A Proof Method based on Folding Lemmas: Applications to Algorithm Correctness

Staffan BONNIER*
Laurent FRIBOURG

Laboratoire d'Informatique, URA 1327 du CNRS
Département de Mathématiques et d'Informatique
Ecole Normale Supérieure
*Linköping University, Sweden

LIENS - 92 - 28

December 1992

A Proof Method based on Folding Lemmas: Applications to Algorithm Correctness

Staffan Bonnier[1], Laurent Fribourg
L.I.E.N.S (URA 1327 CNRS) - 45 rue d'Ulm, 75005 Paris - France
e-mail: bonnier,fribourg@dmi.ens.fr

Abstract

In [Fri92] a proof method was developed for proving arithmetic consequences of Horn clause programs defined over integer lists and integers. To be applicable, the method requires the recursion schemes of all predicates involved to be compatible. This is to guarantee a sequence of unfold transformations to eventually lead to a foldable clause. In this paper we consider the case when such a compatibility is not present. To enable folding *folding lemmas* are constructed. The proof then proceeds using the old method, and the theorem is proved with the lemmas as hypotheses. The method is illustrated by proving correctness criteria for Boyer and Moore's string matching algorithm and for Dijkstra's descending subsequence algorithm.

1 Introduction

In [Fri92], a method was developed for proving implications of the form:

$$p_1(L, \overline{X}_1) \wedge \ldots \wedge p_n(L, \overline{X}_n) \Longrightarrow a(\overline{X}_1 \ldots \overline{X}_n)$$

where L is a list variable, each $\overline{X}_i$ is a vector of integer variables, and $a(\overline{X}_1 \ldots \overline{X}_n)$ is an arithmetic formula. The method works by transforming the hypothesis of the implication into an arithmetic formula $b(\overline{X}_1, .., \overline{X}_n)$, satisfying the equivalence:

$$\exists\, L:\ p_1(L, \overline{X}_1) \wedge \ldots \wedge p_n(L, \overline{X}_n) \iff b(\overline{X}_1, .., \overline{X}_n)$$

Proving the initial implication is thus reduced to proving the arithmetic formula:

$$b(\overline{X}_1, .., \overline{X}_n) \implies a(\overline{X}_1 \ldots \overline{X}_n)$$

[1]On leave from Linköping University, Linköping, Sweden. Financial support from the Swedish National Board for Industrial and Technical Development and from the Swedish Institute is gratefully acknowledged.

A crucial step in generating $b(\overline{X}_1, \ldots, \overline{X}_n)$ is the synthesis of a new predicate p, such that:

$$p(L, \overline{X}_1 \ldots \overline{X}_n) \iff p_1(L, \overline{X}_1) \wedge \ldots \wedge p_n(L, \overline{X}_n)$$

This synthesis is done by applying the unfold/fold transformations due to Sato and Tamaki [TS84] to the conjunction $p_1(L, \overline{X}_1) \wedge \ldots \wedge p_n(L, \overline{X}_n)$.

In addition to the basic scheme in [Fri92], several extensions of the method are considered, such as the admission of more than one list variable. An important assumption made in [Fri92] which is common to all extensions is that the recursion schemes of the p_i's are compatible, and, moreover, that the body of a recursive clause defining some p_i contains only arithmetic constraints in addition to the recursive call. This is essential for the unfolding transformations to eventually yield a clause which can be completely folded, and hence for making the synthesis of the predicate p possible. It may however be desirable to allow one or more p_i to violate these restrictions. In particular this is the case when formulating correctness criteria for algorithms; Several algorithms, which are themselves defined in terms of subroutines, do indeed contain extra literals in conjunction with the recursive calls when encoded as Horn clause programs.

In this paper we develop an extension of the method given in [Fri92]. In order to make folding possible also in the presence of predicates violating the mentioned restrictions, the extended method allows a rule of *folding enabling* to be applied during the synthesis of p. The rule allows replacements of bodies of clauses, so that an unfoldable clause can be turned into a foldable one. The prerequisites of applying the rule constitute *folding lemmas* on which the final proof relies, and which hence need to be proved separately. We then apply the extended method in order to prove correctness criteria for Boyer and Moore's string matching algorithm [BM79] and for Dijkstra's descending subsequence algorithm [Dij80].

The paper is organized as follows: Section 2 presents the general aim of the method, and introduces some notions and notation used in the remaining sections. Section 3 describes the principles behind the *basic method*, that is, the proof method developed in [Fri92]. The limitations of the basic method which serve as motivation for the present paper are also discussed in this section. The following section constitute the core of the paper in that it introduces the notions of *folding lemmas* and *folding enabling*. Finally, Section 5 presents two applications of the new method, and the paper is then concluded in Section 6.

2 General Aim

Let in the following Π denote a program defining certain predicates. It will without further mention be assumed that each argument of these predicates is either of type (non-negative) integer or of type list of (non-negative) integers. Standard arithmetic predicates are allowed to occur in the bodies of the clauses in Π. Our general aim

is to design a method for proving[2] formulae of the form:

$$A_1 \wedge \ldots \wedge A_n \Longrightarrow I \tag{1}$$

where $A_1, \ldots, A_n$ are atoms defined by Π, and where I is an arithmetic formula. We furthermore wish the method to carry out each proof by using $A_1 \wedge \ldots \wedge A_n$ to generate an arithmetic formula J such that:

$$A_1 \wedge \ldots \wedge A_n \Longrightarrow J \tag{2}$$

and hence to reduce the problem of proving (1) to that of proving the arithmetic implication:

$$J \Longrightarrow I \tag{3}$$

The formula J should thus be as strong as possible. Indeed, ideally it should satisfy:

$$(\exists\, \overline{L} :\ A_1 \wedge \ldots \wedge A_n) \Longleftarrow J \tag{4}$$

where $\overline{L}$ is the vector of all list variables occurring in $A_1 \wedge \ldots \wedge A_n$. When (2) holds, the method is said to be *correct* (for $A_1 \wedge \ldots \wedge A_n$). When (4) holds, the method is said to be *complete* (for $A_1 \wedge \ldots \wedge A_n$).

3 The Basic Method

3.1 General principles

The proof method presented in this paper extends the method developed in [Fri92]. This latter method will from now on be referred to as the *basic method*. It relies in itself on Sato and Tamakis unfold/fold transformation system [TS84] with which the reader is assumed to be familiar. In order to give an account of the basic method we first need to define the operation of projection. To simplify notation we assume in the following that each atom is of the form $p(\overline{L}, \overline{N})$, where $\overline{L}$ is the vector of list arguments and $\overline{N}$ is the vector of integer arguments.

Definition 3.1 The *projection* of $p(\overline{L}, \overline{N})$ is the atom $p^*(\overline{N})$, where p^* is a new predicate corresponding to p. By P^* we denote the program obtained by replacing each non arithmetic atom in P by its projection.

■

The following are the steps prescribed by the basic method, starting from the hypothesis $A_1 \wedge \ldots \wedge A_n$ in (1):

Step 1: Define a new predicate p (not occurring in Π) by the equivalence E:

$$p(\overline{L}, \overline{N}) \iff A_1 \wedge \ldots \wedge A_n$$

where $\overline{L}$, $\overline{N}$ are all variables occurring in $A_1, \ldots, A_n$.

[2] That is, proving the truth in the least model of Π over the domain of non-negative integers.

Step 2: Let Ψ be the result of applying unfold/fold transformations to $\Pi \cup \{E\}$.[3]

Step 3: Project Ψ to obtain Ψ^*.

Step 4: Generate an arithmetic formula J, such that:

$$J(\overline{N}) \Longleftrightarrow p^*(\overline{N})$$

holds in the least model of Ψ^*.

In certain cases the formula J in Step 4 may be generated by an extended form of bottom-up evaluation [FVP92].

Example 3.2 Suppose we wish to prove that:

$$\texttt{min(L, M)} \wedge \texttt{memb(E, L)} \implies \texttt{M} \leq \texttt{E}$$

where respectively `min(L, M)` and `memb(E, L)` holds iff `M` is the minimal element of the list `L`, and `E` is a member of `L`. The following clauses defining `min` and `memb` are assumed to be in Π:

```
min([X], X).
min([X|L], X) :-
          min(L, Y),
          X ≤ Y.
min([X|L], Y) :-
          min(L, Y),
          X > Y.
```

```
memb(X, [X|L]).
memb(X, [Y|L]) :-
              memb(X, L).
```

In **Step 1** the following equivalence E is defined:

$$\texttt{p(L, M, E)} \Longleftrightarrow \texttt{min(L, M)} \wedge \texttt{memb(E, L)}$$

By doing unfold/fold computations in **Step 2** we arrive at the program Ψ containing the following clauses, in addition to the clauses defining `min`:

```
p([X|L], X, X) :-
           min(L, Y),
           X ≤ Y.
p([X|L], M, X) :-
           min(L, M).
           X > M.
p([X|L], X, E) :-
           X ≤ Y,
           p(L, Y, E).
```

```
p([X], X, X).
```

```
p([X|L], M, E) :-
            X > M,
            p(L, M, E).
```

Projection of Ψ in **Step 3** thus yields the program Ψ^*, containing the following clauses in addition to the projection of the clauses defining `min`:

[3] We also allow here the removal of clauses on which the definition of p is not dependent.

```
p*(X, X) :-                          p*(X, X).
      min*(Y),
      X ≤ Y.

p*(M, X) :-
      min*(M).
      X > M.

p*(X, E) :-                          p*(M, E) :-
      X ≤ Y,                               X > M,
      p*(Y, E).                            p*(M, E).
```

Taking J(M, E) in **Step 4** to be M ≤ E, it is now easily seen that the following equivalence holds in the least model of Ψ^*:

$$\texttt{p}^*(\texttt{M},\ \texttt{E}) \iff \texttt{J}(\texttt{M},\ \texttt{E})$$

Thus the proof of the original theorem has been reduced to proving:

$$\texttt{M} \leq \texttt{E} \implies \texttt{M} \leq \texttt{E}$$

■

3.2 Correctness

By the definition of p made in Step 1, and by the fact that the unfold/fold transformations in Step 2 preserve the least model of a program [TS84], the following implication holds in the least model of $\Pi \cup \Psi$:

$$(\exists \overline{L} : A_1 \wedge \ldots \wedge A_n) \implies (\exists \overline{L} : p(\overline{L}, \overline{N})) \tag{5}$$

Furthermore, as is easily seen, the ground atom $p^*(\overline{n})$ has an SLD-refutation in Ψ^* whenever this is the case for $p(\overline{l}, \overline{n})$ in Ψ. Thus, by the completeness of SLD-resolution [Llo87] and by the definition of J in Step 4, the implications below hold in the least model of $\Psi^* \cup \Psi$:

$$(\exists \overline{L} : p(\overline{L}, \overline{N})) \implies p^*(\overline{N}) \iff J(\overline{N}) \tag{6}$$

By (5), (6) and the fact that Π does not define p, we may now conclude that the following implication holds in the least model of Π:

$$A_1 \wedge \ldots \wedge A_n \implies J(\overline{N})$$

This settles the correctness of the basic method.

3.3 Limitations

Due to the equivalence preservation of the unfold/fold rules, the implication in (5) is in fact bidirectional. Furthermore, in [Fri92] certain syntactical conditions on $A_1 \wedge \ldots \wedge A_n$ and Π were imposed, effectively ensuring the program Ψ resulting from Step 2 to be *independent of its list arguments*.

Definition 3.3 Ψ is said to be *independent of its list arguments* iff, for each predicate q defined by Ψ, the following equivalence holds in the least model of $\Psi^* \cup \Psi$:

$$(\exists \overline{L} : q(\overline{L}, \overline{N})) \iff q^*(\overline{N})$$

■

Hence, for the class of problems considered in [Fri92], also the first implication in (6) is bidirectional, and the basic method is thus complete.

In general however, the unfold/fold transformation system is not powerful enough to transform $\Pi \cup \{E\}$ into a program which is independent of its list arguments. More generally; restricting the transformations in Step 2 to unfolding and folding puts a definite limit to the class of theorems provable by the basic method: The implication (1) can be obtained only if there exists a sequence of unfold/fold transformations resulting in Ψ, such that:

$$p^*(\overline{N}) \Longrightarrow I \text{ holds in the least model of } \Psi^*. \tag{7}$$

The main obstacle in achieving (7) is that folding (introduction of recursion) often is not applicable due to the syntactical form of the unfolded clauses.

Example 3.4 Let the equivalence E defined in Step 1 be:

```
p(L, N1, N2) ⟺ a1(L, N1)∧ a2(L, N2).
```

Suppose the following are the clauses in Π defining `a1` and `a2`:

```
a1(l1, n1).                    a2(l2, n2).

a1(L, f1(N)) :-                a2(L, f2(N)) :-
       e1(T, L),                      e2(T, L),
       a1(T, N).                      a2(T, N).
```

Then E unfolds into clauses 1 to 4 below:

```
1      p(L, n1, n2) :- L=l1, L=l2.

2      p(l1, n1, f2(N2)) :-
                 e2(T, l1),
                 a2(T, N).

3      p(l2, f1(N1), n2) :-
                 e1(T, l2),
                 a1(T, N).

4      p(L, f1(N1), f2(N2)) :-
                 e1(T1, L),
                 a1(T1, N1),
                 e2(T2, L),
                 a2(T2, N2).
```

Unfolding of the last clause can only lead to a clause foldable by E if `e1` and `e2`, when unfolded, identify `T1` and `T2`.

■

4 Extending the Basic Method

4.1 Aim of the extension

Our aim is to allow other transformations in Step 2 than unfolding and folding, so that the resulting program Ψ is guaranteed to be independent of its list arguments. That is, completeness should not be lost when projecting Ψ in Step 3. Since independence of list arguments is a quite unmanageable property of programs (it is in fact undecidable), we first introduce a sufficient condition for it to hold.

Lemma 4.1 Suppose Ψ satisfies the following two conditions:

1. No list variable occurs more than once in the body of a clause in Ψ.
2. For each head $p(\overline{Lh}, \overline{Nh})$ of a clause in Ψ, and for each atom $p(\overline{Lb}, \overline{Nb})$ occurring in the body of a clause in Ψ, $\overline{Lh}$ is an instance of $\overline{Lb}$.

Then Ψ is independent of its list arguments.

■

The condition provided by Lemma 4.1 is strictly weaker than the conditions on list-recursiveness given in [Fri92]. The next task is thus to reformulate Step 2. To this end we will say that transformations applied to obtain Ψ from $\Pi \cup \{E\}$ are *correctness preserving* iff the implication (5) holds in the least model of $\Pi \cup \Psi$. When the converse of the implication holds, the transformations are said to be *completeness preserving*. Step 2a below now provides a first approximation of a new version of Step 2:

Step 2a: Let Ψ be a program satisfying Lemma 4.1, such that Ψ is the result of correctness preserving transformations of $\Pi \cup \{E\}$.

Note that the implications (5) and (6) are still valid, so the method obtained by replacing Step 2 by Step 2a is also correct. As already pointed out, the requirement on Ψ to satisfy Lemma 4.1 furthermore ensures that (6) can be replaced by:

$$(\exists \overline{L} : p(\overline{L}, \overline{N})) \iff p^*(\overline{N}) \iff J(\overline{N})$$

Thus, whenever the transformations in Step 2a are completeness preserving, the generated arithmetic expression J satisfies (4), i.e. the method is complete.

We will however not worry so much about completeness preservation of the new rules; As long as some order of their application yields a final Ψ such that (7) holds, they are "sufficiently" complete for proving the theorem at hand.

4.2 Goal deletion

Goal deletion allows the removal of arbitrary atoms in bodies of clauses. Let P be a program and let C be a clause. The rule may then be formulated as follows:

Goal deletion allows $P \cup \{C\}$ to be transformed into $P \cup \{C'\}$, where C' is obtained from C by the removal of one or more body atoms in C.

■

Adding goal deletion to the unfold/fold system does not violate correctness preservation of the transformations in Step 2. Completeness is however usually not preserved. It is worthwhile noting that by the application of goal deletion, any program can be transformed into one which satisfies the requirements in Lemma 4.1. Thus, when goal deletion is permitted, Step 2a can always be carried out successfully.

4.3 Folding enabling

As pointed out in Section 3.3, the limitations of the basic method become apparent when unfolding in Step 2 does not lead to foldable clauses. Our aim is therefore to introduce a transformation rule which allows folding to be enabled in situations similar to the one in Example 3.4. What we have in mind is essentially a variant of the goal replacement rule in [TS84]. More precisely, the rule should allow replacements of bodies of clauses, so that an unfoldable clause can be turned into a foldable one. The prerequisites of applying the rule will then constitute a *folding lemma* on which the final proof will rely.

Definition 4.2 Let C be a clause $H :- B_1, \ldots, B_m$, where each B_i is built from a predicate defined by Π. A *folding lemma* for C and the equivalence $p(\overline{L}, \overline{N}) \iff A_1 \wedge \ldots \wedge A_n$ is a conjunction λ of two implications of the forms 1 and 2 below:

1. $B_1 \wedge \ldots \wedge B_m \implies \exists \overline{L}' : \Sigma_L$
2. $B_1 \wedge \ldots \wedge B_m \wedge \Sigma_L \implies \exists \overline{N}' : A'_1 \wedge \ldots \wedge A'_n \wedge \Sigma_N$

where:

- Each A'_i is obtained from A_i by the replacement of $\overline{L}$, $\overline{N}$ for new variables $\overline{L}'$, $\overline{N}'$ not occurring in H or in $B_1, \ldots, B_m$.
- Σ_N is an arithmetic expression, called the *integer relation* of λ.
- Σ_L is a conjunction of atoms, called the *list witness* of λ.

■

Using the same notation as in Definition 4.2, the rule of folding enabling may now be formulated as follows:

Folding enabling allows $P \cup \{H \ :- B_1, \ldots, B_m\}$ to be transformed into the program $P \cup \{H \ :- \Sigma_N,\ A'_1, \ldots, A'_n\}$. The transformation is said to be *justified* by λ.

■

The purpose of the integer relation is to preserve certain arithmetic consequences of $B_1 \wedge \ldots \wedge B_m$ and to provide an appropriate relation in between the integer variables in H and those in $\overline{N}'$. Note that without such a relation, the program resulting from folding enabling can be used to derive that any integers are in the relation defined by H (assuming that $\exists \overline{L}' : A'_1 \wedge \ldots \wedge A'_n$ holds in the least model of Π).

The purpose of the list witness is just to specify values for the list variables in $\overline{L}'$, and hence to facilitate the realization (the proof) that the second implication of λ holds true, since otherwise a proof of this implication would require the explicit construction of appropriate lists. In all subsequent examples, the list witnesses will have the form:

$$L'_1 = L1 \wedge \ldots \wedge L'_n = L_n$$

where $L'_1, \ldots, L'_n$ are all variables in $\overline{L}'$. Hence implication 1 in Definition 4.2 will hold trivially and will therefore in general be omitted.

Example 4.3 Any folding lemma for clause 4 and E in Example 3.4 has the form:

```
e1(T1, L) ∧
a1(T1, N1) ∧
e2(T2, L) ∧
a2(T2, N2) ⟹ ∃ L': Σ_L

∧

e1(T1, L) ∧
a1(T1, N1) ∧
e2(T2, L) ∧
a2(T2, N2) ∧ Σ_L  ⟹ ∃ N1',N2':
                        a1(L', N1') ∧
                        a2(L', N2')
```

When taking for instance Σ_L to be $\mathtt{L'} = \mathtt{T1}$ and Σ_N to be $\mathtt{N1'} = \mathtt{N1} \ \wedge \ \mathtt{N2'} = \mathtt{g(N1, N2)}$, the above lemma immediately reduces to:

```
e1(T1, L) ∧
a1(T1, N1) ∧
e2(T2, L) ∧
a2(T2, N2)  ⟹ a2(T1, g(N1,N2))
```

■

It still remains to give conditions under which folding enabling together with unfolding and folding yields correctness preserving transformations. This will be the

case when the following two conditions hold. The notation is the same as the one used in Definition 4.2:

Correctness conditions:

1. λ holds in the least model of Π.
2. For each ground substitution θ for H there is a ground substitution σ for the remaining variables in λ, such that if $(B_1 \wedge \ldots \wedge B_m \wedge \Sigma_L)\theta\sigma$ holds in the least model of Π, then:
   - $\Sigma_N\theta\sigma$ holds in the least model of Π, and
   - Each $A'_i\sigma$ has a *rank-consistent proof* in the program resulting from the folding enabling justified by λ.

■

Condition 2 is needed to ensure that invariant I2 in [TS84] is preserved. This condition will however not be further considered in this paper. The reader is instead referred to [TS84] for the definition of rank-consistency and for conditions under which it holds.

4.4 The extension

We can now make a final reformulation of Step 2 of the basic method:

Step 2f: Let Ψ be a program satisfying Lemma 4.1, such that Ψ is obtained by transforming $\Pi \cup \{E\}$ using unfold/folding, goal deletion and folding enabling.

5 Two Examples

5.1 The Boyer-Moore string matching algorithm

The problem

The Boyer-Moore string matching algorithm [BM79] deals with the problem of finding the position of the first occurrence of a pattern P in a string S. Let us call this position (when it exists) the *string position* of the pattern in the string, and let us denote it $\sigma(P, S)$. For instance, if the pattern is:

```
P = 123
```

and the string is:

```
S = 121241231234
```

then σ(`P`, `S`) is `5` (assuming the numbering of the positions starts with `0`).

Principles of the algorithm

Let respectively $P[\,i\,]$ and $S[\,j\,]$ denote the i'th and the j'th element in the pattern P and the string S. Let furthermore n be an integer less than the length of P and let d be an integer less than or equal to n. The main insight underlying the correctness of Boyer and Moore's algorithm can now be expressed by the formula $\psi(n,\, d)$:

$$\psi(n,\, d) \;\equiv\; (\forall\, i:\; n - d \le i \le n \implies P[\,i\,] \neq S[\,n\,]) \implies \sigma(P,\, S) \ge d + 1 \qquad (8)$$

Informally $\psi(n,\, d)$ can be explained as follows; Suppose S contains the element e in its n'th position. Suppose furthermore that the n'th position in P, as well as the d consecutive positions to the left of the n'th, contain elements distinct from e. Then the string position of P in S must be at least $d + 1$. Hence, when computing the string position it is safe starting searching for P from the $d + 1$'th position in S.

Letting $|P|$ and $|S|$ respectively denote the lengths of S and P, and letting $S|i$ denote the suffix of S obtained by removing the first i positions of S, we may now give the following informal recursive formulation of Boyer and Moore's algorithm:

(BAS) If $|P| > |S|$, then $\sigma(P,\, S)$ is undefined. If P matches S, then $\sigma(P,\, S) = 0$.

(REC) Suppose $|P| \le |S|$ and that P does not match S. Let n be the last (largest) position such that $P[\,n\,] \neq S[\,n\,]$, and let d be the largest number such that $\psi(n,\, d)$ in (8) holds. Then $\sigma(P,\, S) = \sigma(P,\, S|(d + 1)) + d + 1$.

The following is a trace of the algorithm when `P` and `S` are defined as above:

$$\texttt{P} = \texttt{123} \;\text{ and }\; \texttt{S} = \texttt{121241231234}$$

The last mismatch is the position $\texttt{n} = \texttt{2}$, with $\texttt{S}[\,\texttt{2}\,] = \texttt{1}$. Thus $\texttt{d} = \texttt{1}$, and we drop the two first positions of `S`.

$$\texttt{P} = \texttt{123} \;\text{ and }\; \texttt{S} = \texttt{1241231234}$$

The last mismatch is again $\texttt{n} = \texttt{2}$, but $\texttt{S}[\,\texttt{2}\,] = \texttt{4}$ which does not occur in `P`, so we have also $\texttt{d} = \texttt{2}$. Dropping the three first positions of `S` now yields:

$$\texttt{P} = \texttt{123} \;\text{ and }\; \texttt{S} = \texttt{1231234}$$

Thus a match is found. In the concrete formulation of the algorithm given in [BM79], the search for n starts from the end of P and proceeds towards its beginning. The main reason for the efficiency of the algorithm is that the function which computes d when given n and $S[\,n\,]$ can be completely synthesized from P into an efficiently accessible finite table.

The definition of a correctness criterion

Our aim is to apply the proof method in order to prove (8) when n and d are chosen as in the recursive case of the formulation of the algorithm. The theorem to be proved may thus be expressed as follows:

```
strpos(P, S, Pos) ∧
lastnomatch(P, S, C, Nm) ∧
delta(C, Nm, P, D) ⟹ Pos ≥ D + 1
```

where brief descriptions of each of the predicates `strpos`, `lastnomatch` and `delta` are given below together with their definitions:

`strpos(P, S, Pos)` holds iff `P` matches `S` at position `Pos`. Thus `strpos` does in fact generalize σ as it was defined above. It should however be emphasized that the proof could be carried out just as well without this generalization. The reason for doing it is to keep the proof free from certain irrelevant details which would obscure the presentation.

```
strpos(P, S, 0) :-
          match(P, S).

strpos(P, [X|S], Pos+1) :-
          strpos(P, S, Pos).
```

The predicate `match` is defined in the standard way. We will however not apply unfolding to `match`, and therefore its definition is not given explicitly.

`lastnomatch(P, S, C, Nm)` holds iff `Nm` is the last position on which `P` and `S` do not agree, and `C` is the element at the `Nm`'s position in `S`.

```
lastnomatch([X|P], [Y|S], C, 0) :-
          X ≠ Y,
          match(P, S).

lastnomatch([X|P], [Y|S], C, Pos+1) :-
          lastnomatch(P, S, C, Pos).
```

`delta(C, Nm, P, D)` holds iff there are exactly `D` consecutive positions not containing `C` to the left of position `Nm` in `P`.

```
delta(C, 0, P, 0).

delta(C, D+1, [C|P], D) :-
          delta(C, D, P, D).

delta(C, D+1, [X|P], D+1) :-
          C ≠ H,
          delta(C, D, P, D).

delta(C, Nm+1, [X|P], D) :-
          D < Nm,
          delta(C, Nm, T, D).
```

The proof

Step 1 is carried out by defining the equivalence E:

```
p(P, S, Pos, D, Nm, C) ⟺ strpos(P, S, Pos) ∧
                         lastnomatch(P, S, C, Nm) ∧
                         delta(C, Nm, P, D)
```

Step 2f is commenced by unfolding `strpos` in E, which yields the two clauses:

```
1.1     p(P, S, 0,  D, Nm, C) :-
                  match(P, S),
                  lastnomatch(P, S, C, Nm),
                  delta(C, Nm, P, D).

1.2     p(P, [X|S], Pos+1, D, Nm, C) :-
                  strpos(P, S, Pos),
                  lastnomatch(P, [X|S], C, Nm),
                  delta(C, Nm, P, D).
```

Unfold/fold transformations of a new predicate, defined to be equivalent to the body of clause 1.1, would show that this body can not be satisfied (because of the presence of `match(P, S)` and `lastnomatch(P, S, C, Nm)`). Thus clause 1.1 can be discarded. Unfolding of `lastnomatch` in 1.2 now results in the following clauses (where `delta` has been subsequently unfolded to obtain 1.2.1):

```
1.2.1   p([Y|P], [X|S], Pos+1, 0, 0, X) :-
                  strpos([Y|P], S, Pos),
                  Y ≠ X,
                  match(P, S).

1.2.2   p([Y|P], [X|S], Pos+1, D, Nm+1, C) :-
                  strpos([Y|P], S, Pos),
                  lastnomatch(P, S, C, Nm),
                  delta(C, Nm+1, [Y|P], D).
```

Applying goal deletion to remove the body of clause 1.2.1 gives a unit clause as result:

```
1.2.1.1 p([Y|P], [X|S], Pos+1, 0, 0, X).
```

Unfolding of `delta` in clause 1.2.2 finally yields:

```
1.2.2.1 p([Y|P], [X|S], Pos+1, Nm, Nm+1, Y) :-
                  strpos([Y|P], S, Pos),
                  lastnomatch(P, S, Y, Nm),
                  delta(Y, Nm, P, Nm).
```

1.2.2.2 `p([Y|P], [X|S], Pos+1, Nm+1, Nm+1, C) :-`
`strpos([Y|P], S, Pos),`
`lastnomatch(P, S, C, Nm),`
`Y` $\neq$ `C,`
`delta(C, Nm, P, Nm).`

1.2.2.3 `p([Y|P], [X|S], Pos+1, D, Nm+1, C) :-`
`strpos([Y|P], S, Pos),`
`lastnomatch(P, S, C, Nm),`
`D` $<$ `Nm,`
`delta(C, Nm, P, D).`

It is clear that further unfolding of clauses 1.2.2.1 to 1.2.2.3 does not lead to clauses foldable by the initial equivalence E. Thus, in order to apply folding enabling, we construct the following three folding lemmas:

$\lambda.1$ `strpos([Y|P], S, Pos)` $\wedge$
`lastnomatch(P, S, Y, Nm)` $\wedge$
`delta(Y, Nm, P, Nm)` $\wedge \Sigma_{L1} \implies \exists$ `Pos', C', Nm', D':`
`strpos(P', S', Pos')` $\wedge$
`lastnomatch(P', S', C', Nm')` $\wedge$
`delta(C', Nm', P', D')` $\wedge \Sigma_{N1}$

Where:

Σ_{N1} : $\texttt{Pos}' \leq \texttt{Pos} + 1 \wedge \texttt{C}' = \texttt{Y} \wedge \texttt{Nm}' = \texttt{Nm} \wedge \texttt{D}' = \texttt{Nm}$

Σ_{L1} : $\texttt{P}' = \texttt{P} \wedge \texttt{S}' = \texttt{S}$

$\lambda.2$ `strpos([Y|P], S, Pos)` $\wedge$
`lastnomatch(P, S, C, Nm)` $\wedge$
`Y` $\neq$ `C` $\wedge$
`delta(C, Nm, P, Nm)` $\wedge \Sigma_{L2} \implies \exists$ `Pos', C', Nm', D':`
`strpos(P', S', Pos')` $\wedge$
`lastnomatch(P', S', C', Nm')` $\wedge$
`delta(C', Nm', P', D')` $\wedge \Sigma_{N2}$

Where:

Σ_{N2} : $\texttt{Pos}' \leq \texttt{Pos} + 1 \wedge \texttt{C}' = \texttt{C} \wedge \texttt{Nm}' = \texttt{Nm} \wedge \texttt{D}' = \texttt{Nm} \wedge \texttt{Y} \neq \texttt{C} \wedge (\texttt{Pos} = \texttt{Nm} \implies \texttt{Y} = \texttt{C})$

Σ_{L2} : $\texttt{P}' = \texttt{P} \wedge \texttt{S}' = \texttt{S}$

$\lambda.3$ `strpos([Y|P], S, Pos)` $\wedge$
`lastnomatch(P, S, C, Nm)` $\wedge$
`D` $<$ `Nm` $\wedge$
`delta(C, Nm, P, D)` $\wedge \Sigma_{L3} \implies \exists$ `Pos', C', Nm', D':`

```
                                        strpos(P', S', Pos') ∧
                                        lastnomatch(P', S', C', Nm') ∧
                                        delta(C', Nm', P', D') ∧ Σ_N3
```

Where:

Σ_{N3} : $\mathtt{Pos}' \leq \mathtt{Pos} + 1 \wedge \mathtt{C}' = \mathtt{C} \wedge \mathtt{Nm}' = \mathtt{Nm} \wedge \mathtt{D}' = \mathtt{D}$

Σ_{L3} : $\mathtt{P}' = \mathtt{P} \wedge \mathtt{S}' = \mathtt{S}$

Folding enabling justified by $\lambda.1$ to $\lambda.3$ followed by folding now yields:

```
1.2.2.1.1   p([Y|P], [X|S], Pos+1, Nm, Nm+1, Y) :-
                        Σ_N1,
                        p(P', S', Pos', D', Nm', C').

1.2.2.2.1   p([Y|P], [X|S], Pos+1, Nm+1, Nm+1, C) :-
                        Σ_N2,
                        p(P', S', Pos', D', Nm', C').

1.2.2.3.1   p([Y|P], [X|S], Pos+1, D, Nm+1, C) :-
                        Σ_N3,
                        p(P', S', Pos', D', Nm', C').
```

It is clear that the present program Ψ (consisting of clauses 1.2.1.1, 1.2.2.1.1, 1.2.2.2.1 and 1.2.2.3.1) satisfies the requirement in Lemma 4.1. Thus no further transformations are needed. **Step 3** consists in projecting Ψ to obtain the clauses below:

```
1.2.1.1*    p*(Pos+1, 0, 0, X).

1.2.2.1.1*  p*(Pos+1, Nm, Nm+1, Y) :-
                        Σ_N1,
                        p*(Pos', D', Nm', C').

1.2.2.2.1*  p*(Pos+1, Nm+1, Nm+1, C) :-
                        Σ_N2,
                        p*(Pos', D', Nm', C').

1.2.2.3.1*  p*(Pos+1, D, Nm+1, C) :-
                        Σ_N3,
                        p*(Pos', D', Nm', C').
```

To carry out **Step 4** we confine ourselves with verifying that the following disequality is an invariant of the clauses in Ψ^*:

$\mathtt{Pos} \geq \mathtt{D+1}$

and hence that it also must be a consequence of any formula J which characterizes the least model of $\mathtt{p}^*$. The formulae to be verified are:

1.2.1.1i $\quad \mathtt{Pos+1} \geq \mathtt{0+1}$

1.2.2.1.1i $\quad \mathtt{Pos'} \geq \mathtt{D'+1} \wedge \Sigma_{N1} \implies \mathtt{Pos+1} \geq \mathtt{Nm+1}$

1.2.2.2.1i $\quad \mathtt{Pos'} \geq \mathtt{D'+1} \wedge \Sigma_{N2} \implies \mathtt{Pos+1} \geq \mathtt{Nm+1+1}$

1.2.2.3.1i $\quad \mathtt{Pos'} \geq \mathtt{D'+1} \wedge \Sigma_{N3} \implies \mathtt{Pos+1} \geq \mathtt{D+1}$

It is immediate that 1.2.1.1i, 1.2.2.1.1i and 1.2.2.3.1i all hold. In the case of 1.2.2.2.1i we have:

$$\mathtt{Pos+1} \geq \mathtt{Pos'} \geq \mathtt{D'+1} = \mathtt{Nm+1}$$

We can however not have $\mathtt{Pos}+1 = \mathtt{Nm}+1$ since then, by Σ_{N2}, both $\mathtt{Y} \neq \mathtt{C}$ and $\mathtt{Y} = \mathtt{C}$ would follow. Implication 1.2.2.2.1i is thus a consequence of the fact that $\mathtt{Pos}+1 > \mathtt{Nm}+1$.

Intuition about the lemmas

There are two important implications present in lemmas $\lambda.1$, $\lambda.2$ and $\lambda.3$. The first one is common to all three lemmas:

$$\mathtt{strpos([Y|P],\ S,\ Pos)} \implies \exists\ \mathtt{Pos'}\colon\ \mathtt{strpos(P,\ S,\ Pos')} \wedge\ \mathtt{Pos'} \leq \mathtt{Pos+1}$$

Its truth may informally be explained as follows; If $[\mathtt{Y}|\mathtt{P}]$ occurs in $\mathtt{S}$ at position $\mathtt{Pos}$, then clearly $\mathtt{P}$ must occur in $\mathtt{S}$ at position $\mathtt{Pos}+1$. The first occurrence of $\mathtt{P}$ in $\mathtt{S}$ may however be at a position earlier than $\mathtt{Pos}+1$. The second implication is present only in lemma $\lambda.2$:

$$\mathtt{strpos([Y|P],\ S,\ Pos)} \wedge \mathtt{lastnomatch(P,\ S,\ C,\ Nm)} \implies \mathtt{(Pos = Nm} \implies \mathtt{Y=C)}$$

Its truth may be realized as follows; If $[\mathtt{Y}|\mathtt{P}]$ occurs in $\mathtt{S}$ at position $\mathtt{Pos}$, then the element in $\mathtt{S}$ at this position must be $\mathtt{Y}$. Moreover, by the definition of `lastnomatch`, $\mathtt{lastnomatch}(\mathtt{P},\ \mathtt{S},\ \mathtt{C},\ \mathtt{Nm})$ holds only if $\mathtt{C}$ is the content of the $\mathtt{Nm}$'th position in $\mathtt{S}$. Thus the implication must hold. It is worthwhile noticing that the latter implication does not involve any existentially quantified variables, and therefore could be proved using the method itself.

5.2 Dijkstra's descending subsequence algorithm

The problem

Dijkstra's descending subsequence algorithm [Dij80] deals with the problem of computing the length of a longest descending subsequence of a given sequence S.[4] For instance, if the sequence is:

$$\texttt{S} = \texttt{25341}$$

then `541` is a longest descending subsequence of `S` (and is in this case also unique), and the searched length is thus `3`.

Principles of the algorithm

In [Dij80], Dijkstra shows that the problem at hand can be solved by a one-pass algorithm, that is, an algorithm which takes the elements of S into consideration one by one in their order from right to left.[4] The algorithm recursively builds up a sequence $\pi(S)$ in the following way:

(BAS) If S is empty, then $\pi(S)$ is empty.

(REC) If X is greater than or equal to the last element in $\pi(S)$, or if S is empty, then $\pi(X.S) = \pi(S).X$. Otherwise $\pi(X.S)$ is obtained from $\pi(S)$ by replacing the left most element strictly greater than X by X.

The following is a trace of the steps taken to compute $\pi(\texttt{S})$ in the example above:

$$\begin{aligned}
\pi(\texttt{}) &= \texttt{} \\
\pi(\texttt{1}) &= \texttt{1} \\
\pi(\texttt{41}) &= \texttt{14} \\
\pi(\texttt{341}) &= \texttt{13} \\
\pi(\texttt{5341}) &= \texttt{135} \\
\pi(\texttt{25341}) &= \texttt{125}
\end{aligned}$$

The integer given as output by the algorithm is the length of $\pi(S)$. That is, $\pi(S)$ has the same length as the longest descending subsequence of S. The correctness of the output follows from the fact that the recursive definition of $\pi(S)$ given above implies (and is in fact equivalent to) the following characterization:

> The length of $\pi(S)$ equals the length of the longest descending subsequence in S. Moreover, for each n, the content of the n'th position in S is the least integer being the left most element of a descending subsequence of length $n+1$ in S. (9)

[4] Actually, the algorithm in [Dij80] searches S from left to right and deals with *ascending* subsequences. The algorithm discussed here is however completely dual to the original one.

The definition of a correctness criterion

Our aim is now to show how the proof method can be applied in order to prove the following weaker version of (9):

> The length of $\pi(S)$ is greater than or equal to the length of the longest descending subsequence in S. Moreover, for each n, the content of the n'th position in S is less than or equal to the least integer being the left most element of a descending subsequence of length $n+1$ in S.

The condition may be formalized as follows:

```
down(S, R) ∧
decss(S, [D1|D]) ∧
length(R, LenR) ∧
length([D1|D], LenD) ∧
last(R, Rl) ∧
nth(R, N, Rn) ⟹ LenD ≤ LenR ∧ (N + 1 = LenD ⟹ Rn ≤ D1)
```

The predicates involved are defined below:

`down(S, R)` holds iff `R` is the result of computing π(`S`) according to the recursive definition given above. It may thus be defined as follows:

```
down([], []).

down([X|S], R) :-
          down(S, T),
          ins(X, T, R).
```

where:

`ins(X, T, R)` holds iff `R` is the result of inserting `X` into `T` as specified in the recursive step of the definition of $\pi(S)$. Since we will not unfold `ins`, its definition is omitted.

`decss(S, D)` holds iff `D` is a decreasing subsequence of `S`. That is:

```
decss([], []).

decss([X|S], [X]).

decss([X|S], D) :-
          decss(S, D).

decss([X|S], [X,Y|D]) :-
          X ≥ Y,
          decss(S, [Y|D]).
```

`length(L, N)` holds iff `N` is the length of `L`:

```
length([], 0).

length([X|L], N+1) :-
          length(L, N).
```

`last(L, Ll)` holds iff `Ll` is the last element of `L`:

```
last([X], X).

last([X|L], Ll) :-
          last(L, Ll).
```

`nth(L, N, Ln)` holds iff `Ln` is the `N`'th element of `L`:

```
nth([X|L], 0, X).

nth([X|L], N+1, Ln) :-
          nth(L, N, Ln).
```

The proof

Step 1 consists in defining `p` by the following equivalence E:

$$\texttt{p(S, R, D, LenR, LenD, N, Rn, Rl, D1)} \iff \begin{array}{l}\texttt{down(S, R)} \wedge \\ \texttt{decss(S, [D1|D])} \wedge \\ \texttt{length(R, LenR)} \wedge \\ \texttt{length([D1|D], LenD)} \wedge \\ \texttt{last(R, Rl)} \wedge \\ \texttt{nth(R, N, Rn)}\end{array}$$

We start **Step 2f** by unfolding the equivalence on `down` to obtain the two clauses:

1.1
```
p([], [], D, LenR, LenD, N, Rn, Rl, D1) :-
          decss([], [D1|D]),
          length([], LenR),
          length([D1|D], LenD),
          last([], Rl),
          nth([], N, Rn).
```

1.2
```
p([X|S], R, D, LenR, LenD, N, Rn, Rl, D1) :-
          down(S, T),
          ins(X, T, R),
          decss([X|S], [D1|D]),
          length(R, LenR),
          length([D1|D], LenD),
          last(R, RL),
          nth(R, N, Rn).
```

Clearly the body of clause 1.1 can not be satisfied, and the clause can therefore be discarded. Unfolding of `decss` in 1.2 results in three clauses:

```
1.2.1    p([X|S], R, [], LenR, 1, N, Rn, Rl, X) :-
                 down(S, T),
                 ins(X, T, R),
                 length(R, LenR),
                 last(R, Rl),
                 nth(R, N, Rn).

1.2.2    p([X|S], R, D, LenR, LenD, N, Rn, Rl, D1) :-
                 down(S, T),
                 ins(X, T, R),
                 decss(S, [D1|D]),
                 length(R, LenR),
                 length([D1|D], LenD),
                 last(R, Rl),
                 nth(R, N, Rn).

1.2.3    p([X|S], R, [Y|D], LenR, LenD+1, N, Rn, Rl, X) :-
                 down(S, T),
                 ins(X, T, R),
                 X ≥ Y,
                 decss(S, [Y|D]),
                 length(R, LenR),
                 length([Y|D], LenD),
                 last(R, Rl),
                 nth(R, N, Rn).
```

Goal deletion of the first atom in the body of clause 1.2.1 yields:

```
1.2.1.1  p([X|S], R, [], LenR, 1, N, Rn, Rl, X) :-
                 ins(X, T, R),
                 length(R, LenR),
                 last(R, Rl),
                 nth(R, N, Rn).
```

Folding the body of 1.2.1.1 by the new predicate `q` results in:

```
1.2.1.1.1  p([X|S], R, [], LenR, 1, N, Rn, Rl, X) :-
                 q(T, R, LenR, N, Rn, Rl, X).
```

where the equivalence defining `q` is:

```
q(T, R, LenR, N, Rn, Rl, X)  ⟺  ins(X, T, R) ∧
                                 length(R, LenR) ∧
                                 last(R, Rl) ∧
                                 nth(R, N, Rn)
```

Under the (realistic) assumption that the definition of `ins` recurs on `T` and `R`, pure unfold/folding of this equivalence yields a program which satisfies the requirement in Lemma 4.1. Moreover, the following arithmetic formula is equivalent to

q^*(LenR, N, Rn, Rl, X):

$$\mathtt{N}+1 \leq \mathtt{LenR} \wedge (\mathtt{N}+1 = \mathtt{LenR} \implies \mathtt{Rn} = \mathtt{Rl}) \wedge (\mathtt{N} = 0 \implies \mathtt{Rn} \leq \mathtt{X}) \qquad (10)$$

Neither 1.2.2 nor 1.2.3 can be unfolded into foldable clauses. We therefore construct the following two folding lemmas:

$\lambda.1$

```
down(S, T) ∧
ins(X, T, R) ∧
decss(S, [D1|D]) ∧
length(R, LenR) ∧
length([D1|D], LenD) ∧
last(R, Rl) ∧
nth(R, N, Rn) ∧ Σ_L1  ⟹ ∃ D1', LenR', LenD', Rl', N', Rn':
                        down(S', R') ∧
                        decss(S', [D1'|D']) ∧
                        length(R', LenR') ∧
                        length([D1'|D'], LenD') ∧
                        last(R', Rl') ∧
                        nth(R', N', Rn') ∧ Σ_N1
```

Where:

Σ_{N1} : $\mathtt{D1}' = \mathtt{D1} \wedge \mathtt{LenD}' = \mathtt{LenD} \wedge$
$(\mathtt{LenR}' \geq \mathtt{LenD}' \implies \mathtt{N}'+1 = \mathtt{LenD}') \wedge$
$\mathtt{N}+1 \leq \mathtt{LenR} \wedge (\mathtt{N}+1 = \mathtt{LenR} \implies \mathtt{Rn} = \mathtt{Rl}) \wedge$
$(\mathtt{X} \geq \mathtt{Rl}' \implies \mathtt{Rl} = \mathtt{X} \wedge \mathtt{LenR} = \mathtt{LenR}'+1) \wedge$
$(\mathtt{X} < \mathtt{Rl}' \implies \mathtt{LenR} = \mathtt{LenR}') \wedge$
$(\mathtt{N} = \mathtt{N}' \implies \mathtt{Rn} \leq \mathtt{Rn}')$

Σ_{L1} : $\mathtt{S}' = \mathtt{S} \wedge \mathtt{R}' = \mathtt{T} \wedge \mathtt{D}' = \mathtt{D}$

$\lambda.2$

```
down(S, T) ∧
ins(X, T, R) ∧
X ≥ Y ∧
decss(S, [Y|D]) ∧
length(R, LenR) ∧
length([Y|D], LenD) ∧
last(R, Rl) ∧
nth(R, N, Rn) ∧ Σ_L2  ⟹ ∃ D1', LenR', LenD', Rl', N', Rn':
                        down(S', R') ∧
                        decss(S', [D1'|D']) ∧
                        length(R', LenR') ∧
                        length([D1'|D'], LenD') ∧
                        last(R', Rl') ∧
                        nth(R', N', Rn') ∧ Σ_N2
```

Where:

Σ_{N2}: $\mathtt{D1}' = \mathtt{Y} \wedge \mathtt{LenD}' = \mathtt{LenD} \wedge$

$(\texttt{LenR}' \geq \texttt{LenD}' \implies \texttt{N}' + 1 = \texttt{LenD}') \wedge$
$\texttt{N} + 1 \leq \texttt{LenR} \wedge (\texttt{N} + 1 = \texttt{LenR} \implies \texttt{Rn} = \texttt{Rl}) \wedge$
$(\texttt{X} \geq \texttt{Rl}' \implies \texttt{Rl} = \texttt{X} \wedge \texttt{LenR} = \texttt{LenR}' + 1) \wedge$
$(\texttt{X} < \texttt{Rl}' \implies \texttt{LenR} = \texttt{LenR}') \wedge$
$(\texttt{N} = \texttt{N}' \implies \texttt{Rn} \leq \texttt{Rn}') \wedge$
$\texttt{X} \geq \texttt{Y}$

Σ_{L2} : $\texttt{S}' = \texttt{S} \wedge \texttt{R}' = \texttt{T} \wedge \texttt{D}' = \texttt{D}$

Folding enabling followed by folding of the two clauses 1.2.2 and 1.2.3 now yields:

```
1.2.2.1     p([X|S], R, D, LenR, LenD, N, Rn, Rl, D1) :-
                        Σ_N1,
                        p(S', R', D', LenR' LenD', N', Rn', Rl', D1').
```

```
1.2.3.1     p([X|S], R, [Y|D], LenR, LenD+1, N, Rn, Rl, X) :-
                        Σ_N2,
                        p(S', R', D', LenR' LenD', N', Rn', Rl', D1').
```

At this point in the transformation the program Ψ contains the clauses 1.2.1.1.1, 1.2.2.1 and 1.2.3.1, plus those clauses which define q. As already remarked, the latter ones do satisfy the requirement in Lemma 4.1. Thus this is also the case for the whole program Ψ. This means that Step 2f is concluded. Projecting Ψ to obtain Ψ^* as prescribed by **Step 3**, now yields the clauses below plus the projection of the clauses defining q:

```
1.2.1.1.1*  p*(LenR, 1, N, Rn, Rl, X) :-
                        q*(LenR, N, Rn, Rl, X).
```

```
1.2.2.1*    p*(LenR, LenD, N, Rn, Rl, D1) :-
                        Σ_N1,
                        p*(LenR' LenD', N', Rn', Rl', D1').
```

```
1.2.3.1*    p*(LenR, LenD+1, N, Rn, Rl, X) :-
                        Σ_N2,
                        p*(LenR' LenD', N', Rn', Rl', D1').
```

As in the Boyer-Moore example, we will not carry out **Step 4** by generating an arithmetic expression equivalent to $\texttt{p}^*(\texttt{LenR}, \texttt{LenD}, \texttt{N}, \texttt{Rn}, \texttt{Rl}, \texttt{D1})$. Instead we will construct a formula $\phi(\texttt{LenR}, \texttt{LenD}, \texttt{N}, \texttt{Rn}, \texttt{Rl}, \texttt{D1})$ and prove that it is an invariant of the clauses defining $\texttt{p}^*$, and hence that it must hold in the least model of Ψ^*. The formula $\phi(\texttt{LenR}, \texttt{LenD}, \texttt{N}, \texttt{Rn}, \texttt{Rl}, \texttt{D1})$ is defined to be:

(a) $\texttt{LenR} \geq \texttt{LenD} \wedge$
(b) $\texttt{N} + 1 = \texttt{LenD} \implies \texttt{Rn} \leq \texttt{D1} \wedge$
(c) $\texttt{N} + 1 \leq \texttt{LenR} \wedge$
(d) $\texttt{N} + 1 = \texttt{LenR} \implies \texttt{Rn} = \texttt{Rl}$

The implications that need to be shown are thus:

1.2.1.1.1i $\texttt{q}^*(\texttt{LenR}, \texttt{N}, \texttt{Rn}, \texttt{Rl}, \texttt{X}) \implies \phi(\texttt{LenR}, 1, \texttt{N}, \texttt{Rn}, \texttt{Rl}, \texttt{X})$

1.2.2.1i $\quad \phi(\mathtt{LenR'\ LenD',\ N',\ Rn',\ Rl',\ D1'}) \wedge\ \Sigma_{N1}$
$\qquad\qquad\qquad \Longrightarrow\ \ \phi(\mathtt{LenR,\ LenD,\ N,\ Rn,\ Rl,\ D1})$

1.2.3.1i $\quad \phi(\mathtt{LenR'\ LenD',\ N',\ Rn',\ Rl',\ D1'}) \wedge\ \Sigma_{N2}$
$\qquad\qquad\qquad \Longrightarrow\ \ \phi(\mathtt{LenR,\ LenD+1,\ N,\ Rn,\ Rl,\ X})$

Implication 1.2.1.1.1i is immediate from (10). For 1.2.2.1i and 1.2.3.1i, (c) and (d) in the formulation of ϕ follows directly, since they are explicitly stated in Σ_{N1} and Σ_{N2}. Furthermore, in the implication 1.2.2.1i, (a) and (b) may be realized by:

(a) $\quad \mathtt{LenR} \geq \mathtt{LenR'} \geq \mathtt{LenD'} = \mathtt{LenD}.$

(b) $\quad \mathtt{LenR'} \geq \mathtt{LenD'},$ and hence $\mathtt{N}+1 = \mathtt{LenD} = \mathtt{LenD'} = \mathtt{N'}+1$
i.e. $\mathtt{N} = \mathtt{N'}$ which implies $\mathtt{Rn} \leq \mathtt{Rn'} \leq \mathtt{D1'} = \mathtt{D1}.$

For 1.2.3.1i, (b) can obtained in the same way as for 1.2.2.1i. Moreover, when $\mathtt{X} \geq \mathtt{Rl'}$ also (a) can be derived as for 1.2.2.1i, since then $\mathtt{LenR} = \mathtt{LenR'}+1$. What remains is to realize that (a) holds for 1.2.3.1i also in the case when $\mathtt{X} < \mathtt{Rl'}$. Thus, assume that:

```
LenR'=LenD'
```

Then, since $\mathtt{N'}+1 = \mathtt{LenD'}$, we can use (d) and (b) to conclude that:

$$\mathtt{Rl'} \leq \mathtt{Rn'} \leq \mathtt{D1'}$$

But we also have:

$$\mathtt{D1'} = \mathtt{Y} \leq \mathtt{X}$$

Hence the contradiction $\mathtt{X} < \mathtt{Rl'} \leq \mathtt{X}$ follows, which proves that $\mathtt{LenR'} = \mathtt{LenD'}$ can not be the case. We may therefore conclude:

$$\mathtt{LenR} = \mathtt{LenR'} \geq \mathtt{LenD'}+1 = \mathtt{LenD}+1$$

from which (a) follows.

Intuition about the lemmas

In essence, the two folding lemmas $\lambda.1$ and $\lambda.2$ serve as specifications of the predicate `ins(X, T, R)`, since their integer relations relate certain integer attributes of `T` to those of `R`. In particular it is expressed how the length of `R` depends on the relation between `X` and the last element of `T` (the fourth and the fifth lines in Σ_{N1} and Σ_{N2}).

It should also be noted that the truths of $\lambda.1$ and $\lambda.2$ do not depend on the presence of `down(S, T)` in the hypotheses, provided that the definition of `ins` agrees with the recursive step in the definition of $\pi(S)$ given before. As noted in [Pri81], a more efficient definition of `ins` may be used, which makes use of the fact that `T` is ordered whenever `down(S, T)` holds. Consequently, when such a definition is used, `down(S, T)` must be involved in the proof of the lemmas.

6 Final Remarks

We have given a method for proving arithmetic consequences of Horn clause programs defined over integer lists and integers. The new method extends the method developed in [Fri92]. In contrast to the latter, the method presented here may also handle predicates with incompatible recursion schemes and/or extra literals occurring in the bodies of recursive clauses. The usefulness of the method was demonstrated by proving correctness criteria for Boyer and Moore's string matching algorithm and for Dijkstra's descending subsequence algorithm.

There is one main problem caused by extending the class of theorems which can be proved by the old method; The unfold/fold transformation (which constitute a central step in [Fri92]) can no longer be carried out in a satisfactory way. More precisely, there is no longer any guarantee that unfolding leads to foldable clauses. In this paper we have proposed the introduction of *folding lemmas* in order to remedy this problem. The purpose of such a lemma is to justify the transformation of an unfoldable clause into a foldable one. Although the notion of a folding lemma was formally defined and some non-trivial examples of its use were given, there are still several problems which require further investigation in order for the method to become really useful. Some of these problems are mentioned below:

Checking rank-consistency

As was pointed out in Section 4.3 it is in general not sufficient to ensure the truths of the folding lemmas in order to ensure the correctness of the proof method. The reason is that fold enabling followed by folding may *decrease* the least model of a program if certain *rank-consistency* requirements are not satisfied. Informally, these requirements ensure that no infinite loops are introduced in the folding step. Indeed, the verification of these requirements remains in the two examples of Section 5. The conditions given in [TS84] for the safe application of goal replacement (of which fold enabling can be seen as a special case) do however seem somewhat complicated, since they require the comparison of the sizes of certain proof-trees. In many cases it seems however as if the requirements may be obtained by observing certain simple structural properties of the folding lemma. For instance, in the the Boyer-Moore example, the list witness specifies that `P` should take the place of `[Y|P]` while `S` remains unchanged. This together with information about how the predicates `strpos`, `lastnomatch` and `delta` recur over their list arguments, may be used to conclude that the rank-consistency requirement holds.

Strategies for generating folding lemmas

It still remains to be investigated at which points in the unfold/fold transformation the applications of folding enabling (and hence the introduction of folding lemmas) are needed. In particular this requires taking the definitions of the predicates in Π

into account, in order to realize that continued unfolding is not sufficient in order to obtain appropriate foldable clauses.

Generating the list witness

At present the list witness needs to be specified by the user. The heuristic used in the string-matching example of Section 5 was to "copy" most variables from the hypothesis of the lemma to the conclusion, in order to make the implication of most of the atoms in the conclusion trivial.

Generating the integer relation

Also the integer relation is given by the user in the current version of the method. As already remarked, there are however two kinds of relations present in the integer relation; (1) Those which are consequences of the hypothesis of the lemma, and (2) those which relate the old variables to the existentially quantified ones. It should at least be possible to infer the relations in (1) by using the method itself. It seems however as if generating appropriate relations of the kind (2) often requires a certain degree of insight into the problem at hand. For instance, in order to realize the need for the implication ($\texttt{LenR}' \geq \texttt{LenD}' \implies \texttt{N}' + \texttt{1} = \texttt{LenD}'$) in the descending subsequence example, one must realize that the N''th element of `R` is of relevance only when $\texttt{N}' + \texttt{1} = \texttt{LenD}'$.

References

[BM79] R.S Boyer and J.S Moore. *A Computational Logic.* ACM, 1979.

[Dij80] E.W Dijkstra. Some Beautiful Arguments Using Mathematical Induction. *Acta Informatica*, 13:1–8, 1980.

[Fri92] L. Fribourg. Mixing List Recursion and Arithmetic. In *Proc. 7th Symp. on Logic in Computer Science*, pages 419–429, Santa Cruz, 1992. IEEE.

[FVP92] L. Fribourg and M. Veloso Peixoto. Bottom-Up Evaluation of Datalog Programs with Arithmetic Constraints. Technical Report LIENS - 92 - 13, Laboratoire d'Informatique, Ecole Normale Supérieure, 1992.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987. 2nd edition.

[Pri81] P. Pritchard. Another Look at the "Longest Ascending Subsequence" Problem. *Acta Informatica*, 16:87–91, 1981.

[TS84] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. 2nd Int. Conf. on Logic Programming*, pages 127–138, Uppsala, 1984.