resolution we know that

$$\text{success set of } P \supseteq \text{success set of } S$$

and hence we also have

$$[Q] \cap \text{success set of } P \supseteq [Q] \cap \text{success set of } S.$$

Conversely, if $G \in [Q] \cap$ success set of P, then $G \in [Q]$ and by the definition of a complete set of frontiers we know that G is an instance of $R(F)$ for a frontier F in S. Thus we can apply the previous lemma (G is a ground goal, so the substitutions that take place during the derivation do not concern us here) to conclude that $G \in$ success set of S, and hence

$$[Q] \cap \text{success set of } P \subseteq [Q] \cap \text{success set of } S$$

and the proof is completed. ■

It should be noted that the frontier theorem is valid for frontiers of *any* derivation tree, not just the Prolog derivation tree. In this way, selection methods other than Prolog's can be "compiled into" the frontier.

In the next section, we give a detailed example of how the frontier theorem can be used. To illustrate the ideas involved we give a small example here, similar to one in [3]. Consider the following program to transpose a matrix:

```
% A matrix is represented as a list of rows, for example
% [[1,2,3],[4,5,6]].
% transpose( M1, M2 ): matrix M2 is the transpose of matrix M1.
% Example: transpose( [[1,2,3],[4,5,6]], [[1,4],[2,5],[3,6]] )
transpose( M, [] ) <- nullrows( M );
transpose( M, [R|Rs] ) <- colMat( R, M1, M ) & transpose( M1, Rs );

% colMat( C, M, CM ): matrix CM has column C as its first column; the
%                     remainder of its columns constitute matrix M.
% Example: colMat( [1,4], [[2,3],[5,6]], [[1,2,3],[4,5,6]] )
colMat( [], [], [] );
colMat( [X|Y], [Xs|Ys], [[X|Xs] | T] ) <- colMat( Y, Ys, T );

% nullRows( M ): matrix M only has empty rows in it.
% Example: nullRows( [[],[],[]] )
nullRows( [] );
nullRows( [[] | Rows] ) <- nullRows( Rows );
```

Suppose we know a matrix has two rows; then we obtain a specialized program by calculating a complete set of frontiers for the query `?transpose([R1,R2],T)`. The frontier of the partial derivation tree in Figure 2 consists of the clauses

```
T1: transpose( [[],[]], [] );
T2: transpose( [[X|Xs],[X'|Xs']], [[X,X'] | Rs] ) <-
            transpose( [Xs,Xs'], Rs );
```

where `T1` is the unconditional answer corresponding to the derivation ending with the empty query. `T2` is the conditional answer corresponding to right branch of the partial derivation tree. Since the only goal appearing in the body of the clauses is an instance of the root of the derivation tree, we know that this is a complete set of frontiers. Therefore, we have derived a specialized program to transform $n \times 2$ matrices.

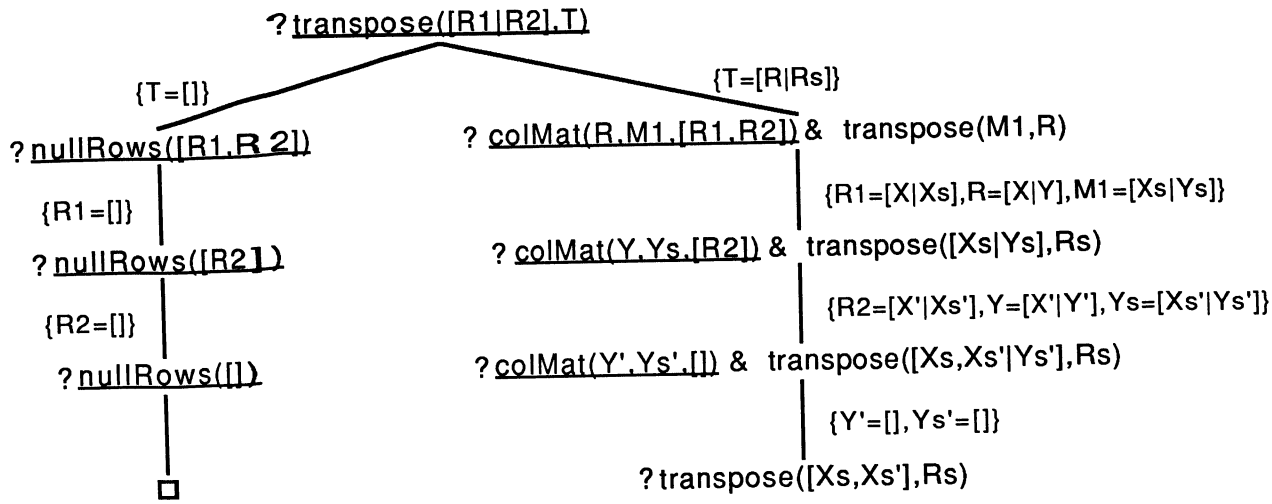

Figure 2: Partial derivation tree for transpose.

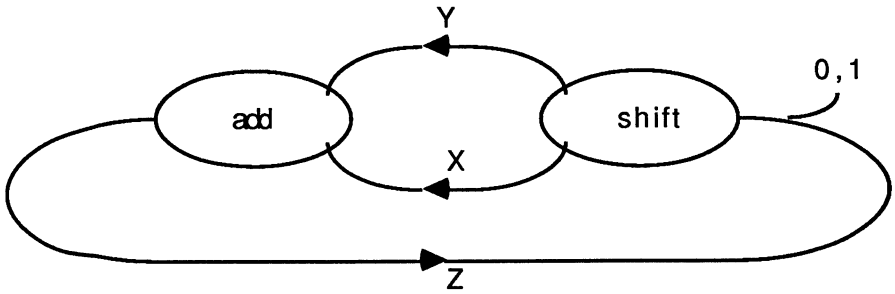

Figure 3: Dataflow network for the Fibonacci problem.

3 Example: specializing a meta-interpreter

Meta programming is a powerful technique in logic programming. An example is a meta-interpreter allowing the programmer to separate the control and the data component of the program. A major problem with meta programming is the inherent lack of efficiency. Partial evaluation can be used to solve this efficiency problem [3,11,10]. In this section we show how we can use partial derivation trees to perform this task. The frontier theorem guarantees us that the resulting program is both correct and complete.

The application we will use is dataflow programming. Consider the dataflow network shown in Figure 3, which finds the sequence of Fibonacci numbers (this type of network was proposed in [6]). To keep the exposition simple, we consider an example with only two nodes; more interesting examples of dataflow programming are shown in [16]. Each node represents a processor, each arc a communication channel. Data flows through the arcs in the direction associated with it. Each processor computes independently of the others, and is activated as soon as enough data is available in its input channels.

The `add` node adds the two numbers on its input channels, and puts the result on its output channel. The `shift` node needs two numbers on its input channel; it copies one of these to each of its output channels. It removes only the first number from its input

channel during the computation. The initial state of the network is shown in Figure 3; the channel between add and shift contains a 0 and a 1, the others are empty.

To represent the network in logic (see [13]), we first translate the nodes into a logic program. The definitions for the nodes in the network are

```
add( [X|Xs], [Y|Ys], [Z|Zs] ) <- sum( X, Y, Z ) & add( Xs, Ys, Zs );

shift( [X1,X2|Xs], [X1|Ys], [X2|Zs] ) <- shift( [X2|Xs], Ys, Zs );
```

The relation sum(X, Y, Z) holds if Z = X + Y. To solve the Fibonacci problem, we have to specify how the nodes in the network are connected, and what the initial state of the network is. This is done by the condition in

```
fib( [0,1|Z] ) <- shift( [0,1|Z], X, Y ) & add( X, Y, Z );
```

The connections are represented by shared variables between the nodes, where the variables represent the stream of data flowing through a particular channel. The initial state is the difference between the arguments of two nodes that are connected. If we now ask the query ?fib(F) we would like F to be instantiated to the stream of Fibonacci numbers (note that we have an infinite derivation, so that in fact we never obtain a successful answer substitution; so what we want is F to be instantiated during this infinite computation).

Unfortunately, when we run this query on the standard Prolog interpreter, F only gets instantiated to a stream of uninstantiated variables. This is because the AND-control required to solve this problem is more complex than the one provided by Prolog, which attempts to prove a goal completely before proving the next goal in a query. One solution to this problem is to transform this program so that the AND-control is replaced by OR-control [14], which is easy to provide in Prolog. In this example we demonstrate the alternative of building a meta-interpreter providing the required AND-control.

In this problem, it is sufficient to treat the goals in a query as a queue rather than a stack; as in the meta-interpreter:

```
% prove( L ): the list of goals L can be proven.
prove( [] );
prove( [Goal | Goals] ) <-
        clause( Goal, Body ) &
        append( Goals, Body, NewGoals ) &
        prove( NewGoals );

% append( U, V, W ): list W is list V appended to list U.
% example: append( [1,2], [3,4,5], [1,2,3,4,5] )
append( [], L, L );
append( [X|Xs], L, [X|NL] ) <- append( Xs, L, NL );
```

More complex dataflow networks require a meta-interpreter based on the "freeze" concept [4].

Now that we are using a meta-interpreter, we have to specify the clauses of our program through the clause predicate. The clauses for the add and shift relation are

```
% clause( H, B ) asserts that there is a clause with head H and
%                 body B in the program.
clause( add([X|Xs],[Y|Ys],[Z|Zs]),[add(Xs,Ys,Zs)] ) <- sum( X, Y, Z );
clause( shift([X1,X2|Xs],[X1|Ys],[X2|Zs]),[shift([X2|Xs],Ys,Zs)] );
```

Note that the `sum` predicate is moved outside the body as defined by the `clause` predicate. There are two reasons for this. First, it is a built-in predicate, which cannot be handled by our meta-interpreter as it stands. Moreover, we do not want it to be placed on the queue of goals to be executed. Instead, we want to execute it whenever we use the `add` clause. This is to make sure that the appropriate values get instantiated for the other clauses. As it turns out, this is not necessary in this case, as the `shift` clause does not require that the first two variables are instantiated before it can execute.

We also have to change the clause defining the actual `fib` predicate to

```
fib( [0,1|Z] ) <- prove( [shift([0,1|Z],X,Y),add(X,Y,Z)] );
```

If we now ask the query `?fib(F)` to Prolog, it will instantiate `F` to the stream of Fibonacci numbers.

Although the above program solves the Fibonacci problem, the meta-interpreter makes it inefficient. We will now eliminate the overhead of the meta-interpreter by using unfolding to obtain a complete set of frontiers. We are interested in the query `?fib(F)`. So we first create a frontier for this goal. By taking the partial derivation tree with only one level, we obtain the frontier with the clause

```
fib( [0,1|Z] ) <- prove( [shift([0,1|Z],X,Y),add(X,Y,Z)] );
```

which is the original clause defining `fib`. The goal

```
prove( [shift([0 1 | Z],X,Y),add(X,Y,Z)] )
```

is not an instance of the root of the initial partial derivation tree, and so we have to create a frontier for it. We use the query `?prove([shift(U,V,W),add(X,Y,Z)])` with a more general goal as the root of the partial derivation tree. This has the advantage that any goal with the same predicates is an instance of this goal, so we do not have to create any new frontiers when such a goal appears in a frontier. By taking the appropriate partial derivation tree, we obtain the frontier with the single clause

```
prove( [shift([U1,U2|Us],[U1|Vs],[U2|Ws]),add(X,Y,Z)] ) <-
        prove( [add(X,Y,Z),shift([U2|Us],Vs,Ws)] );
```

in it. The goal

```
prove( [add(X,Y,Z),shift([U2|Us],Vs,Ws)] )
```

is not an instance of any of the roots of the partial derivation trees created thus far. Again, using the query `?prove([add(X,Y,Z),shift(U,V,W)])` with a more general goal, we can obtain the frontier with the clause

```
prove( [add([X|Xs],[Y|Ys],[Z|Zs]),shift(U,V,W)] ) <-
        sum( X, Y, Z ) & prove( [add(Xs,Ys,Zs),shift(U,V,W)] );
```

in it.

Now we have a complete set of frontiers (note that the sum predicate is built-in, and we assume the definitions for built-in predicates are included in any set of frontiers). Hence, the program

```
fib( [0,1|Z] ) <- prove( [shift([0,1|Z],X,Y),add(X,Y,Z)] );
prove( [shift([U1 U2|Us],[U1|Vs],[U2|Ws]),add(X,Y,Z)] ) <-
        prove( [add(X,Y,Z),shift([U2|Us],Vs,Ws)] );
prove( [add([X|Xs],[Y|Ys],[Z|Zs]),shift(U,V,W)] ) <-
        sum( X, Y, Z ) & prove( [shift(U,V,W),add(Xs,Ys,Zs)] );
```

is equivalent to the original program. If we ask the query ?fib(F) it will instantiate F to the stream of Fibonacci numbers.

Alternatively, we can obtain the frontier

```
prove( [shift([U1,U2|Us],[U1|Vs],[U2|Ws]),add([X|Xs],[Y|Ys],[Z|Zs])] ) <-
        sum( X, Y, Z ) & prove( [shift([U2|Us],Vs,Ws),add(X,Y,Z)] );
```

for the query ?prove([shift(U,V,W),add(X,Y,Z)]) by considering a larger partial derivation tree. That way we obtain the program

```
fib( [0,1|Z] ) <- prove( [shift([0,1|Z],X,Y),add(X,Y,Z)] );
prove( [shift([U1,U2|Us],[U1|Vs],[U2|Ws]),add([X|Xs],[Y|Ys],[Z|Zs])] ) <-
        sum( X, Y, Z ) & prove( [shift([U2|Us],Vs,Ws),add(X,Y,Z)] );
```

which is equivalent to the original one, and is even more efficient than the previous version since an additional level of the meta-interpreter has been removed.

4 Concluding remarks

We have extended the state of the art in logic-based program transformation by combining a completeness-preserving unfold operation with specialization to a particular query. However, in doing so, we have not included other important program transformations, such as folding, the use of functionality, and the use of various other properties of predicates. Of course, these other transformations can be applied to the complete set of frontiers generated by our method in such a way that Sato and Tamaki's results guarantee completeness. In another iteration a set of frontiers can be generated which is complete and nonredundant with respect to the query of interest.

It would be more natural to apply the other transformations as soon as applicable, that is, to include them into the algorithm for generating complete sets of frontiers. This is in fact done in the program transformation system implemented by Strooper [9]. Under guidance of the user, this system builds a frontier for a specified query. The user determines interactively whether the current SLD-derivation should be extended, in this way exerting full control over the choice of frontier, while avoiding the error-prone operations such as performing substitutions and recording the resulting frontier. As soon as the frontier is completed, the user specifies any of the transformation rules other than

unfold to be applied to the frontier. The resulting set of frontiers is then tested for completeness and new frontiers are generated, as described in this paper.

As shown in [9], the implemented system has considerable practical potential. Future work is needed to analyze its correctness and completeness properties.

5 Acknowledgements

We gratefully acknowledge contributions to research facilities from the Natural Sciences and Engineering Research Council of Canada and from the Advanced Systems Institute of British Columbia. Thanks to Rajiv Bagai for his careful reading of an earlier version.

References

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

[2] K. Clark. *The Synthesis and Verification of Logic Programs.* Research Report, Imperial College, 1978.

[3] J. Gallagher. Transforming logic programs by specializing interpreters. In *Proc. of ECAI 86*, pages 109–122, 1986.

[4] F. Giannesini, H. Kanoui, R. Pasero, and M. van Caneghem. *Prolog.* Addison-Wesley, 1986.

[5] C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.

[6] G. Kahn and D.B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998, 1977.

[7] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[8] J.W. Lloyd and J.C. Shepherdson. *Partial Evaluation in Logic.* Technical Report CS-87-09, Bristol University, 1987.

[9] P.A. Strooper. *A Transformation System for Logic Programs.* ICR Report UW/ICR 87-10, University of Waterloo, December 1987.

[10] A. Takeuchi. Affinity between meta interpreters and partial evaluation. In *Information Processing 86*, pages 279–282, 1986.

[11] A. Takeuchi and K. Furukawa. *Partial Evaluation of Prolog Programs and its Application to Meta Programming.* Technical Report TR-126, ICOT, 1985.

[12] H. Tamaki and T. Sato. *A Transformation System for Logic Programs which Preserve Equivalence.* Technical Report TR-018, ICOT, 1983.

[13] M.H. van Emden and G.J. de Lucena Filho. Predicate logic as a language for parallel programming. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 189–198, Academic Press, London, 1982.

[14] M.H. van Emden and P. Szeredi. Converting and-control to or-control by program transformation. In J. Minker, editor, *Foundations of Databases and Logic Programming*, Morgan Kaufmann Publishers, 1987. to appear.

[15] P. Vasey. Qualified answers and their application to transformation. In *Proc. of the Third International Logic Programming Conference*, pages 425–432, 1986.

[16] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, London, 1985.

Appendix: proof of lemma 1

Before we can prove lemma 1, we have to prove another lemma consisting of two parts. The first part is a more general form of the lifting lemma ([7, page 43]) which deals with partial derivations as well as successful derivations.

Lemma 2 *Given a program P, a goal G and a substitution θ. If there exists a (possibly incomplete) derivation D in P starting from $G\theta$ and ending in $G_1, ..., G_m$ with mgu's $\theta_1, ..., \theta_n$ appearing along the way. Then:*

1. *There exists a derivation D' in P starting with G, in which the same goals are selected as in the corresponding nodes in D, and the same clauses from P are used to resolve these goals. If $\theta'_1, ..., \theta'_n$ are the mgu's along the way and $G'_1, ..., G'_{m'}$ are the final goals then $m = m'$ and there exists a substitution δ such that $\theta\theta_1...\theta_n = \theta'_1...\theta'_n\delta$ and $(G'_1...G'_m)\delta = G_1...G_m$.*
2. *$G\theta'_1...\theta'_n$ unifies with $G\theta$, say with mgu γ, and $G\theta\theta_1...\theta_n \leftarrow G_1...G_m$ is an instance of $(G\theta \leftarrow G'_1...G'_m)\gamma$.*

Proof We will prove the first part by induction on the length of D.

Base case: length of D is 1. Then $G\theta$ unifies with the head of a clause $F \leftarrow F_1...F_n$ with mgu θ_1. Thus $F\theta_1 = G\theta\theta_1$ and so F and G unify, say with mgu θ'_1. Moreover, since θ'_1 is the mgu of F and G, there exists a substitution δ such that $\theta'_1\delta = \theta\theta_1$. Hence $(F_1...F_n)\theta'_1\delta = (F_1...F_n)\theta\theta_1 = (F_1...F_n)\theta_1$ (we may assume θ does not act on any variables of $F \leftarrow F_1, ..., F_n$).

Induction step: let D be a derivation of length $n+1$ starting from $G\theta$. Assume that after n steps the goals are $G_1, ..., G_m$, and that G_i is the selected goal which unifies with the head of the clause $F \leftarrow F_1, ..., F_k$. Let $\theta_1...\theta_{n+1}$ be the mgu's along the way.

By the induction hypothesis, there exists a derivation D' of length n with mgu's $\theta'_1...\theta'_n$ ending in $G'_1, ..., G'_m$. There also exists a substitution δ such that $\theta'_1...\theta'_n\delta = \theta\theta_1...\theta_n$ and $(G'_1...G'_m)\delta = G_1...G_m$. Since $G'_i\delta = G_i$, G'_i also unifies with the head of $F \leftarrow F_1, ..., F_k$, say with mgu θ'_{n+1}. Since we also have $G_i\theta_{n+1} = G'_i\delta\theta_{n+1} = F\theta_{n+1}$, there exists a substitution δ' such that $\theta'_{n+1}\delta' = \delta\theta_{n+1}$, and hence $\theta'_1...\theta'_{n+1}\delta' = \theta\theta_1...\theta_{n+1}$. Moreover,

since we may assume that δ does not act on any variables of $F \leftarrow F_1, ..., F_k$, we have $(G'_1...G'_{i-1}F_1...F_kG'_{i+1}...G'_m)\theta'_{n+1}\delta' = (G_1...G_{i-1}F_1...F_kG_{i+1}...G_m)\theta_{n+1}$.

For the second part, we note that $G\theta'_1...\theta'_n\delta = G\theta\theta_1...\theta_n$, which means that $G\theta$ and $G\theta'_1...\theta'_n$ unify. Let γ be the mgu.

Since γ is the mgu of $G\theta$ and $G\theta'_1...\theta'_n$, there must exist a substitution α such that $G\theta'_1...\theta'_n\gamma\alpha = G\theta'_1...\theta'_n\delta$, Now, $(G\theta \leftarrow G'_1...G'_m)\gamma = (G\theta'_1...\theta'_n \leftarrow G'_1...G'_m)\gamma$, and $G\theta\theta_1...\theta_n \leftarrow G_1...G_m = (G\theta'_1...\theta'_n \leftarrow G'_1...G'_m)\delta$. From which it follows that $(G\theta \leftarrow G'_1...G'_m)\gamma\alpha = G\theta\theta_1...\theta_n \leftarrow G_1...G_m$, since we may assume that γ and δ do not act on any variables that appear in a body but not the corresponding head of the two clauses. ∎

Using the above lemma we can prove lemma 1.

Lemma 1 *Let S be a complete set of frontiers for a query Q and a program P. If the goal G is an instance of $R(F)$ for a frontier F in S, and if there is a successful derivation D of G in P, then there is a successful derivation D' of G in S. Moreover, if θ_1 and θ_2 are the compositions of all substitutions in D and D' respectively, then $G\theta_1$ is an instance of $G\theta_2$.*

Proof The proof will be by induction on the length of D. Let θ be the substitution such that $G = R(F)\theta$, and let T be the partial derivation tree corresponding to F.

We can use the switching lemma [7, page 45] repeatedly to obtain a successful derivation D'' of G in P in which the goals are selected in the same way as the corresponding path in T (since $G = R(F)\theta$ we know by the previous lemma that such a corresponding path exists). For the queries beyond the frontier of T we always select the leftmost goal in D''. Then D'' is of the same length as D, and if θ_3 is the composition of all substitutions in D'', then $G\theta_3$ and $G\theta_1$ are variants.

There are two cases to consider:

1. We reach success within the frontier (this includes the base case where the length of D'' is 1). Since G is $R(F)\theta$, we can use the previous lemma and conclude that F (and hence S) contains an unconditional answer of the form $R(F)\theta'$, which unifies with G. Thus we can use this unconditional answer to obtain a succesful derivation of G in S. Moreover, if θ_2 is the mgu of G and $R(F)\theta'$ then we can also use the lemma to conclude that $G\theta_3$ (and hence also $G\theta_1$) is an instance of $G\theta_2$.

2. We cross the frontier at a point with the remaining subgoals $G'_1, ..., G'_n$. By the previous lemma, the corresponding query in D'' contains the goals $G_1, ..., G_n$ and there exists a substitution δ so that $G_1...G_n = (G'_1...G'_n)\delta$. This means that the frontier contains the conditional answer $R(F)\theta' \leftarrow G'_1, ..., G'_n$. Since $R(F)\theta'$ unifies with G we can use this conditional answer to resolve G in S to obtain the goals $(G'_1, ..., G'_n)\gamma$, assuming γ is the mgu of G and $R(F)\theta'$. Moreover, $G\alpha \leftarrow G_1...G_n$ is an instance of $(G \leftarrow G'_1...G'_n)\gamma$, where α is the composition of the substitutions in D'' before $G_1...G_n$ is reached.

   We always select the leftmost goal in D'' from $G_1...G_n$ on. Hence, it must contain a succesful derivation D_1 of G_1, say with α_1 being the composition of the substitutions. Now G_1 is an instance of $G'_1\gamma$, and G'_1 appears in the frontier of S, so G'_1,

and hence G_1, must be the instance of a root of a frontier of S. Since D_1 is also shorter than D, we can use the induction hypothesis to conclude that there is a successful derivation of G_1 in S. But G_1 is an instance of $G'_1\gamma$, and so there is a derivation D'_1 of $G'_1\gamma$ in S. Moreover, if γ_1 is the composition of the substitutions in D'_1, then $G_1\alpha_1$ is an instance of $G'_1\gamma\gamma_1$.

Similarly, D'' must contain a succesful derivation D_i of $G_i\alpha_1...\alpha_{i-1}$, with α_i being the composition of the substitutions. By the induction hypothesis, there must exist derivations D'_i of $G'_i\gamma\gamma_1...\gamma_{i-1}$ in S such that $G_i\alpha_1...\alpha_i$ is an instance of $G'_i\gamma\gamma_1...\gamma_i$, where γ_i is the composition of the substitutions in D'_i.

So we obtain a successful derivation of G in S by first using the rule $R(F)\gamma' \leftarrow G'_1...G'_n$, and then using the derivations $D'_1, ..., D'_n$. Moreover, $G\alpha\alpha_1...\alpha_n$ is an instance of $G\gamma\gamma_1...\gamma_n$. ∎