TLP **4** (3): 355–369, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S147106840300190X Printed in the United Kingdom

TECHNICAL NOTE

Speedup of logic programs by binarization and partial deduction

JAN HRŮZA and PETR ŠTĚPÁNEK

Department of Theoretical Computer Science and Mathematical Logic, Charles University, Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
(*e-mail:* hruza@kti.mff.cuni.cz, petr.stepanek@mff.cuni.cz)

Abstract

Binary logic programs can be obtained from ordinary logic programs by a binarizing transformation. In most cases, binary programs obtained this way are less efficient than the original programs. (Demoen, 1992) showed an interesting example of a logic program whose computational behaviour was improved when it was transformed to a binary program and then specialized by partial deduction. The class of B-stratifiable logic programs is defined. It is shown that for every B-stratifiable logic program, binarization and subsequent partial deduction produce a binary program which does not contain variables for continuations introduced by binarization. Such programs usually have a better computational behaviour than the original ones. Both binarization and partial deduction can be easily automated. A comparison with other related approaches to program transformation is given.

KEYWORDS: logic programming, binarization, transformation, partial deduction, continuation

1 Introduction

Binary programs – programs consisting of clauses with at most one atom in the body – appear quite naturally when simulating computations of Turing machines by logic programs. Tärnlund (1977) introduced the concept of binary clauses. Since then various binarizing transformations have been defined (Maher, 1986; Štěpánková and Štěpánek, 1989; Sato and Tamaki, 1989; Tarau and Boyer, 1990). It is not difficult to show that the last three transformations produce programs with identical computational behaviour.

While in the beginning binarization was a rather theoretical issue, later, with the advent of Prolog compilers for programs consisting of binary clauses, it found important applications. Tarau (1992) built a Prolog system called BinProlog that makes use of binarization. In a preprocessing phase, the Prolog program is binarized (see Tarau and Boyer (1990)), and the binary program is compiled using BinWAM, a specialized version of the Warren Abstract Machine for binary programs. BinWAM is simpler than WAM and the size of the code of the binary program is reduced.

Hence, it is of practical use to investigate transformations changing a logic program to an equivalent binary logic program. It turned out that on some programs, binarization and partial deduction produce programs with a better performance, whereas on others, programs with a worse performance are produced. The goal of this paper is to describe a class of programs for which binarization followed by partial deduction produces programs with a better computational behaviour.

The paper is organized as follows. Section 2 presents the above mentioned transformation of logic programs to binary logic programs. Section 3 deals with the problem of computational efficiency of binarized programs. In section 4, B-stratifiable programs are introduced, and it is proved that the transformation consisting of binarization and partial deduction succeeds on these programs. This transformation usually leads to a computationally more efficient program. Section 5 gives results and conclusions.

We shall adopt the terminology and notation of Apt (1996). Let H be an atom, and let

$$\mathbb{A} \equiv A_1, A_2, \ldots, A_m \quad \text{and} \quad \mathbb{B} \equiv B_1, B_2, \ldots, B_n, \qquad m, n \geqslant 0$$

be (possibly empty) sequences of atoms. We restrict our attention to definite logic programs, that is programs consisting of clauses $H \leftarrow \mathbb{B}$ with the atom H in the head and a sequence $\mathbb{B}$ of atoms in the body. If $\mathbb{B}$ is empty, we write simply $H \leftarrow$. A clause is called *binary* if it has at most one atom in the body. A program consisting of binary clauses is called *binary*.

A *query* is a sequence of atoms. Queries are denoted by Q with possible subscripts. The *empty query* is denoted by $\square$. A computation of a logic program starts from a non-empty query and generates a possibly infinite sequence of queries by SLD-resolution steps. Maximal sequences of queries generated by this way are called *SLD-derivations*. Finite SLD-derivations are *successful* if they end with the empty query, otherwise they are *failed*.

In what follows, by an *LD-resolvent* we mean an SLD-resolvent with respect to the leftmost selection rule, and by an *LD-derivation* we mean an SLD-derivation w.r.t. the leftmost selection rule. Similarly, an *LD-tree* is an SLD-tree w.r.t. the leftmost selection rule. By *continuation* we mean a (possibly empty) list of terms representing goals (Štěpánková and Štěpánek, 1989; Tarau, 1992).

2 A transformation to binary logic programs

We shall describe the transformation (Štěpánková and Štěpánek, 1989) of definite logic programs to programs consisting of binary clauses. We define the operator B_S transforming the queries and the clauses of the input program. The resulting binary program is completed by an additional clause c_S.

Definition 2.1
Given a logic program P, let q be a new unary predicate symbol,

(i) for a query

$$Q \equiv A_1, A_2, \ldots, A_n$$

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

to P, let

$$B_S(Q) \equiv q([A_1, A_2, \ldots, A_n])$$

in particular, for the empty query, we put $B_S(\Box) \equiv q([\,])$.

(ii) for a clause

$$C \equiv H \leftarrow B_1, B_2, \ldots, B_n$$

let

$$B_S(C) \equiv q([H|Cont]) \leftarrow q([B_1, B_2, \ldots, B_n|Cont])$$

where *Cont* is a *continuation variable*. In particular, if C is a unit clause, then $B_S(C) \equiv q([H|Cont]) \leftarrow q(Cont)$.

(iii) the clause c_S is $q([\,]) \leftarrow$

(iv) for a program P, we put

$$B_S(P) \equiv \{B_S(C) | C \in P\} \cup \{c_S\}$$

Note that c_S is the only unit clause of the binarized program that provides the step $B_S(\Box) \Rightarrow \Box$ in successful SLD-derivations.

Example 2.2

Transformation of a program to its binary form

```
a :- b,c.                q([a|Cont]) :-  q([b,c|Cont]).
b :- d.                  q([b|Cont]) :-  q([d|Cont]).
c.                       q([c|Cont]) :-  q(Cont).
d.                       q([d|Cont]) :-  q(Cont).

                         q([]).
```

Note that

- we use a different syntax for Prolog programs and for the theory of logic programs, and
- continuation variables have been introduced for binarized programs. In what follows, we will use the term *continuation variable* also for variables in binary programs obtained by partial deduction of binarized programs containing clauses such as

```
q_b([a(X,1)|Cont]) :-  q_e([b(X),c(1)|Cont]).
```

Lemma 2.3 Let P be a program and Q a query. Then $B_S(P) \cup \{B_S(Q)\}$ has a successful LD-derivation with computed answer θ iff $P \cup \{Q\}$ does.

This follows from the fact that for every step of any LD-derivation of $P \cup \{Q\}$, there is a corresponding step of a corresponding LD-derivation of $B_S(P) \cup \{B_S(Q)\}$.

3 Transformations and binarization

Binarization can lead to more efficient programs

Contrary to a natural expectation – that binarization can only slow down the computations of a program because extra arguments and extra computation steps are involved in the transformed program – binarization followed by partial deduction can

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

in some cases speed the computation of a program up significantly. Demoen (1992) was the first to present a case study of such behaviour.

Transformation steps (1)

We consider the following steps of transformation:

1. Binarization.
2. Partial deduction with the empty continuation [ ] (i.e. the empty list) in the top-level call (see section 4).
3. Further partial deduction with final optimization steps such as removing duplicate variables (Leuschel and Sørensen, 1996; De Schreye *et al.*, 1999).

We show in Example 3.1 how the above transformation steps are applied to the SAMELEAVES program from Demoen (1992). We investigate programs for which this transformation gives more efficient programs when applied to programs with certain syntactical features, and why it leads to programs with identical or worse performance if applied to other programs. First, we recall the SAMELEAVES program.

Example 3.1

The program SAMELEAVES tests whether two binary trees have the same sequence of leaves. The trees with the same sequence of leaves need not be isomorphic.

Program SAMELEAVES

```
sameleaves(leaf(L),leaf(L)).
sameleaves(tree(T1,T2),tree(S1,S2)):-
           getleaf(T1,T2,L,T),
           getleaf(S1,S2,L,S),
           sameleaves(S,T).
getleaf(leaf(A),C,A,C).
getleaf(tree(A,B),C,L,O):-getleaf(A,tree(B,C),L,O).
```

As the first step of transformation, we apply the binarizing operator B_S from section 2 and obtain the following program:

```
q([sameleaves(leaf(L),leaf(L))|Cont]):-q(Cont).
q([sameleaves(tree(T1,T2),tree(S1,S2))|Cont]):-
           q([getleaf(T1,T2,L,T),
           getleaf(S1,S2,L,S),
           sameleaves(S,T)|Cont]).
q([getleaf(leaf(A),C,A,C)|Cont]):-q(Cont).
q([getleaf(tree(A,B),C,L,O)|Cont]):-
       q([getleaf(A,tree(B,C),L,O)|Cont]).
q([]).
```

Then we perform Steps 2 and 3. Using an automated partial deduction system Mixtus (Sahlin, 1993), we partially deduce the binarized program with the goal

```
q([sameleaves(Tree1,Tree2)])
```

where the continuation is empty (i.e. [ ]).

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

Applying steps 1 and 2, we obtain the following program:

```
sameleaves1(leaf(A), leaf(A)).
sameleaves1(tree(A,B), tree(C,D)) :-
            getleaf1(A,B,C,D).

getleaf1(leaf(C),D,A,B) :-
            getleaf2(A,B,C,D).
getleaf1(tree(A,D),E,B,C) :-
            getleaf1(A,tree(D,E),B,C).

getleaf2(leaf(C),A,C,B) :-
            sameleaves1(A,B).
getleaf2(tree(A,D),E,B,C) :-
            getleaf2(A,tree(D,E),B,C).
```

The resulting program is binary and has two specialized predicates for the two calls of `getleaf`. Demoen showed that it is faster by approximately 40%.

The `SAMELEAVES` example is interesting for yet another reason. If we skip binarization and perform only partial deduction on the original non-binary program, we get only an identical copy of the logic program. On the other hand, by binarization and partial deduction w.r.t. continuation [ ], that is by adding no information, we get a computationally more efficient binary program by partial deduction.

The program `FRONTIER` below computes the frontier, i.e. list of leaves of a binary tree. It serves as an example where the above described steps of binarization and partial deduction do not give any significant improvement.

Example 3.2 `FRONTIER`

```
frontier(leaf(X),[X]).
frontier(tree(Left,Right),Res):-
            frontier(Left,L1),
            frontier(Right,R1),
            append(L1,R1,Res).
```

If we perform the above steps on this program, we do not get a computationally more efficient program. Its performance is worse in terms of time and space. The length of the program obtained by binarization and partial deduction is significantly larger. It is so due to the fact that the partial deduction system cannot remove calls with a free continuation variable such as

```
q1([append([],B,B)| Cont]):- q(Cont).
```

Definition 3.3
We say that the binarization and partial deduction transformation consisting of Steps 1–3 *succeeds* if it terminates and eliminates all continuation variables in Steps 1 and 2.

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

4 B-stratifiable programs

In this section, we define the class of B-stratifiable programs, and prove that for this class of programs the transformation consisting of steps 1 and 2 succeeds (i.e. it eliminates continuation variables.)

Definition 4.1
We say that a program P is *B-stratifiable* if there is a partition of the set of all predicates of P into disjoint sets

$$S_0, S_1, \ldots, S_n \tag{2}$$

called *strata*, such that

(i) if there is a clause C such that a predicate symbol p, $p \in S_i$ occurs in the head of C and a predicate q, $q \in S_j$ occurs in the body of the same same clause C, then $i \geqslant j$, i.e. q belongs to a lower or the same stratum, and

(ii) in any clause $H \leftarrow \mathcal{B}$ of P where the predicate symbol p belongs to S_i from the head H there is at most one predicate symbol q from the same stratum S_i in the body $\mathcal{B}$. In this case, q is the predicate symbol of the rightmost atom in $\mathcal{B}$.

Then the set of strata (2) is called a *B-stratification* of P.

Example 4.2
Program

```
p :- q,p.
q :- r,r.                (3)
r.
r :- q.                  (4)
```

is not B-stratifiable because `q`,`r` are mutually dependent, and hence in the same stratum, but in the body of (3) there are two calls to `r`. If we remove the clause (4), the program becomes B-stratifiable. It suffices to take the B-stratification $S_1 = \{\mathtt{r}\}$, $S_2 = \{\mathtt{q}\}$, $S_3 = \{\mathtt{p}\}$. It is easy to check that the program `SAMELEAVES` is B-stratifiable while the program `FRONTIER` is not.

Note that the notion of B-stratifiable programs includes several classes of programs. It can be proved, for example, that *non-recursive* and *binary* programs are B-stratifiable. As for *tail-recursive* programs, in the literature we found no mathematical definition, but the notions of "tail-recursive" and "B-stratifiable" are similar.

We shall show that on B-stratifiable programs, the transformation consisting of binarization and partial deduction succeeds. B-stratifiable programs can be transformed with binarization and partial deduction into binary programs that are free of continuation variables – and usually more efficient. This is due to the fact that the number of terms representing goals in continuations is bounded.

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

Elimination of continuation variables

We shall show that for every B-stratifiable program P and a query Q, a there is partial deduction of $B_S(P)$ w.r.t. $B_S(Q)$ such that the resulting program does not contain any continuation variables.

To do this we introduce a simple partial deduction alogorithm, and prove that it terminates on B-stratifiable programs, giving a new program without continuation variables. Intuitively, we shall compute a partial deduction of $B_S(P)$ w.r.t. a set S. As the program $B_S(P)$ is binary, we can use an instance of the general partial deduction (Lloyd and Shepherdson, 1991) to remove the continuation variables. To this purpose, it is sufficient to compute (incomplete) LD-trees to the depth one. (A similar technique has been used in Gallagher and Bruynooghe (1990) and Sagonas and Warren (1995)).

To make sure that the conditions of so-called S-closedness and independence of S hold to guarantee termination, and that the partially deduced program computes the same set of answer substitutions, we use the following generalization operator.

Definition 4.3
We define a *generalization operator* G. Let

$$Q \equiv p_1(t_1, t_2, \ldots), p_2(t_{j_2}, t_{j_2+1}, \ldots), \ldots p_n(t_{j_n}, \ldots)$$

be a general (non-binary) query and

$$B_S(Q) \equiv q([p_1(t_1, t_2, \ldots), p_2(t_{j_2}, t_{j_2+1}, \ldots), \ldots p_n(t_{j_n}, \ldots)])$$

be the respective binarized query. We put

$$G(B_S(Q)) \equiv q([p_1(X_1, X_2, \ldots), p_2(X_{j_2}, X_{j_2+1}, \ldots), \ldots p_n(X_{j_n}, \ldots)])$$

where X_i are new variables. In particular, $G(q([])) \equiv q([])$.

Note that $B_S(Q)$ is an instance of $G(B_S(Q))$. Furthermore, we extend G so that it will be applied to sets of binarized queries and atoms. If S is a set of binarized queries, we put $G(S) \equiv \{G(Q) | Q \in S\}$.

Algorithm 1

Input: binarized program $B_S(P)$ and the set $\{G(B_S(Q))\}$, where P is a program and Q a query.
Output: a program *New_Prog* with no continuation variables, a set S, a new query Q'.

I. $S := \{\}$,
To_be_evaluated $:= \{G(B_S(Q))\}$,
Prog $:= \{\}$

II. **While** *To_be_evaluated* $\neq \{\}$ **do**

(a) take an atom $a \in$ *To_be_evaluated*; $S := S \cup \{a\}$;
(b) compute partial deduction of $B_S(P) \cup \{a\}$ obtaining an incomplete LD-tree of depth 1 (i.e. perform one unfolding step)

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

$R :=$ the set of resultants.
$B :=$ the set of bodies of resultants from R.

It follows from the fact that the program $B_S(P)$ is binary that all elements of B are atoms.

(c) $Prog := Prog \cup R$;
$To\_be\_evaluated := (To\_be\_evaluated \cup G(B)) - S$.

III. **Renaming** We define an operator *Ren* which renames each atom

$$q\big([p_1(t_1, t_2, \ldots), p_2(t_{j_2}, t_{j_2+1}, \ldots), \ldots, p_n(t_{j_n}, \ldots)]\big)$$

in *Prog* to

$$q\_p_1\_p_2\_\ldots\_p_n(t_1, t_2, \ldots, t_{j_2}, t_{j_2+1}, \ldots, t_{j_n}, \ldots)$$

obtaining the program *New_Prog* and the new query $Q' := Ren(B_S(Q))$. □

We can see that the continuation variables have been eliminated by Algorithm 1. To show that, we can verify that the following invariant holds during the computation of Algorithm 1, and that Algorithm 1 will terminate.

Invariant 4.4
No clause in *Prog* contains a free continuation variable.

Now we come to the main result of this section:

Theorem 4.5
Let P be a B-stratifiable program and Q a query. Then

1. Algorithm 1 terminates on the input $B_S(P)$, $\{G(B_S(Q))\}$.
2. Let *New_Prog* be the output program of Algorithm 1. Then $New\_Prog \cup \{Ren(B_S(Q))\}$ has an LD-derivation with a computed answer θ iff $P \cup \{Q\}$ does.

Proof
We need a definition and two lemmas that will enable us to prove termination of the algorithm.

Definition 4.6
Let P be a logic program, A an atom and let

$$A_1, \ldots, A_n$$

be a sequence of atoms. Let

$$A,\ \mathbb{B} \Rightarrow A_1, \ldots, A_n,\ \mathbb{B}\theta$$

be an LD-resolution step of $P \cup \{A\}$, where $\mathbb{B}$ denotes a (possibly empty) conjunction. We say that each atom $A_i \in \mathbb{A}$, $1 \leqslant i \leqslant n$ is an *immediate successor* of A, and write $A \succ A_i$. Let $\succeq$ be the reflexive and transitive closure of the immediate successor relation $\succ$. If $A \succeq B$, we say that B is a *successor* of A.

Lemma 4.7 Let P be a B-stratifiable logic program, let

$$S_0, S_1, \ldots, S_n \tag{5}$$

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

be a B-stratification of P, and let m be the maximum number of atoms in the body of a clause from P, and let $Q \equiv A_1, \ldots, A_l$ be a query and ξ be an arbitrary LD-derivation of $P \cup \{Q\}$. Then

(i) each atom $A_i, i = 1, \ldots, l$, has at most $n * (m-1) + 1$ successors in every LD-resolvent of ξ;
(ii) for each query Q' of at most l atoms, the number of atoms in any LD-resolvent of $P \cup \{Q'\}$is at most $n * (m-1) + l$. Hence there is a bound on the number of atoms in LD-resolvents of $P \cup \{Q\}$.

Proof

(i) In general, if A is an atom with a predicate symbol from a stratum S_k, $1 \leqslant k \leqslant n$, then A has at most $k * (m-1) + 1$ successors in every LD-resolvent in ξ. Hence $n * (m-1) + 1$ is a bound on the number of successors of an arbitrary atom in every LD-resolvent in ξ.
(ii) follows from (i).

□

Note that the number of elements of a continuation in any LD-resolvent of the binarized program $B_S(P) \cup \{B_S(Q)\}$ is equal to the number of atoms of the corresponding LD-resolvent of $P \cup \{Q\}$ minus 1 because for any LD-resolvent $A_1, A_2, \ldots A_n$ of $P \cup \{Q\}$, the corresponding continuation in the binarized program is $[A_2, \ldots, A_n]$.

Lemma 4.8
Let P be a program and let Q be a query. Assume that there is a bound on the number of atoms in all continuations in computations of $B_S(P) \cup \{B_S(Q)\}$. Then there is a bound on the number of sequences of predicate symbols in continuations that occur in computations of $B_S(P) \cup \{B_S(Q)\}$, too.

Proof of termination of Algorithm 1

Algorithm 1 terminates if the set *To_be_evaluated* of goals for partial deduction is empty. The elements of this set are atoms obtained by application of the generalization operator G. To guarantee the so called S−closedness condition of partial deduction (see Lloyd and Shepherdson (1991)), each goal evaluated by partial deduction is removed from *To_be_evaluated* and is put to S. The goals from the set $G(B) - S$ are added to *To_be_evaluated*, where B is the set of goals from the bodies of resultants obtained by partial deduction. It follows from the definition of G that it maps any two goals with the same sequence of predicate symbols to the same atom. It follows also that S is independent. We assumed that P is a B−stratifiable program, hence it follows from Lemmas 4.7 and 4.8 that there is a bound on the number of sequences of predicate symbols in continuations that occur in any resultant obtained by partial deduction of $B_S(P) \cup \{G(Q')\}$, where $G(Q')$ is a goal from *To_be_evaluated*. It turns out that after a finite number of steps, *To_be_evaluated* is empty and the computation of the algorithm terminates.

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

First, by Lemma 2.3 we can see that $B_S(P) \cup \{B_S(Q)\}$ has a successful LD–derivation with computed answer θ iff $P \cup \{Q\}$ does. Secondly, since $Prog \cup \{B_S(Q)\}$ is S-closed and S is independent, it follows from Theorem 4.2 of Lloyd and Shepherdson (1991) that the resulting program $Prog \cup \{B_S(Q)\}$ has the same computed-answer substitutions as the binarized program $B_S(P) \cup \{B_S(Q)\}$.

Thirdly, it is easy to see that the same holds for the renamed $New\_Prog \cup \{Ren(B_S(Q))\}$.

Once we have obtained a binary program without the continuation variables, further partial deduction can be performed without a limitation on the depth of LD trees. That partial deduction can improve performance of the program. Further improvement may be obtained by the RAF procedure (De Schreye *et al.*, 1999; Leuschel and Sørensen, 1996).

A negative result

Now we discuss the question of whether B-stratifiable programs are exactly those on which this transformation succeeds, i.e. whether for every non B-stratifiable program, the binarization and partial deduction fail to eliminate continuation variables.

Due to the fact that Algorithm 1 abstracts w.r.t. predicate symbols only and disregards terms, there are non-B-stratifiable programs for which the transformation still succeeds. For example, if we add the following clause to the `SAMELEAVES` program, we obtain a program which is not B-stratifiable but the transformation described in Example 3.1 still may succeed when applied to it:

```
getleaf(1,2,3,4) :- getleaf(5,6,7,8), getleaf(9,10,11,12).
```

This clause, which caused the program to become non-B-stratifiable, is in fact never used in LD-resolution for the query `sameleaves(X,Y)`.

On the other hand, we give a sufficient condition for programs for which the transformation does not succeed. We can show that for non-B-stratifiable programs for which the continuation can grow arbitrarily, Algorithm 1 does not terminate. This class of programs is large enough to include most of reasonable non-B-stratifiable programs. The idea of this proposition is analogous to the idea of Theorem 4.5.

Proposition 4.9
Suppose that a program P is not B-stratifiable, C is a clause of P containing a recursive call not in the last position in the body, and let there be an atomic query Q such that there is a successful LD-derivation for $P \cup \{Q\}$ in which C is used at least once. Then Algorithm 1 does not terminate with inputs $B_S(P)$ and $G(B_S(Q))$.

Proof
Let $C \equiv p(t_1, \ldots, t_n) \leftarrow \boldsymbol{A}, p(s_1, \ldots, s_n), \mathbb{B}$ be the recursive clause from the assumptions of the Proposition and let $\boldsymbol{A}$ and $\mathbb{B}$ be sequences of atoms such that $\mathbb{B}$ is not empty. Assume that the conditions of Proposition 4.9 hold. We shall proceed by contradiction. Assume that Algoritm 1 with inputs $B_S(P)$ and $G(B_S(Q))$ terminates.

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

As there is a successful LD-derivation which uses the clause C, Algorithm 1 will also use the binarized clause $B_S(C)$ and add the atom

$$A_1 \equiv q([\boldsymbol{A}', p(X_1, \ldots, X_n), \boldsymbol{B}', \ldots])$$

to the set *To_be_evaluated*. Note that, due to the generalization operator, $\boldsymbol{B}$ is an instance of $\boldsymbol{B}'$ and $\boldsymbol{A}$ is an instance of $\boldsymbol{A}'$.

It follows from the assumption that Algorithm 1 with the given inputs terminates that it will make empty the set *To_be_evaluated*. Hence, after a finite number of steps of Algorithm 1, an atom

$$A_1' \equiv q([p(X_1, \ldots, X_n), \boldsymbol{B}', \ldots])$$

will be added to the set *To_be_evaluated*.

This atom will later be selected for unfolding and using the clause $B_S(C)$, an atom

$$A_2 \equiv q([\boldsymbol{A}', p(X_1, \ldots, X_n), \boldsymbol{B}', \boldsymbol{B}', \ldots])$$

will be added to the set *To_be_evaluated*. This process is repeated infinitely many times, and Algoritm 1 does not terminate, a contradiction. This completes the proof of Proposition 4.9. □

Example 4.10

For the program `FRONTIER`, its only recursive clause and the query `front(X,Y)` which meet the assumptions of this proposition, the following atoms are added to the set *To_be_evaluated*:

```
q([front(X1,X2)]),
q([front(X1,X2),front(X3,X4),append(X5,X6,X7)]),
q([front(X1,X2),front(X3,X4),append(X5,X6,X7),front(X8,X9),
    append(X10,X11,X12)]),
q([front(X1,X2),front(X3,X4),append(X5,X6,X7),front(X8,X9),
    append(X10,X11,X12),front(X13,X14),append(X15,X16,X17)])
.........
```

Hence, the size of atoms that are added to the set *To_be_evaluated* grows indefinitely, and Algorithm 1 does not terminate.

5 Results and comparison

We present the results of our experiments with binarization and partial deduction and give some comparison. It may seem that the class of B-stratifiable programs is relatively small. Nonetheless, some transformations transform programs into B-stratifiable programs, improve the efficiency of the program significantly and allow for further binarization and partial deduction. We have experimented with a set of programs taken from Leuschel (1999), Leischel and Sørensen (1996), Demoen (1992) and Apt (1996).[1] We used Sicstus Prolog 3.8 running on a Linux workstation and test data of a reasonable size. The first column of Table 1 gives the name of the

[1] Listings of the programs can be found at `http://kti.mff.cuni.cz/~hruza/binary/`.

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

Table 1. *Speedups achieved*

Program	*Binarized*	*Algorithm 1*	*Final output*
sameleaves	0.42	0.97	1.31
frontier1	0.17	1.03	1.16
permutation	0.62	0.90	1.22
double-append	0.54	0.86	1.01
applast	0.67	0.95	1.00
match-append	0.48	0.93	1.05
remove	0.45	0.82	0.92
contains.lam	0.56	1.02	1.21

Table 2. *Memory usage data*

Program	*Binarized*	*Algorithm 1*	*Final program*
sameleaves	1.31	1.22	0.91
frontier1	1.34	1.06	0.95
permutation	1.38	1.16	0.93
double-append	1.33	1.18	0.99
match-append	1.47	1.24	0.93
applast	1.75	1.41	1.24
remove	1.27	1.16	1.06
contains.lam	1.30	1.26	1.20

program, the second through fourth columns give the respective speedups induced by binarization, partial deduction (Algorithm 1) and final optimization. (Speedup greater than 1 means the transformed program was faster.)

We can see that binarization slows programs down (as expected). Subsequent removal of continuation variables without optimization produces programs approximately as fast as the original ones, and the final optimization leads to speedups in some cases. On the other hand, in some cases, left-most selection rule (fixed by binarization) does not allow for optimizations achievable with flexible selection rule (e.g. double-append).

Table 2 shows the memory usage data for the programs, binarized programs and subsequently partially deduced programs. Memory usage was measured in Sictus Prolog 3.8.

This table gives relative usage of heap (global stack) space for the binarized programs, partially evaluated programs and the programs transformed in Steps 1–3. We can see an increase in heap usage of approximately 35% and a decrease for the transformed programs to approximately 95%.

Binarization with partial deduction and other approaches

Aanother transformation using binarization is described in Proietti and Pettorossi (1997). In their approach, continuations are introduced flexibly *during* transformation, and that allows for transformation during binarization. Their approach

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

can be understood as complementary to ours, as it transforms a program using unfolding, folding and generalization producing a binary program which can be further transformed. Our approach consits in using a straightforward binarization which is followed by tranformation and partial deduction.

Another related approach to transformation is Conjunctive Partial Deduction (CPD) (De Schreye *et al*., 1999). Unlike traditional partial deduction, which considers only atoms for partial deduction, conjunctive partial deduction attempts to specialize entire conjunctions of atoms. This approach is closely related to binarization with partial deduction. There is a difference, however. In the present approach, a program is first binarized, and hence does not contain any conjunctions. Then standard partial deduction can be used. Unlike that, in conjunctive partial deduction the conjunctions are left and the system decides on splitting conjunctions into appropriate subconjunctions.

Another difference is given by the fact that once a program is binarized, the selection rule is fixed, and no reordering of atoms can take place after binarization.

On some programs, conjunctive partial deduction profits from reordering of atoms during partial deduction, as it treats clause bodies as conjuctions and not as sequences of atoms.

It is a natural question as to whether on B-stratifiable programs binarization followed by partial deduction always gives results similar to conjunctive partial deduction. We shall show that it is not the case. We give an example of a program on which transformation consisting of binarization and partial deduction cannot give as good a result as conjunctive partial deduction.

Example 5.1
Program DOUBLEAPPEND

```
double_append(X,Y,Z,W) :-
        append(X,Y,V),
        append(V,Z,W).

append([X|Xs],Y,[X|Zs]) :- append(Xs,Y,Zs).
append([],Y,Y).
```

For this program, CPD uses flexible selection rule in the construction of SLD-trees, eliminating a second traversal of the first list. Such an optimization is not achievable using the standard leftmost selection rule in binarization with partial deduction, but is possible in conjunctive partial deduction.

The CPD approach may be somewhat more difficult to control, but it gives greater flexibility and applicability.

It follows from our experiments with the ECCE conjunctive partial deduction system (De Schreye *et al*., 1999) that binarization with partial deduction and CPD yields similar results when applied to the B-stratifiable programs, where optimization is achievable without atom reordering. A summary of results can be found in Table 3, where the first column gives the name of program, the second gives ratio of run–time for the output of conjunctive partial deduction system ECCE and binarization with

https://doi.org/10.1017/S147106840300190X Published online by Cambridge University Press

Table 3. *Summary of results*

Program	*Time of ECCE/bin + PD*
sameleaves	1.00
frontier1	1.15
permutation	0.84
match-append	0.59
double-append	0.61

partial deduction (greater than 1 means that binarization with partial deduction was faster). The output programs are similar yet not always identical (for the `SAMELEAVES` program ECCE produced the same programgram as binarization with partial deduction). For most programs, ECCE produced a faster output. In some cases, this is because of its flexible computation rule (e.g. double-append).

In general, conjunctive partial deduction can do all that binarization and partial deduction can. While classical partial deduction cannot handle conjunctions (as needed, for instance, for the `SAMELEAVES` program), binarization followed by partial deduction cannot use flexible selection rule (as in the `DOUBLEAPPEND` program) or split a conjunction in more parts, conjunctive partial deduction is the strongest of these transformation techniques.

Acknowledgements

We would like to thank Jan Hric and anonymous referees for their suggestions and comments. This research was partially supported by the Czech Science Foundation under Contract No. 209/01/0942.

References

Apt, K. R. 1996. *From Logic Programming to Prolog*. Prentice Hall.

Demoen, B. 1992. On the transformation of a Prolog program to a more efficient binary program. In: K.-K. Lau and T. Clement (Eds.) *Proceedings of LOPSTR 92, International Workshop on Logic Program Synthesis and Transformation, University of Manchester, Workshops in Computing series*, pp. 242–252. Springer-Verlag.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B. and Sørensen, M. H. 1999. Conjunctive partial deduction: foundations, control, algorithms and experiments. *Journal of Logic Programming 41*, 231–277.

Gallagher, J. P. and Bruynooghe, M. 1990. Some low-level source transformations for logic programs. *Proceedings of Meta90 Workshop on Meta Programming in Logic*, Leuven, Belgium.

Hrůza, J. and Štěpánek, P. 2000. Binary speed up for logic programs. In: J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv and P. J. Stuckey (Eds.) *Computational Logic – CL 2000, First International Conference, London 2000, Proceedings: Lecture Notes in Computer Science 1861*, pp. 116–130. Springer-Verlag.

Leuschel, M. 1999. *Dozens of Problems for Partial Deduction (A Set of Benchmarks)*. Available from `http://www.ecs.soton.ac.uk/~mal/systems/dppd/`

Leuschel, M. and Bruynooghe, M. 2002. Logic program specialisation through partial deduction: control issues. *Theory and Practice of Logic Programming 2*(4–5), 461–515.

Leuschel, M. and Sørensen, M. H. 1996. Redundant arguments filtering of logic programs. In: J. Gallagher (Ed.) *Logic Program Synthesis and Transformation, Proceedings of the 6th International Workshop, LOPSTR'96: Lecture Notes in Computer Science 1207*, pp. 83–103. Springer-Verlag.

Lloyd, J. W. and Shepherdson, J. C. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 217–242.

Maher, M. J. 1986. Equivalences of logic programs. In: E. Shapiro (Ed.) *Proceedings Third International Conference on Logic Programming: Lecture Notes in Computer Science 225*, pp. 410–424. Springer-Verlag.

Pettorossi, A. and Proietti, M. 1997. Flexible continuations in logic programs via unfold/fold transformatios and goal generalization. In: O. Danvy (Ed.) *Proceedings of the ACM Sigplan Workshop on Continuations*, BRICS Notes N6–93–13.

Proietti, M. and Pettorossi, A. 1994. Transformations of logic programs: foundations and techniques. *Journal of Logic Programming 19*, 261–320.

Sagonas, K. F. and Warren, D. S. 1995. Efficient execution of HiLog in WAM-based Prolog implentations. In: L. Sterling (Ed.) *Proceedings of the Twelfth International Conference on Logic Programming*, pp. 349–363. MIT Press.

Sahlin, D. 1993. Mixtus: an automatic partial evaluator for full Prolog. *New Generation Computing 12*(1), 7–51.

Sato, T. and Tamaki, H. 1989. Existential continuation. *New Generation Computing 6*, 421–438.

Štěpánková, O. and Štěpánek, P. 1989. Stratification of definite clause programs and of general logic programs. *Proceedings CSL '89, Third Workshop on Computer Science Logic: Lecture Notes in Computer Science 440*, pp. 396–408. Springer-Verlag.

Tarau, P. (1992) A continuation passing style Prolog engine. *Proceedings PLILP'92: Lecture Notes in Computer Science 631*, pp. 479–480. Springer-Verlag.

Tarau, P. and Boyer, M. 1990. Elementary logic programs. In: P. Deransart and J. Maluzsyński (Eds.) *Proceedings of PLILP'90: Lecture Notes in Computer Science 456*, pp. 159–173. Springer-Verlag.

Tärnlund, S. Å. 1977. Horn clause computability. *BIT 17*, 215–226.

Wand, M. 1980. Continuation-based program transformation strategies. *Journal of the ACM 27*(1), 164–180.