Mixing List Recursion and Arithmetic

Laurent Fribourg
L.I.E.N.S. (URA 1327 CNRS)
45 rue d'Ulm, 75005 Paris - France
fribourg@dmi.ens.fr

Abstract

We consider in this paper a special class of predicates. Every class predicate, say p, has one argument denoted Y of type list. All the other arguments are integers and form a vector denoted $\overline{X}$. *The predicates are primitive recursively defined over the structure of Y, and the auxiliary predicates are arithmetic relations over the integer arguments. When two atoms of this class, say* $p_1(Y, \overline{X}_1)$ *and* $p_2(Y, \overline{X}_2)$*, share the same list argument Y, it induces an implicit relation between the integers of* $\overline{X}_1$ *and* $\overline{X}_2$*. We describe a method for generating under certain conditions an arithmetic expression that characterizes this relation. The method is useful for proving or synthesizing inductive assertions about programs with arrays.*

1 Introduction

When proving assertions about programs with arrays, one often meets boundary conditions which can be encoded under the following form

$(\heartsuit)\qquad p_1(Y, \overline{X}_1)\wedge...\wedge p_n(Y, \overline{X}_n) \;\Rightarrow\; a(\overline{X}_1 \cup ... \cup \overline{X}_n)$

where Y is a list variable, $\overline{X}_1,...,\overline{X}_n$ are vectors of natural number variables, $a(\overline{X}_1 \cup ... \cup \overline{X}_n)$ is a linear arithmetic formula, and $p_1,...,p_n$ are predicates recursively defined [2, 4]. An example taken from [4] of such a form is

$$\min(Y,X_1) \wedge \max(Y,X_2) \;\Rightarrow\; \forall K,L\ (L \leq X_1 \wedge 0<K \;\Rightarrow\; L<X_2+K)\ .$$

where $\min(Y,X_1)$ and $\max(Y,X_2)$ mean that X_1 and X_2 are the minimum and maximum elements of the list Y.

Boyer and Moore have proposed to integrate a decision procedure for linear arithmetic into an inductive theorem prover capable of proving general properties of the recursive predicates [4]. The method they have developed is interesting but heavily relies on the knowledge of additional information (lemmas) about the recursive predicates. For example, in order to prove the above example, the following information should be available as a lemma: $\min(Y,X_1)\wedge\max(Y,X_2) \;\Rightarrow\; X_1 \leq X_2$. Their method is therefore essentially heuristic and has no completeness result.

In this paper we describe a procedure which constructs mechanically the appropriate lemmas. More precisely our procedure transforms the hypothesis $\exists Y\ p_1(Y, \overline{X}_1)\wedge...\wedge p_n(Y, \overline{X}_n)$ into an equivalent arithmetic formula $q(\overline{X}_1 \cup ... \cup \overline{X}_n)$. For example $\exists Y\ \min(Y,X_1)\wedge\max(Y,X_2)$ is mechanically transformed into the inequation $X_1 \leq X_2$. The whole formula $(\heartsuit)$ is thus reduced to a linear arithmetic formula and becomes truth-decidable. (See, e.g., [7, 17, 10] for linear arithmetic decision procedures.) We exhibit a certain subclass of formulas for which the procedure is guaranteed to terminate and thus constitutes a decision procedure (sections 2 and 3). This subclass allows for ordering over integers but not for incrementation. We then consider a more general subclass that allows for incrementation but we lose the termination property (section 4). We also indicate how to apply the method to a still more general subclass that allows for full arithmetic (section 5). Finally these results are extended to the case where predicates have more than one list argument (section 6).

2 A decidable subtheory of arithmetic with lists

In this section and the next one, we explain how to construct a decision procedure for a restricted class of arithmetic formulas with lists. We exploit a powerful result of Revesz [16], which gives a terminating bottom-up procedure for a class of Datalog programs (i.e., function-free logic programs) enriched with natural numbers and the order predicate <. We will use the conventions of Logic Programming: e.g, capital letters will denote variables, [] will denote the empty list, and [N|Y] a list of first element N and tail Y. Let

us recall some basic notions of [16]:
A *gap-ordering relation* $<_d$ is an ordering relation on natural numbers where the subscript d is a natural number denoting the minimum difference between two terms: $t_1 <_d t_2$ means $t_1 + d \leq t_2$. In particular, $<_0$ means $\leq$ and $<_1$ means $<$.
A *gap-configuration* is a total gap-ordering of variables and constants with at most one constant between variables: e.g., $X <_0 5 <_3 Y <_0 15 <_2 V = W$. (Note that a gap-configuration of the form $3 <_0 Y <_5 < X <_0 8$ means $X=8 \wedge Y=3$.)
An *ordering atom* is a formula of one of the following forms:
$X+d \leq W$, $X = W$, $X < d$, $X = d$, or $d < X$,
where X,W are variables of type natural number and d is a constant of type natural number. An *ordering set* is a finite conjunction of ordering atoms.

We now introduce a class of recursive definitions over lists and natural numbers based on gap-configurations.

definition 2.1

Let $\overline{X}$ be a vector of m distinct variables $X^1,...,X^m$ of type natural number, Y a variable of type list of natural numbers, and $p(Y,\overline{X})$ an atom. We say that $p(Y,\overline{X})$ is *orderly list-recursive* iff p is defined by the following program Π

$\{\ p([],\overline{X}) \leftarrow a_i(\overline{X}).\ \ \}_{i \in I}$
$\{\ p([N|Y],\overline{X}) \leftarrow b_j(\overline{X},N,\overline{X}'),\ p_j(Y,\overline{X}'_j).\ \ \}_{j \in J}$
$\{\ p([N|Y],\overline{X}) \leftarrow c_k(\overline{X},N,\overline{X}'),\ p(Y,\overline{X}').\ \ \}_{k \in K}$

where

I,J and K are finite sets of indices,
N is a variable of type natural number,
$\overline{X}'$ is a vector of m distinct variables $X'^1,...,X'^m$ of type natural number,
$\overline{X}'_j$ is a subvector of $\overline{X}'$,
$p_j(Y,\overline{X}'_j)$ is either the special atom *true* or an orderly list-recursive atom and $p_j \ll p$ for some well-founded predicate ordering $\ll$,
$a_i(\overline{X})$ is a gap-configuration over $\overline{X}$,
$b_j(\overline{X},N,\overline{X}')$ and $c_k(\overline{X},N,\overline{X}')$ are ordering sets over $\overline{X} \cup \{N\} \cup \overline{X}'$.

In the following we will also say that predicate p (resp. program Π) is orderly list-recursive.

examples 2.1

% lines preceded by '%' are comments .
% empty(Y) $\leftrightarrow$ Y is the empty list .

empty([]).

% min(Y,L) $\leftrightarrow$ L is the minimum of the list Y

min([N|Y],L) ← L=N, empty(Y).
min([N|Y],L) ← L′ ≤N, L=L′, min(Y,L′).
min([N|Y],L) ← L′ >N, L=N, min(Y,L′).

% max(Y,M) $\leftrightarrow$ M is the maximum of the list Y

max([N|Y],M) ← M=N, empty(Y).
max([N|Y],M) ← M′ ≤N, M=N, max(Y,M′).
max([N|Y],M) ← M′ >N, M=M′, max(Y,M′).

% member(Y,X) $\leftrightarrow$ X is an element of the list Y

member([N|Y],X) ← X=N.
member([N|Y],X) ← X=X′, member(Y,X′).

We now state our basic theorem (as well as a preliminary definition).

definition 2.2

Let Y be a variable of type list of natural numbers and $\overline{X}$ a vector of distinct natural number variables. A formula $f(Y,\overline{X})$ is said to be *gap-reducible* iff there exists a finite disjunction of gap-configurations $g(\overline{X})$ such that $g(\overline{X}) \Leftrightarrow \exists Y\ f(Y,\overline{X})$. The formula $g(\overline{X})$ is called a *gap reduction* of $f(Y,\overline{X})$.

Theorem 2.2

Let Y be a variable of type list of natural numbers, $\overline{X}_1,...,\overline{X}_n$ n pairwise disjoint vectors of distinct natural number variables, and $p_1(Y,\overline{X}_1)$, ..., $p_n(Y,\overline{X}_n)$ n orderly list-recursive atoms. There exists a procedure which produces a gap reduction of $p_1(Y,\overline{X}_1) \wedge ... \wedge p_n(Y,\overline{X}_n)$.

An immediate corollary of this theorem is that any formula
(♡) $\quad p_1(Y, \overline{X}_1) \wedge ... \wedge p_n(Y, \overline{X}_n) \ \Rightarrow\ a(\overline{X}_1 \cup ... \cup \overline{X}_n)$
where $a(\overline{X}_1 \cup ... \cup \overline{X}_n)$ is linear arithmetic, reduces to a linear arithmetic formula and is therefore truth-decidable. Note that the assumption that the numeric arguments of $p_1,...,p_n$ be all distinct variables is not restrictive: any term t_i^j located in the j-th position of p_i can indeed be replaced by a new variable X_i^j provided that $t_i^j = X_i^j$ be added as a hypothesis in the arithmetic subformula a.

The sketch of the procedure and its correctness proof are given in the following section.

3 Sketch of the procedure

The procedure makes use of the operations of folding/unfolding, (arithmetic) projection, variable elimination and bottom-up evaluation. We briefly review these operations and state the lemmas that underly the proof of the procedure correctness.

3.1 Folding/unfolding

The operation of folding/unfolding is a classical transformation defined for logic programs by Tamaki and Sato in [18]. We have

lemma 3.1

Consider two orderly list-recursive atoms $p_1(Y,\overline{X}_1)$ and $p_2(Y,\overline{X}_2)$ where Y is a variable of type list of natural numbers and $\overline{X}_1,\overline{X}_2$ are disjoint vectors of distinct natural number variables. Then the conjunction $p_1(Y,\overline{X}_1) \wedge p_2(Y,\overline{X}_2)$ can be transformed by folding/unfolding into an equivalent atom $p(Y,\overline{X}_1,\overline{X}_2)$ which is orderly list-recursive.

The lemma basically says that the class of orderly list-recursive atoms is preserved by conjunction. This allows to reduce the proof of theorem 2.2 to the case $n = 1$.

example 3.1

% minmax(Y,L,M) $\leftrightarrow$ min(Y,L) $\wedge$ max(Y,M) .
% By folding/unfolding we get

minmax([N|Y],L,M) $\leftarrow$ L=N, M=N, empty(Y).
minmax([N|Y],L,M) $\leftarrow$ L′ $\leq$N, M′ $\leq$N, L=L′, M=N, minmax(Y,L′,M′).
minmax([N|Y],L,M) $\leftarrow$ L′ $\leq$N, M′ >N, L=L′, M=M′, minmax(Y,L′,M′).
minmax([N|Y],L,M) $\leftarrow$ L′ >N, M′ $\leq$N, L=N, M=N, minmax(Y,L′,M′).
minmax([N|Y],L,M) $\leftarrow$ L′ >N, M′ >N, L=N, M=M′, minmax(Y,L′,M′).

3.2 Projection

The operation of projection consists in dropping one (or more) argument in the definition of recursive predicates [15, 11]: here the dropped argument is the list Y. We have

definition and lemma 3.2

Consider an orderly list-recursive predicate p defined by the following program

$\{\ p([],\overline{X}) \leftarrow a_i(\overline{X}).\ \}_{i\in I}$
$\{\ p([N|Y],\overline{X}) \leftarrow b_j(\overline{X},N,\overline{X}'),\ p_j(Y,\overline{X}'_j).\ \}_{j\in J}$
$\{\ p([N|Y],\overline{X}) \leftarrow c_k(\overline{X},N,\overline{X}'),\ p(Y,\overline{X}').\ \}_{k\in K}$

We say that the predicate p′ is the *projection* of p iff p′ is defined by the following program

$\{\ p'(\overline{X}) \leftarrow a_i(\overline{X}).\ \}_{i\in I}$
$\{\ p'(\overline{X}) \leftarrow b_j(\overline{X},N,\overline{X}'),\ p'_j(\overline{X}'_j).\ \}_{j\in J}$
$\{\ p'(\overline{X}) \leftarrow c_k(\overline{X},N,\overline{X}'),\ p'(\overline{X}').\ \}_{k\in K}$

where p'_j denotes the projection of p_j.

We have: $p'(\overline{X}) \Leftrightarrow \exists Y\ p(Y,\overline{X})$.

Informally the lemma says that one can forget about the list argument. This allows to reduce the class of orderly list-recursive programs to a class of Datalog programs with natural numbers and <.

The fact that $p(Y,\overline{X})$ implies $p'(\overline{X})$ simply holds because p′ is a projection of p. (It can be proven, e.g., by computation induction [6].) The fact that $p'(\overline{X})$ implies $\exists Y\ p(Y,\overline{X})$ holds because of the particular form of the recursion scheme of p over Y, and because Y is the removed argument. (Such an implication would not hold if an element of $\overline{X}$ was removed instead of Y, *cf.* §5.)

example 3.2

% empty(Y) $\leftrightarrow$ see example 2.1.
% By projection we get
empty′.

% minmax(Y,L,M) $\leftrightarrow$ see example 2.1.
% By projection we get
minmax′(L,M) $\leftarrow$ L=N, M=N, empty′.
minmax′(L,M) $\leftarrow$ L′ $\leq$N, M′ $\leq$N, L=L′, M=N, minmax′(L′,M′).
minmax′(L,M) $\leftarrow$ L′ $\leq$N, M′ >N, L=L′, M=M′, minmax′(L′,M′).
minmax′(L,M) $\leftarrow$ L′ >N, M′ $\leq$N, L=N, M=N, minmax′(L′,M′).
minmax′(L,M) $\leftarrow$ L′ >N, M′ >N, L=N, M=M′, minmax′(L′,M′).

3.3 Bottom-up evaluation

We now define a special class of Datalog programs with natural numbers.

definition 3.3

An *Orderlog program* is a logic program of the form

$$\{\ p^*(\overline{X}) \leftarrow u_j(\overline{X}).\quad \}_{j\in J^*}$$
$$\{\ p^*(\overline{X}) \leftarrow v_k(\overline{X},\overline{X}'),\ p^*(\overline{X}').\qquad \}_{k\in K^*}$$

where J^* and K^* are finite sets of indices, $\overline{X}$ (resp. $\overline{X}'$) is a vector of distinct variables, every $u_j(\overline{X})$ ($j \in J^*$) is a gap-configuration over $\overline{X}$, and every $v_k(\overline{X},\overline{X}')$ ($k \in K^*$) is an ordering set over $\overline{X}\cup\overline{X}'$. The predicate p^* is said to be *Orderlog-defined.*

The interest of focusing on Orderlog programs lies in the fact that any projection p' of an orderly list-recursive predicate p can be transformed into an an equivalent Orderlog-defined predicate p^*. This transformation essentially relies on standard techniques of variable elimination in linear arithmetic (see example below and discussion at the end of the section).

example 3.3

% *minmax' (L,M)* $\leftrightarrow$ *see example 3.2.*
% *By replacing empty' with true*
% *and eliminating variable N we get*

$$\text{minmax}^*(L,M) \leftarrow L{=}M.$$
$$\text{minmax}^*(L,M) \leftarrow L{\le}M,\ L{=}L',\ M'{\le}M,\ \text{minmax}^*(L',M').$$
$$\text{minmax}^*(L,M) \leftarrow M{>}L,\ L{=}L',\ M{=}M',\ \text{minmax}^*(L',M').$$
$$\text{minmax}^*(L,M) \leftarrow L{=}M,\ L'{>}L,\ M'{\le}M,\ \text{minmax}^*(L',M').$$
$$\text{minmax}^*(L,M) \leftarrow M{>}L,\ L'{>}L,\ M{=}M',\ \text{minmax}^*(L',M').$$

The process of bottom-up evaluation is a basic technique for handling a Datalog program. The process consists of iterations of the immediate consequence operator T associated with the program. When it terminates, it computes the minimal Herbrand model of the program and captures the program declarative semantics [9, 12]. The process has been adapted by Revesz in order to handle natural numbers and the interpreted relation $<$ [16]. From [16] it follows

lemma 3.4

Let Π^* be an Orderlog program defining a predicate p^*. The bottom-up evaluation process of Revesz on Π^* terminates and produces a finite disjunction of gap-configurations. This disjunction is called the *closed definition* of p^*.

example 3.4

Revesz's bottom-up computation over the minmax* program produces $L{=}M \vee L{\le}M \vee M{>}L$ as a closed definition, which reduces to $L{\le}M$. (The computation, which is not explained by lack of place, requires the transformation of the ordering sets appearing in the minmax* program into a disjunction of gap-configurations.) It follows that $L{\le}M$ is a gap-reduction of $\text{min}(Y,L) \wedge \text{max}(Y,M)$.

We can now explain how the procedure constructs the gap reduction, say $q(\overline{X}_1 \cup ...\cup\overline{X}_n)$, of $p_1(Y,\overline{X}_1)\wedge...\wedge p_n(Y,\overline{X}_n)$. First we transform $p_1(Y,\overline{X}_1)\wedge...\wedge p_n(Y,\overline{X}_n)$ into an equivalent atom $p(Y,\overline{X}_1,...,\overline{X}_n)$ by folding/unfolding. Then we construct the projection p' of p and replace p' with an equivalent Orderlog-defined predicate p^* as explained later on. Finally we compute the closed definition of p^* and take it as for $q(\overline{X}_1 \cup ...\cup\overline{X}_n)$.

Let us briefly explain how to transform the program defining p' (see §3.2) into the Orderlog program defining p^*. On the one hand, every $c_k(\overline{X},N,\overline{X}')$ is replaced with an equivalent atom $c_k{}^*(\overline{X},\overline{X}')$ by eliminating the variable N. On the other hand, by recursively applying the procedure, every $p_j(\overline{X}'_j)$ is replaced with its gap reduction $\bigvee_h q_j^h(\overline{X}'_j)$; then every $b_j(\overline{X},N,\overline{X}') \wedge q_j^h(\overline{X}'_j)$ is replaced with an equivalent disjunction of gap configurations over $\overline{X}$ by eliminating variables of $\overline{X}'\cup\{N\}$.

4 List recursion with integer incrementation

An important restriction in the class of orderly list-recursive programs is the prohibition of incrementation (e.g., $X{=}X'{+}1$) over recursion. The reason for this restriction originates from the definition of Revesz's Datalog programs that precludes incrementation. In order to overcome this restriction, we now consider the framework of Baudinet-Niézette-Wolper [1] instead of that of Revesz. The framework of Baudinet-Niézette-Wolper originally aims at representing temporal data and allow for incrementation over recursion. The numeric data they consider are integers rather than natural numbers. Revesz's key-notion of gap-configuration is there replaced by the notion of "constrained periodic set". Accordingly, the notion of ordering set will be replaced by that of "incrementation set". Constrained periodic sets are much

more general than gap-configurations. (They can indeed express all primitive recursive relation while gap-configurations cannot even express all the Presburger definable relations [1].) The price to be paid for this generality is that the bottom-up evaluation process for Datalog programs based on periodic sets does not always terminate. Let us recall basic notions of [1]:
A *linear repeating point* is a set $\{\alpha\nu+\beta\}_{\nu\in Z}$ where α and β are integers. (Note that such a set is reduced to the constant β when $\alpha=0$.) A *periodic tuple* is a tuple of linear repeating points.
An *incrementation set* $\mathrm{b}(\mathrm{t}_1,...,\mathrm{t}_m)$ where $\mathrm{t}_1,...,\mathrm{t}_m$ denote integer terms, is a finite conjunction of atoms of one of the following forms:

$\mathrm{t}_i+\delta\leq\mathrm{t}_j,\ \ \mathrm{t}_i+\delta=\mathrm{t}_j,\ \ \mathrm{t}_i<\delta,\ \ \mathrm{t}_i=\delta,\ \text{ or }\ \delta<\mathrm{t}_i,$

where δ is an integer constant, and $i,j\in\{1,...,m\}$.
A *constrained periodic set* $\mathrm{a}(\mathrm{t}_1,...,\mathrm{t}_m)$ is a periodic tuple $(\{\alpha_1\nu_1+\beta_1\},...,\{\alpha_m\nu_m+\beta_m\})$ together with an incrementation set $\mathrm{b}(\mathrm{t}_1,...,\mathrm{t}_m)$. It represents the set of tuples

$\{(\mathrm{t}_1,...,\mathrm{t}_m)\mid \mathrm{t}_1\in\{\alpha_1\nu_1+\beta_1\},...,\mathrm{t}_m\in\{\alpha_m\nu_m+\beta_m\}$ and $\mathrm{b}(\mathrm{t}_1,...,\mathrm{t}_m)$ is satisfied$\}$.

For instance, the constrained periodic set $\mathrm{a}(\mathrm{t}_1,\mathrm{t}_2)$ made of the tuple $(\{2\nu_1+3\},\{2\nu_2+5\})$ together with the incrementation set $\mathrm{t}_2=\mathrm{t}_1+2$ represents the infinite set $\{..., (-1,1), (1,3), (3,5), ...\}$.

We now introduce a class of recursive definitions over lists and integers based on constrained periodic sets.

definition 4.1

Let $\overline{\mathrm{X}}$ be a vector of m distinct variables $\mathrm{X}^1,...,\mathrm{X}^m$ of type integer, Y a variable of type list of integers, and $\mathrm{p}(\mathrm{Y},\overline{\mathrm{X}})$ an atom. We say that $\mathrm{p}(\mathrm{Y},\overline{\mathrm{X}})$ is *incrementally list-recursive* iff p is defined by the following program Π

$\{\ \mathrm{p}([],\overline{\mathrm{X}}) \leftarrow \mathrm{a}_i(\overline{\mathrm{X}}).\quad\}_{i\in I}$
$\{\ \mathrm{p}([\mathrm{N}|\mathrm{Y}],\overline{\mathrm{X}}) \leftarrow \mathrm{b}_j(\overline{\mathrm{X}},\mathrm{N},\overline{\mathrm{X}}'),\ \mathrm{p}_j(\mathrm{Y},\overline{\mathrm{X}}'_j).\quad\}_{j\in J}$
$\{\ \mathrm{p}([\mathrm{N}|\mathrm{Y}],\overline{\mathrm{X}}) \leftarrow \mathrm{c}_k(\overline{\mathrm{X}},\mathrm{N},\overline{\mathrm{X}}'),\ \mathrm{p}(\mathrm{Y},\overline{\mathrm{X}}').\quad\}_{k\in K}$

where

I,J and K are finite sets of indices, N is a variable of type integer,
$\overline{\mathrm{X}}'$ is a vector of m distinct variables $\mathrm{X}'^1,...,\mathrm{X}'^m$ of type integer,
$\overline{\mathrm{X}}'_j$ is a subvector of $\overline{\mathrm{X}}'$,
$\mathrm{p}_j(\mathrm{Y},\overline{\mathrm{X}}'_j)$ is either the special atom *true* or an incrementally list-recursive atom and $\mathrm{p}_j\ll\mathrm{p}$ for some well-founded predicate ordering $\ll$,
$\mathrm{a}_i(\overline{\mathrm{X}})$ is a constrained periodic set over $\overline{\mathrm{X}}$,
$\mathrm{b}_j(\overline{\mathrm{X}},\mathrm{N},\overline{\mathrm{X}}')$ and $\mathrm{c}_k(\overline{\mathrm{X}},\mathrm{N},\overline{\mathrm{X}}')$ are incrementation sets over $\overline{\mathrm{X}}\cup\{\mathrm{N}\}\cup\overline{\mathrm{X}}'$.

The use of incrementation enhances considerably the language expressivity. Besides the predicates given in example 2.1, one can for example define the N-th element of a list or the distance between the last occurrence of an integer in a list and the end of this list (see [4]). Here follows some explicit incrementally list-recursive definitions.

examples 4.1

% *length(Y,M)* $\leftrightarrow$ *M is the length of the list Y .*

```
length([],M) ← M=0.
length([N|Y],M) ← M=M'+1, length(Y,M').
```

% *lendel(Y,K,L)* $\leftrightarrow$ *L is the length of the list*
% *obtained by deleting all occurrences*
% *of K from the list Y .*

```
lendel([],K,L) ← L=0.
lendel([N|Y],K,L) ← K=N, K=K', L=L',
                        lendel(Y,K',L').
lendel([N|Y],K,L) ← K<N, K=K', L=L'+1,
                        lendel(Y,K',L').
lendel([N|Y],K,L) ← K>N, K=K', L=L'+1,
                        lendel(Y,K',L').
```

We define the counterpart of Orderlog programs (with periodic sets rather than gap-configurations) as follows

definition 4.2

An *Incrementalog program* is a logic program of the form

$\{\ \mathrm{p}^*(\overline{\mathrm{X}}) \leftarrow \mathrm{u}_j(\overline{\mathrm{X}}).\quad\}_{j\in J^*}$
$\{\ \mathrm{p}^*(\overline{\mathrm{X}}) \leftarrow \mathrm{v}_k(\overline{\mathrm{X}},\overline{\mathrm{X}}'),\ \mathrm{p}^*(\overline{\mathrm{X}}').\quad\}_{k\in K^*}$

where J^* and K^* are finite sets of indices, $\overline{\mathrm{X}}$ (resp. $\overline{\mathrm{X}}'$) is a vector of distinct integer variables, every $\mathrm{u}_j(\overline{\mathrm{X}})$ ($j\in J^*$) is a constrained periodic set over $\overline{\mathrm{X}}$, and every $\mathrm{v}_k(\overline{\mathrm{X}},\overline{\mathrm{X}}')$ ($k\in K^*$) is an incrementation set over $\overline{\mathrm{X}}\cup\overline{\mathrm{X}}'$. The predicate p^* is said to be *Incrementalog-defined.*

example 4.2

```
len2del*(K,L,M) ← L=0, M=0.
len2del*(K,L,M) ← K=K', L=L', M=M'+1,
                        len2del*(K',L',M').
len2del*(K,L,M) ← K=K', L=L'+1,
                  M=M'+1, len2del*(K',L',M').
```

The counterpart of lemma 3.4 for Incrementalog programs follows from Baudinet, Niézette and Wolper' work [1]:

lemma 4.3

Let p* be an Incrementalog-defined predicate. The bottom-up evaluation process of Baudinet-Niézette-Wolper produces a disjunction of constrained periodic sets whenever it terminates. This disjunction is called the *closed definition* of p*.

Note the essential difference with respect to lemma 3.4: the bottom-up evaluation process is no longer guaranteed to terminate. Our basic theorem 2.2 (and its preliminary definition) can now be restated as follows.

definition 4.3

Let Y be a variable of type list of integers and $\overline{X}$ a vector of distinct integer variables. A formula f(Y,$\overline{X}$) is said to be *periodic reducible* iff there exists a finite disjunction of constrained periodic sets g($\overline{X}$) such that g($\overline{X}$) $\Leftrightarrow$ $\exists$Y f(Y,$\overline{X}$) . The formula g($\overline{X}$) is called a *periodic reduction* of f(Y,$\overline{X}$).

Theorem 4.3

Let Y be a variable of type list of integers, $\overline{X}_1$, ..., $\overline{X}_n$ n pairwise disjoint vectors of distinct integer variables, and $p_1(Y,\overline{X}_1)$, ..., $p_n(Y,\overline{X}_n)$ n incrementally list-recursive atoms. There exists a procedure which, whenever it terminates, produces a periodic reduction of $p_1(Y,\overline{X}_1)\wedge ... \wedge p_n(Y,\overline{X}_n)$.

The procedure is similar to that described previously and also makes use of folding/unfolding, projection, variable elimination and bottom-up evaluation. This is illustrated by the following examples.

example 4.3 (folding/unfolding)

% *len2del(Y,K,L,M)* $\leftrightarrow$
% *lendel(Y,K,L)* $\wedge$ *length(Y,M)* .
% *By folding/unfolding we get*

len2del([],K,L,M) $\leftarrow$ L=0, M=0.
len2del([N|Y],K,L,M) $\leftarrow$ K=N, K=K′, L=L′, M=M′+1, len2del(Y,K′,L′,M′).
len2del([N|Y],K,L,M) $\leftarrow$ K<N, K=K′, L=L′+1, M=M′+1, len2del(Y,K′,L′,M′).
len2del([N|Y],K,L,M) $\leftarrow$ K>N, K=K′, L=L′+1, M=M′+1, len2del(Y,K′,L′,M′).

example 4.4 (projection)

% *By projection we get*

len2del′(K,L,M) $\leftarrow$ L=0, M=0.
len2del′(K,L,M) $\leftarrow$ K=N, K=K′, L=L′, M=M′+1, len2del′(K′,L′,M′).
len2del′(K,L,M) $\leftarrow$ K<N, K=K′, L=L′+1, M=M′+1, len2del′(K′,L′,M′).
len2del′(K,L,M) $\leftarrow$ K>N, K=K′, L=L′+1, M=M′+1, len2del′(K′,L′,M′).

example 4.5 (variable elimination and bottom-up evaluation)

% *By eliminating variable N,*
% *we get (cf. example 4.2)*

len2del*(K,L,M) $\leftarrow$ L=0, M=0.
len2del*(K,L,M) $\leftarrow$ K=K′, L=L′, M=M′+1, len2del*(K′,L′,M′).
len2del*(K,L,M) $\leftarrow$ K=K′, L=L′+1, M=M′+1, len2del*(K′,L′,M′).

Formally for the bottom-up evaluation process, we should make use of a generalized form of the immediate consequence operator T that computes over constrained periodic sets (see [1]). For the sake of simplicity, we will keep on using the classical operator T. Using the nonrecursive clause and starting from the empty set, application of T yields

C1: L=0 $\wedge$ M=0 .

Then, using the first recursive clause, iterated application of T yields

C2.1: L=0 $\wedge$ M>0 .

while, using the second recursive clause, iterated application of T yields

C2.2: L>0 $\wedge$ M>0 $\wedge$ L=M .

Using the second recursive clause, iterated application of T on C2.1 yields

C3: L>0 $\wedge$ M>1 $\wedge$ M>L .

Using the first recursive clause, iterated application of T on C2.2 yields L>0 $\wedge$ M>1 $\wedge$ M>L , which is subsumed by C3. Using the second recursive clause, iterated application of T on C3 yields L>1 $\wedge$ M>2 $\wedge$ M>L , which again is subsumed by C3. The bottom-up evaluation thus ends. The closed definition of len2del* is

(L=0 $\wedge$ M=0) $\vee$ (L=0 $\wedge$ M>0) $\vee$ (L>0 $\wedge$ M>0 $\wedge$ L=M) $\vee$ (L>0 $\wedge$ M>1 $\wedge$ M>L) .

This formula (which is equivalent to L$\leq$M) is the periodic reduction of lendel(Y,K,L) $\wedge$ length(Y,M).

5 Beyond incrementation

One can go one step further in extending the class of our list programs by not only allowing for incrementation over natural numbers (or integers) but allowing also for all arithmetic operations. Here we will not introduce a new class of terms such as 'gap-reduction' or 'linear repeating point', but will directly manipulate standard arithmetic terms (terms built upon natural numbers, + and $\times$) and standard arithmetic expressions (formulas built upon $<$, $=$ and arithmetic terms).

definition 5.1

Let $\overline{\mathrm{X}}$ be a vector of m distinct variables $\mathrm{X}^1,...,\mathrm{X}^m$ of type natural number, Y a variable of type list of natural numbers, and $\mathrm{p}(\mathrm{Y},\overline{\mathrm{X}})$ an atom. We say that $\mathrm{p}(\mathrm{Y},\overline{\mathrm{X}})$ is *arithmetically list-recursive* iff p is defined by the following program Π

$\{\ \mathrm{p}([],\overline{\mathrm{X}}) \leftarrow \mathrm{a}_i(\overline{\mathrm{X}}).\quad \}_{i\in I}$
$\{\ \mathrm{p}([\mathrm{N}|\mathrm{Y}],\overline{\mathrm{X}}) \leftarrow \mathrm{b}_j(\overline{\mathrm{X}},\mathrm{N},\overline{\mathrm{X}}'),\ \mathrm{p}_j(\mathrm{Y},\overline{\mathrm{X}}'_j).\quad \}_{j\in J}$
$\{\ \mathrm{p}([\mathrm{N}|\mathrm{Y}],\overline{\mathrm{X}}) \leftarrow \mathrm{c}_k(\overline{\mathrm{X}},\mathrm{N},\overline{\mathrm{X}}'),\ \mathrm{p}(\mathrm{Y},\overline{\mathrm{X}}').\quad \}_{k\in K}$
where

I,J and K are finite sets of indices, N is a variable of type natural number,
$\overline{\mathrm{X}}'$ is a vector of m distinct variables $\mathrm{X}'^1,...,\mathrm{X}'^m$ of type natural number,
$\overline{\mathrm{X}}'_j$ is a subvector of $\overline{\mathrm{X}}'$,
$\mathrm{p}_j(\mathrm{Y},\overline{\mathrm{X}}'_j)$ is either the special atom *true* or an arithmetically list-recursive atom and $\mathrm{p}_j \ll \mathrm{p}$ for some well-founded predicate ordering $\ll$,
$\mathrm{a}_i(\overline{\mathrm{X}})$ is a set of arithmetic expressions over $\overline{\mathrm{X}}$,
$\mathrm{b}_j(\overline{\mathrm{X}},\mathrm{N},\overline{\mathrm{X}}')$ and $\mathrm{c}_k(\overline{\mathrm{X}},\mathrm{N},\overline{\mathrm{X}}')$ are sets of arithmetic expressions over $\overline{\mathrm{X}}\cup\{\mathrm{N}\}\cup\overline{\mathrm{X}}'$.

examples 5.1

% *productlist(Y,M)* $\leftrightarrow$ *M is the product of all the*
% *elements of the list Y .*

productlist([],M) $\leftarrow$ M=1.
productlist([N|Y],M) $\leftarrow$ M=N$\times$M$'$, productlist(Y,M$'$).

% *productlist3* $\leftrightarrow$ *version of productlist with*
% *accumulator .*

productlist3([],K,L) $\leftarrow$ K=L.
productlist3([N|Y],K,L) $\leftarrow$ K$'$=N$\times$K, L=L$'$, productlist3(Y,K$'$,L$'$).

% *factlist(Y,M,P)* $\leftrightarrow$ *M is the product of all the*
% *elements of the list Y and Y is made of*
% *consecutive elements decreasing down to 1.*

factlist([],M,P) $\leftarrow$ M=1, P=0.
factlist([N|Y],M,P) $\leftarrow$ M=N$\times$M$'$, N=P, P=P$'$+1, factlist(Y,M$'$,P$'$).

% *factlist4* $\leftrightarrow$ *version of factlist with accumulator*

factlist4([],K,L,Q) $\leftarrow$ K=L, Q=0.
factlist4([N|Y],K,L,Q) $\leftarrow$ K$'$=N$\times$K, N=Q, L=L$'$, Q=Q$'$+1, factlist4(Y,K$'$,L$'$,Q$'$).

The basic theorem 2.2 (and its preliminary definition) is restated as follows.

definition 5.2

Let Y be a variable of type list of natural numbers and $\overline{\mathrm{X}}$ a vector of distinct natural number variables. A formula $\mathrm{f}(\mathrm{Y},\overline{\mathrm{X}})$ is said to be *arithmetic reducible* iff there exists a finite disjunction of arithmetic expressions sets $\mathrm{g}(\overline{\mathrm{X}})$ such that $\mathrm{g}(\overline{\mathrm{X}}) \Leftrightarrow \exists\mathrm{Y}\ \mathrm{f}(\mathrm{Y},\overline{\mathrm{X}})$. The formula $\mathrm{g}(\overline{\mathrm{X}})$ is called an *arithmetic reduction* of $\mathrm{f}(\mathrm{Y},\overline{\mathrm{X}})$.

Theorem 5.2

Let Y be a variable of type list of natural numbers, $\overline{\mathrm{X}}_1$, ..., $\overline{\mathrm{X}}_n$ n pairwise disjoint vectors of distinct natural number variables, and $\mathrm{p}_1(\mathrm{Y},\overline{\mathrm{X}}_1)$, ..., $\mathrm{p}_n(\mathrm{Y},\overline{\mathrm{X}}_n)$ n arithmetically list-recursive atoms. There exists a procedure which, whenever it terminates, produces an arithmetic reduction of $\mathrm{p}_1(\mathrm{Y},\overline{\mathrm{X}}_1)\wedge...\wedge\mathrm{p}_n(\mathrm{Y},\overline{\mathrm{X}}_n)$.

The procedure is similar to that described previously and also makes use of folding/unfolding, projection and bottom-up evaluation. However we will not perform variable elimination. Note that the reduction formula is generally no longer a linear arithmetic formula. (Therefore, even if such a reduction formula is eventually generated, the initial formula ($\heartsuit$) may not reduce to a truth-decidable one.) This is illustrated by the following examples.

example 5.2

% *newproduct(Y,K,L,M)* $\leftrightarrow$
% *productlist3(Y,K,L)* $\wedge$ *productlist(Y,M) .*
% *By folding/unfolding we get*

newproduct([],K,L,M) $\leftarrow$ K=L, M=1.
newproduct([N|Y],K,L,M) $\leftarrow$ K$'$=N$\times$K, L=L$'$, M=N$\times$M$'$, newproduct(Y,K$'$,L$'$,M$'$).

% By projection we get

newproduct$'$(K,L,M) $\leftarrow$ K=L, M=1.
newproduct$'$(K,L,M) $\leftarrow$ K$'$=N$\times$K, L=L$'$, M=N$\times$M$'$, newproduct$'$(K$'$,L$'$,M$'$).

Let T be the immediate consequence operator associated with newproduct$'$. Using the nonrecursive clause and starting from the empty set, application of T yields

C1: K=L $\wedge$ M=1 .

Then, using the recursive clause, application of T yields $\exists$K$'$,N (K$'$=N$\times$K $\wedge$ L=K$'$ $\wedge$ M=N$\times$1), which simplifies to

C2: L=M$\times$K .

Now C1 can be removed because it is subsumed by C2. A second application of T then yields $\exists$K$'$,M$'$,N (K$'$=N$\times$K $\wedge$ L=M$'\times$K$'\wedge$ M=N$\times$M$'$), but this is subsumed by C2 (because K$'$=N$\times$K$\wedge$L=M$'\times$K$'\wedge$M=N$\times$M$'$ implies L=M$\times$K). The bottom-up evaluation thus ends. The formula L=M$\times$K is the arithmetic reduction of productlist3(Y,K,L) $\wedge$ productlist(Y,M).

example 5.3

% newfact(Y,K,L,M,P,Q) $\leftrightarrow$
% factlist4(Y,K,L,Q) $\wedge$ factlist(Y,M,P) .
% By folding/unfolding we get

newfact([],K,L,M,P,Q) $\leftarrow$ K=L, M=1, P=0, Q=0.
newfact([N|Y],K,L,M,P,Q) $\leftarrow$ K$'$=N$\times$K, N=P, L=L$'$, M=N$\times$M$'$, N=Q, P=P$'$+1, Q=Q$'$+1, newfact(Y,K$'$,L$'$,M$'$,P$'$,Q$'$).

% By projection we get

newfact$'$(K,L,M,P,Q) $\leftarrow$ K=L, M=1, P=0, Q=0.
newfact$'$(K,L,M,P,Q) $\leftarrow$ K$'$=N$\times$K, N=P, L=L$'$, M=N$\times$M$'$, N=Q, P=P$'$+1, Q=Q$'$+1, newfact$'$(K$'$,L$'$,M$'$,P$'$,Q$'$).

Let T be the immediate consequence operator associated with newfact$'$. Using the nonrecursive clause and starting from the empty set, application of T yields

C1: K=L $\wedge$ M=1 $\wedge$ P=0 $\wedge$ Q=0 .

Then, using the recursive clause, application of T yields $\exists$K$'$,N (K$'$=N$\times$K $\wedge$ N=P $\wedge$ L=K $\wedge$ M=N$\times$1 $\wedge$ N=Q $\wedge$ P=1+0 $\wedge$ Q=1+0), which simplifies to

C2: L=K $\wedge$ M=1 $\wedge$ P=1 $\wedge$ Q=1 .

A second application of T then yields $\exists$K$'$,N (K$'$=N$\times$K $\wedge$ N=P $\wedge$ L=K$'$ $\wedge$ M=N$\times$1 $\wedge$ N=Q $\wedge$ P=1+1 $\wedge$ Q=1+1), that is

C3: L=2$\times$K $\wedge$ M=2 $\wedge$ P=2 $\wedge$ Q=2 .

A third application of T yields

C4: L=6$\times$K $\wedge$ M=6 $\wedge$ P=3 $\wedge$ Q=3 .

and so on, the process generates at step r

Cr: L=r!$\times$K $\wedge$ M=r! $\wedge$ P=r $\wedge$ Q=r .

The bottom-up evaluation loops forever, generating an infinity of independent formulas. We thus fail to compute the arithmetic reduction of factlist4(Y,K,L,Q) $\wedge$ factlist(Y,M,P).

In the case where the bottom-up evaluation does not terminate (as in the last example), it is sometimes possible to compute an *overestimation* of the arithmetic reduction of $p_1 \wedge ... \wedge p_n$. The method consists in applying projection to the fold/unfolded program so that not only the list argument but also some additional numeric arguments be eliminated. For such an "overprojected" program, the bottom-up evaluation process may then terminate. The resulting formula is an overestimation (of the reduction of) $p_1 \wedge ... \wedge p_n$ in the sense that it is implied by $p_1 \wedge ... \wedge p_n$ but the converse generally does not hold.

example 5.4

% newfact$'$(K,L,M,P,Q) $\leftrightarrow$ see example 5.3 .
% By further projection we eliminate the last two
% arguments and get

newfact$''$(K,L,M) $\leftarrow$ K=L, M=1, P=0, Q=0.
newfact$''$(K,L,M) $\leftarrow$ K$'$=N$\times$K, N=P, L=L$'$, M=N$\times$M$'$, N=Q, P=P$'$+1, Q=Q$'$+1, newfact$''$(K$'$,L$'$,M$'$).

It is easy to see that the subpart P=0 $\wedge$ Q=0 can be removed from the first clause body as well as the subpart N=P $\wedge$ N=Q $\wedge$ P=P$'$+1 $\wedge$ Q=Q$'$+1 from the second clause body. We then retrieve the program defining the predicate newproduct$'$ of example 5.2, for which we know that the bottom-up evaluation terminates and yields L=M$\times$K. Let g(K,L,M) abbreviate $\exists$Y,P,Q (factlist4(Y,K,L,Q) $\wedge$ factlist(Y,M,P)). The equation L=M$\times$K is an overestimation of factlist4(Y,K,L,Q) $\wedge$ factlist(Y,M,P) because g(K,L,M) implies L=M$\times$K, but not the converse. (Indeed: g(K,L,M) $\Leftrightarrow$ $\exists$R L=M$\times$K $\wedge$ M=R! .)

Note that the bottom-up evaluation process for a program Π computes the *least* fixpoint of the operator T associated with Π, and that the least fixpoint corresponding to an "overprojection" of Π is a fixpoint (not necessarily the least one) of T. From a general point of view, any arithmetic expression which is a fixpoint of Π provides an overestimation of the predicate defined by Π. Another terminology often used in program verification is to say that the arithmetic expression is an *inductive assertion* (or a *loop invariant*) for Π [20]. Bottom-up evaluation over an overprojected program can be regarded as a technique for automatically synthesizing inductive assertions (*cf.* [8]).

6 Recursion over two lists

So far we have only considered predicates recurring over one list argument. Our approach can be extended in order to consider predicates recurring over two (or more) list arguments. Several recursive schemes over lists can be introduced for two-list predicates. For each recursive scheme there are as many extensions as classes of one-list recursive predicates (ordered class, incremental class, arithmetic class). For the sake of conciseness, we will consider only one recursive scheme and focus on the class of incrementally recursive predicates.

definition 6.1

Let $\overline{X}$ be a vector of m distinct variables $X^1,\dots,X^m$ of type integer, Y,Z two variables of type list of integers, and $p(Y,Z,\overline{X})$ an atom. We say that $p(Y,Z,\overline{X})$ is *incrementally two-list recursive* iff p is defined by the following program Π

$$\{\ p([],Z,\overline{X}) \leftarrow a_i(Z,\overline{X}).\quad \}_{i\in I}$$
$$\{\ p(Y,[],\overline{X}) \leftarrow b_j(Y,\overline{X}).\quad \}_{j\in J}$$
$$\{\ p([N|Y],[M|Z],\overline{X}) \leftarrow c_k(\overline{X},N,M,\overline{X}'),\ p_k(Y,Z,\overline{X}'_k).\quad \}_{k\in K}$$
$$\{\ p([N|Y],[M|Z],\overline{X}) \leftarrow d_l(\overline{X},N,M,\overline{X}'),\ p(Y,Z,\overline{X}').\quad \}_{l\in L}$$

where

I,J,K and L are finite sets of indices, N,M are variables of type integer,

$\overline{X}'$ is a vector of m distinct variables $X'^1,\dots,X'^m$ of type integer,

$\overline{X}'_k$ is a subvector of $\overline{X}'$,

$p_k(Y,Z,\overline{X}'_k)$ either is incrementally one-list recursive (over Y or Z), or is incrementally two-list recursive and $p_k \ll p$ for some well-founded (two-list) predicate ordering $\ll$,

$a_i(Z,\overline{X})$ and $b_j(Y,\overline{X})$ are incrementally one-list recursive,

$c_k(\overline{X},N,M,\overline{X}')$ and $d_l(\overline{X},N,M,\overline{X}')$ are incrementation sets over $\overline{X}\cup\{N,M\}\cup\overline{X}'$.

Note that a one-list recursive predicate can be considered as a degenerate case of a two-list recursive predicate.

example 6.1

```
% match(Y,Z) ↔ the list Y is an initial
%                    subsequence of the list Z .
    match([],Z) .
    match([N|Y],[M|Z]) ← N=M, match(Y,Z).
```

The theorem 4.3 (and its preliminary definition) now becomes

definition 6.2

Let Y and Z be variables of type list of integers and $\overline{X}$ a vector of distinct integer variables. A formula $f(Y,Z,\overline{X})$ is said to be *periodic reducible* iff there exists a finite disjunction of constrained periodic sets $g(\overline{X})$ such that $g(\overline{X}) \Leftrightarrow \exists Y,Z\ f(Y,Z,\overline{X})$. The formula $g(\overline{X})$ is called a *periodic reduction* of $f(Y,Z,\overline{X})$.

Theorem 6.2

Let Y and Z be variables of type list of integers, $\overline{X}_1$, ..., $\overline{X}_n$ n pairwise disjoint vectors of distinct integer variables, and $p_1(Y,Z,\overline{X}_1)$, ..., $p_n(Y,Z,\overline{X}_n)$ n incrementally list-recursive atoms. There exists a procedure which, whenever it terminates, produces a periodic reduction of $p_1(Y,Z,\overline{X}_1)\wedge\ldots\wedge p_n(Y,Z,\overline{X}_n)$.

This is illustrated by the following example.

example 6.2

```
% nthmatch(Y,Z,I,J,K,L) ↔
%        match(Y,Z) ∧ nth(Y,I,K) ∧ nth(Z,J,L) .
% where
% nth(Y,I,K) means that K is the (I+1)-th
% element of Y.
    nth([N|Y],I,K) ← I=0, K=N.
    nth([N|Y],I,K) ← I=I'+1, K=K',
                               nth(Y,I',K').

% By folding/unfolding we get
    nthmatch([N|Y],[M|Z],I,J,K,L) ← N=M, I=0,
                           K=N, J=0, L=M.
    nthmatch([N|Y],[M|Z],I,J,K,L) ← N=M, I=0,
               K=N, J=J'+1, L=L', nth(Z,J',L').
    nthmatch([N|Y],[M|Z],I,J,K,L) ← N=M,
```

$I=I'+1,\ K=K',\ J=0,\ L=M,\ nth(Y,I',K').$

$nthmatch([N|Y],[M|Z],I,J,K,L) \leftarrow N=M,$
$I=I'+1,\ J=J'+1,\ K=K',\ L=L',$
$nthmatch(Y,Z,I',J',K',L').$

% By projection we get

$nthmatch'(I,J,K,L) \leftarrow N=M,\ I=0,\ K=N,\ J=0,\ L=M.$

$nthmatch'(I,J,K,L) \leftarrow N=M,\ I=0,\ K=N,\ J=J'+1,\ L=L',\ nth'(J',L').$

$nthmatch'(I,J,K,L) \leftarrow N=M,\ I=I'+1,\ K=K',\ J=0,\ L=M,\ nth'(I',K').$

$nthmatch'(I,J,K,L) \leftarrow N=M,\ I=I'+1,\ J=J'+1,\ K=K',\ L=L',\ nthmatch'(I',J',K',L').$

%It can be seen that $nth'(J',L') \leftrightarrow J' \geq 0$.
%Likewise: $nth'(I',K') \leftrightarrow I' \geq 0$.
%Therefore we get

$nthmatch'(I,J,K,L) \leftarrow N=M,\ I=0,\ K=N,\ J=0,\ L=M.$

$nthmatch'(I,J,K,L) \leftarrow N=M,\ I=0,\ K=N,\ J=J'+1,\ L=L',\ J' \geq 0.$

$nthmatch'(I,J,K,L) \leftarrow N=M,\ I=I'+1,\ K=K',\ J=0,\ L=M,\ I' \geq 0.$

$nthmatch'(I,J,K,L) \leftarrow N=M,\ I=I'+1,\ J=J'+1,\ K=K',\ L=L',\ nthmatch'(I',J',K',L').$

% By variable elimination we get

$nthmatch'(I,J,K,L) \leftarrow I=0,\ J=0,\ K=L.$
$nthmatch'(I,J,K,L) \leftarrow I=0,\ J>0.$
$nthmatch'(I,J,K,L) \leftarrow J=0,\ I>0.$
$nthmatch'(I,J,K,L) \leftarrow I=I'+1,\ J=J'+1,\ K=K',\ L=L',\ nthmatch'(I',J',K',L').$

Bottom-up evaluation then generates
$(I=J\geq 0 \wedge K=L) \vee J>I\geq 0 \vee I>J\geq 0$.
which is thus a periodic reduction of
$match(Y,Z) \wedge nth(Y,I,K) \wedge nth(Z,J,L)$.
Note that one can get a straightforward proof of theorem MATCH.EPSILON (see [3], p. 374) using the above reduction.

7 Final Remarks

We have presented a general method for reducing a formula over lists and integers to a formula of linear arithmetic. The method has two key parameters: first, the basic recursion scheme over lists; second, the class of Datalog programs with integers over which the bottom-up evaluation process runs.

We have only considered one basic list recursion scheme, which has been chosen both for its simplicity and its generality. More general list recursion schemes (e.g., some with multiple recursion) could be envisioned. The essential properties that a recursion scheme must satisfy is its preservation by conjunction and the existence of an equivalent list-free recursion scheme.

We have considered two basic classes of Datalog programs with integers: Revesz's class and Baudinet-Niézette-Wolper's one. These two classes can be viewed as two extremes: the first one ensures termination of our procedure but has little expressive power; the second one has great expressive power but offers us no guarantee of termination. An interesting topic of research is of course to characterize intermediate classes (e.g., syntactically restricted class of Baudinet-Niézette-Wolper programs with finite bottom-up evaluation). Such a research has been initiated for enriching databases with time representation (*cf.* [5, 19]). Our work gives a new motivation for this research: the design of decision procedures for proving correctness of programs with arrays.

Our initial goal was to reduce an assertion about lists and integers into an equivalent truth-decidable assertion about integers. Unfortunately our reduction process does not always terminate. A technique ("overprojection") has been indicated that improves process termination. The price to be paid is that the reduced formula is no longer equivalent to the original one ("overestimation"). This may be an interesting way of synthesizing inductive assertions of programs with no need for generalization heuristics (*cf.* [8, 20]). The technique bears also some resemblance to the geometrical approach of [13, 14] which treats linear constraint problems by computing a convex hull approximation in a projection space.

Acknowledgements.
This work was partly done while I was visiting Prof. Maluszynski's group LOGPRO at Linköping University. I would like to thank Ron van der Meyden for directing me to the reference [16] . This work has been partially supported by ESPRIT Project BRA 3020 "Integration".

References

[1] Baudinet, M., Niézette, M. and Wolper P. (1991). "On the Representation of Infinite Tem-

poral data and Queries", *Proc. Symp. on Principles of Database Systems*, Denver, pp. 280-290.

[2] Bledsoe, W.W. (1975). "A New Method for Proving certain Presburger Formulas", *Proc. 4th Intl. Joint Conf. on Artificial Intelligence*, Tbilissi, pp. 15-20.

[3] Boyer, R.S. and Moore, J S. (1979). *A Computational Logic*, ACM Monograph Series, Academic Press, New York.

[4] Boyer, R.S. and Moore, J S. (1988). "Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic", in *Machine Intelligence 11*, J.E. Hayes, D. Michie and J. Richards (eds.), Clarendon Press, Oxford, pp. 83-124.

[5] Chomicki, J. and Imieliński T. (1988). "Temporal Deductive Databases and Infinite Objects", *Proc. 7th ACM Symp. on Principles of Database Systems*, Austin, pp. 61-73.

[6] Clark, K.L. (1979). *Predicate Logic As a Computational Formalism*, Research Monograph:79/59, TOC, Imperial College, London.

[7] Cooper, D.C. (1972). "Theorem Proving in Arithmetic without Multiplication", in *Machine Intelligence 7*, B. Meltzer and D. Michie (eds.), American Elsevier, N.Y., pp. 91-99.

[8] Cousot, P. and Cousot, R. (1977). "Automatic Synthesis of Optimal Invariant Assertions: Mathematical Foundations", *Proc. ACM Symp. on Artificial Intelligence and Programming Languages*, Rochester, NY, *SIGPLAN Notices, Vol. 12, No 8*, pp. 1-12.

[9] van Emden, M.H. and Kowalski, R. (1976). "The Semantics of Predicate Logic As a Programming Language", *Journal of the ACM, Vol. 23, No 4*, pp. 733-742.

[10] Fribourg, L. (1990). "A New Presburger Arithmetic Decision Procedure Based on Extended Prolog Execution", *Programming Language Implementation and Logic Programming*, LNCS 456, Springer-Verlag, pp. 174-188.

[11] Fribourg, L. (1990). "Extracting Logic Programs from Proofs that Use Extended Prolog Execution and Induction", *Intl. Conf. on Logic Programming*, MIT Press, pp. 685-699.

[12] Jaffar, J. and Lassez, J.L. (1987). "Constraint Logic Programming", *Proc. 14th ACM Symp. on Principles of Programming Languages*, Muenchen, p. 111-119.

[13] Lassez, C. and Lassez, J.L. (1991). *Quantifier Elimination for Conjunction of Linear Constraints via a Convex Hull Algorithm*, IBM Research Report, T.J. Watson Research Center, New York.

[14] Lassez, J.L. (1991). "From LP to LP: Programming with Constraints", *Theoretical Aspects of Computer Software*, LNCS 526, Springer-Verlag, pp. 420-446.

[15] Pettorossi, A. and Proietti, M. (1989). "Decidability Results and Characterization of Strategies for the Development of Logic Programs", *Intl. Conf. on Logic Programming*, MIT Press, pp. 539-553.

[16] Revesz, P.Z. (1990). "A Closed Form for Datalog Queries with Integer Order", *Intl. Conf. on Database Theory*, LNCS 470, Springer-Verlag, pp. 187-201.

[17] Shostack, R. (1977). "On the SUP-INF Method for Proving Presburger Formulas", *Journal of the ACM, Vol. 24, No 4*, pp. 529-543.

[18] Tamaki, H. and Sato, T. (1984). "Unfold/Fold Transformations of Logic Programs", *Proc. 2nd Intl. Conf. on Logic Programming*, Uppsala, pp. 127-138.

[19] Vardi, M.Y. (1988). "A Temporal Fixpoint Calculus", *Proc. 15th ACM Symp. on Principles of Programming Languages*, San Diego, pp. 250-259.

[20] Wegbreit, B. (1977). "Complexity of Synthesizing Inductive Assertions", *Journal of the ACM, Vol. 24, No 3*, pp. 504-512.