A Complete Method for the Synthesis of Linear Ranking Functions

Andreas Podelski and Andrey Rybalchenko

Max-Planck-Institut für Informatik
Saarbrücken, Germany

Abstract. We present an automated method for proving the termination of an unnested program loop by synthesizing linear ranking functions. The method is complete. Namely, if a linear ranking function exists then it will be discovered by our method. The method relies on the fact that we can obtain the linear ranking functions of the program loop as the solutions of a system of linear inequalities that we derive from the program loop. The method is used as a subroutine in a method for proving termination and other liveness properties of more general programs via transition invariants; see [PR03].

1 Introduction

The verification of termination and other liveness properties of programs is a difficult problem. It requires the discovery of invariants and ranking functions to prove the termination of program loops.

We present a complete and efficient method for the synthesis of linear ranking functions for unnested program loops whose guards and update statements use linear arithmetic expressions. We have implemented the method. Preliminary experiments show that the method is efficient not only in theory but also in practice.

Roughly, the method works as follows. Given a program loop for which we want to find a linear ranking function, we construct a corresponding system of linear inequalities over rationals. As we show, the solutions of this system encode the linear ranking functions of the program loop. That is, we can check the existence of a linear ranking function by constraint solving. If it exists, a linear ranking function can be constructed from a solution of the system of linear inequalities, a solution that we obtain by constraint solving. If the system has no solutions then (and only then) a linear ranking function does not exist. As a consequence of our approach, one can use existing highly-optimized tools for linear programming as the engine in a complete method (to our knowledge the first) for the synthesis of linear ranking functions.

We admit unnested program loops with nondeterministic update statements. This is potentially useful to model `read` statements. It is strictly required in the context where we employ our method, described next.

In a work described elsewhere [PR03], we show that one can reduce the test of termination and other liveness properties (in the presence of fairness

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 239–251, 2004.
© Springer-Verlag Berlin Heidelberg 2004

assumptions) to the test of termination of unnested program loops. That is, we use the algorithm described in this paper as a subroutine in the software model checking method for liveness properties via transition invariants proposed in [PR03]. The experiments that we present in this paper stem from this context.

2 Unnested Program Loops

We formalize the notion of unnested program loops by a class of programs that are built using a single "while" statement and that satisfy the following conditions:

- the loop condition is a conjunction of atomic propositions,
- the loop body may only contain update statements,
- all update statements are executed simultaneously.

We call this class *simple while programs*. Pseudo-code notation for the programs of this class is given below.

while $(Cond_1$ **and** $\ldots$ **and** $Cond_m)$ **do**
    Simultaneous Updates
od

We consider the subclass of simple while programs built using linear arithmetic expressions over program variables.

Definition 1. *A linear arithmetic simple while (LASW) program over the tuple of program variables $x = (x_1, \ldots, x_n)$ is a simple while program such that:*

- *program variables have integer domain,*
- *every atomic proposition in the loop condition is a linear inequality over (unprimed) program variables:*

$$c_1 x_1 + \cdots + c_n x_n \leq c_0,$$

- *every update statement is a linear inequality over unprimed and primed program variables*

$$a'_1 x'_1 + \cdots + a'_n x'_n \leq a_1 x_1 + \cdots + a_n x_n + a_0.$$

Note that we allow the left-hand side of an update statement to be a linear expression over program variables, and that an update can be nondeterministic, e.g., $x' + y' \leq x + 2y - 1$. This is necessary, because we use simple while programs, and LASW programs in particular, to approximate the transitive closure of a transition relation (see Section 4).

We define a *program state* to be a valuation of program variables. The set of all program states is called the *program domain*. The *transition relation* denoted by the loop body of an LASW program is the set of all pairs of program states (s, s') such that the state s satisfies the loop condition, and (s, s') satisfies

each update statement. A *trace* is a sequence of states such that each pair of consecutive states belongs to the transition relation of the loop body.

We observe that the transition relation of a LASW program can be expressed by a system of inequalities over unprimed and primed program variables. The translation procedure is straightforward. For the rest of this paper, we assume that an LASW program over the tuple of program variables $x = (x_1, \ldots, x_n)$ (treated as a column vector) can be represented by the system

$$(AA')\begin{pmatrix} x \\ x' \end{pmatrix} \leq b$$

of inequalities. We identify an LASW program with the corresponding system of inequalities.

Example 1. The following program loop with nondeterministic updates

while $(i - j \geq 1)$ **do**
$\quad (i, j) := (i - Nat, j + Pos)$
od

is represented by the following system of inequalities.

$$\begin{aligned} -i + j &\leq -1 \\ -i + i' &\leq 0 \\ j - j' &\leq -1 \end{aligned}$$

Note that the relations between program variables denoted by the nondeterministic update statements $i := i - Nat$ and $j := j + Pos$, where Nat and Pos stand for any nonnegative and positive integer number respectively, can be expressed by the inequalities $i' \leq i$ and $j' \geq j + 1$.

3 The Algorithm

We say that a simple while program is *terminating* if the program domain is well-founded by the transition relation of the loop body of the program, i.e., if there is no infinite sequence $\{s_i\}_{i=1}^{\infty}$ of program states such that each pair (s_i, s_{i+1}), where $i \geq 1$, is an element of the transition relation.

The following theorem allows us to use linear programming over rationals to test the existence of a linear ranking function, and thus to test a sufficient condition for termination of LASW programs. The corresponding algorithm is shown in Figure 1.

Theorem 1. *A linear arithmetic simple while program given by the system* $(AA')\binom{x}{x'} \leq b$ *is terminating if there exist nonnegative vectors over rationals*

input
program $(AA')\binom{x}{x'} \leq b$
begin
if exists rational-valued λ_1 **and** λ_2 **such that**
$\lambda_1, \lambda_2 \geq 0$
$\lambda_1 A' = 0$
$(\lambda_1 - \lambda_2)A = 0$
$\lambda_2(A + A') = 0$
$\lambda_2 b < 0$
then
return("Program Terminates")
else
return("Linear ranking function does not exist")
end.

Given λ_1 and λ_2, solutions of the systems above, define $r \stackrel{\text{def}}{=} \lambda_2 A'$, $\delta_0 \stackrel{\text{def}}{=} -\lambda_1 b$, and $\delta \stackrel{\text{def}}{=} -\lambda_2 b$. A linear ranking function ρ is defined by

$$\rho(x) \stackrel{\text{def}}{=} \begin{cases} rx & \text{if exists } x' \text{ such that } (AA')\binom{x}{x'} \leq b, \\ \delta_0 - \delta & \text{otherwise.} \end{cases}$$

Fig. 1. Termination Test and Synthesis of Linear Ranking Functions.

λ_1 and λ_2 such that the following system is satisfiable.

$$\lambda_1 A' = 0 \tag{1a}$$
$$(\lambda_1 - \lambda_2)A = 0 \tag{1b}$$
$$\lambda_2(A + A') = 0 \tag{1c}$$
$$\lambda_2 b < 0 \tag{1d}$$

Proof. Let the pair of nonnegative (row) vectors λ_1 and λ_2 be a solution of the system (1a)–(1d). For every x and x' such that $(AA')\binom{x}{x'} \leq b$, by assumption that $\lambda_1 \geq 0$, we have $\lambda_1(AA')\binom{x}{x'} \leq \lambda_1 b$. We carry out the following sequence of transformations.

$$\begin{aligned} \lambda_1(Ax + A'x') &\leq \lambda_1 b \\ \lambda_1 Ax + \lambda_1 A'x' &\leq \lambda_1 b \\ \lambda_1 Ax &\leq \lambda_1 b && \text{by (1a)} \\ \lambda_2 Ax &\leq \lambda_1 b && \text{by (1b)} \\ -\lambda_2 A'x &\leq \lambda_1 b && \text{by (1c)} \end{aligned}$$

From the assumption $\lambda_2 \geq 0$ follows $\lambda_2(AA')\binom{x}{x'} \leq \lambda_2 b$. Then, we continue with

$$\begin{aligned} \lambda_2(Ax + A'x') &\leq \lambda_2 b \\ \lambda_2 Ax + \lambda_2 A'x' &\leq \lambda_2 b \\ -\lambda_2 A'x + \lambda_2 A'x' &\leq \lambda_2 b \qquad \text{by (1c)} \end{aligned}$$

We define $r \stackrel{\text{def}}{=} \lambda_2 A'$, $\delta_0 \stackrel{\text{def}}{=} -\lambda_1 b$, and $\delta \stackrel{\text{def}}{=} -\lambda_2 b$. Then, we have $rx \geq \delta_0$ and $rx' \leq rx - \delta$ for all x and x' such that $(AA')\binom{x}{x'} \leq b$. Due to (1d) we have $\delta > 0$.

We define a function ρ as shown in Figure 1. Any program trace induces a strictly descending sequence of values under ρ that is bounded from below, and the difference between two consecutive values is at least δ. Since no such infinite sequence exists, the program is terminating. □

The theorem above states a sufficient condition for termination. We observe that if the condition applies then a linear ranking function, i.e., a linear arithmetic expression over program variables which maps program states into a well-founded domain, exists. The following theorem states that our termination test is complete for the programs with linear ranking functions.

Theorem 2. *If there exists a linear ranking function for the linear arithmetic simple while program with nonempty transition relation then the termination condition of Theorem 1 applies.*

Proof. Let the vector r together with the constants δ_0 and $\delta > 0$ define a linear ranking function. Then, for all pairs x and x' such that $(AA')\binom{x}{x'} \leq b$ we have $rx \geq \delta_0$ and $rx' \leq rx - \delta$.

By the non-emptiness of the transition relation, the system $(AA')\binom{x}{x'} \leq b$ has at least one solution. Hence, we can apply the 'affine' form of Farkas' lemma (in [Sch86]), from which follows that there exists δ_0' and δ' such that $\delta_0' \geq \delta_0$, $\delta' \geq \delta$, and each of the inequalities $-rx \leq -\delta_0'$ and $-rx + rx' \leq -\delta'$ is a nonnegative linear combination of the inequalities of the system $(AA')\binom{x}{x'} \leq b$. This means that there exist nonnegative rational-valued vectors λ_1 and λ_2 such that

$$\begin{aligned} \lambda_1(AA')\tbinom{x}{x'} &= -rx \\ \lambda_1 b &= -\delta_0' \end{aligned}$$

and

$$\begin{aligned} \lambda_2(AA')\tbinom{x}{x'} &= -rx + rx' \\ \lambda_2 b &= -\delta'. \end{aligned}$$

After multiplication and simplification we obtain

$$\begin{aligned} \lambda_1 A &= -r & \lambda_1 A' &= 0 \\ \lambda_2 A &= -r & \lambda_2 A' &= r, \end{aligned}$$

from which equations (1a)–(1c) follow directly. Since $\delta' \geq \delta > 0$, we have $\lambda_2 b < 0$, i.e., the equation (1d) holds. □

The following corollary is an immediate consequence of Theorems 1 and 2.

Corollary 1. *Existence of linear ranking functions for linear arithmetic simple while programs with nonempty transition relation is decidable in polynomial time.*

Not every LASW program has a linear ranking function (see the following example).

Example 2. Consider the following program.

$$\begin{array}{l}\textbf{while } (x \geq 0) \textbf{ do}\\ \quad x := -2x + 10\\ \textbf{od}\end{array}$$

The program is terminating, but it does not have a linear ranking function. For termination proof consider the following ranking function into the domain $\{0, \ldots, 3\}$ well-founded by the less-than relation $<$.

$$\rho(x) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x \in \{0,1,2\},\\ 2 & \text{if } x \in \{4,5\},\\ 3 & \text{if } x = 3,\\ 0 & \text{otherwise.}\end{cases}$$

It can be easily tested that the system (1a)–(1d) is not satisfiable for the LASW program

$$\begin{pmatrix} -1 & 0\\ 2 & 1\\ -2 & -1\end{pmatrix}\begin{pmatrix} x\\ x'\end{pmatrix} \leq \begin{pmatrix} 0\\ 10\\ -10\end{pmatrix}.$$

By Theorem 2, this implies that no linear ranking function exists for the program above. □

The following example illustrates an application of the algorithm based on Theorem 1.

Example 3. We prove termination of the LASW program from Example 1. The program translates to the system $(AA')\binom{x}{x'} \leq b$, where:

$$A \stackrel{\text{def}}{=} \begin{pmatrix} -1 & 1\\ -1 & 0\\ 0 & 1\end{pmatrix}, \qquad A' \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 0\\ 1 & 0\\ 0 & -1\end{pmatrix},$$

$$x \stackrel{\text{def}}{=} \begin{pmatrix} i\\ j\end{pmatrix}, \qquad b \stackrel{\text{def}}{=} \begin{pmatrix} -1\\ 0\\ -1\end{pmatrix}.$$

Let $\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3)$ and $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3)$. The system (1a)–(1d) is feasible, it has the following solutions:

$$\begin{aligned} \lambda'_2 = \lambda'_3 = \lambda''_1 = 0, \\ \lambda'_1 = \lambda''_2 = \lambda''_3, \\ \lambda'_1, \lambda''_2, \lambda''_3 > 0. \end{aligned}$$

Since the system is feasible the program is terminating. We construct a linear ranking function following the algorithm in Figure 1. We define $r \stackrel{\text{def}}{=} \lambda_2 A'$, $\delta_0 \stackrel{\text{def}}{=} -\lambda_1 b$, and $\delta \stackrel{\text{def}}{=} -\lambda_2 b$, and obtain $r = (\lambda'_1 \ \ -\lambda'_1)$, $\delta_0 = \delta = \lambda'_1$. Taking $\lambda'_1 = 1$ we obtain the following ranking function.

$$\rho(i,j) = \begin{cases} i - j & \text{if } i - j \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

4 Application to General Programs

In this section we illustrate how our method for proving termination of program loops can be used in the software model checking method for liveness properties via transition invariants proposed in [PR03]. That method applies to general-purpose programs (imperative, concurrent, ...); it is different from other approaches to special classes of infinite-state systems, e.g. [BS99]. We then provide experimental results obtained by applying the transition invariants approach for proving termination of singular value decomposition program.

Software model checking for liveness properties is a new approach for the automated verification of liveness properties of infinite-state systems by the computation of *transition invariants*. A transition invariant is an over-approximation of the transitive closure of the transition relation of the system. The presentation of a transition invariant as nothing but a finite set of unnested program loops. One can characterize the validity of a liveness property via the existence of transition invariants [PR03]. Namely, the liveness property is valid if each of the unnested program loops is terminating.

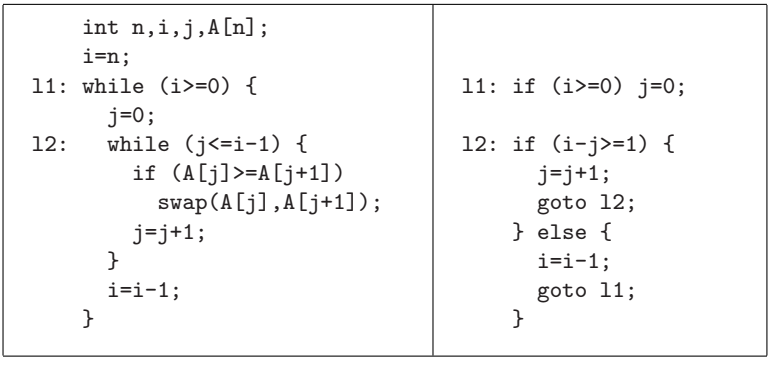
That is, the general method for the verification of liveness properties described in [PR03] is parameterized by an algorithm that tests whether each unnested program loop in the transition invariant is terminating. i.e., a procedure implementing a termination test for simple while programs.

Proving termination of simple while programs built using linear arithmetic expressions is required for the verification of a large class of software systems, e.g., liveness properties for mutual exclusion protocols (bakery, ticket), termination proofs of imperative programs (sorting algorithms, numerical algorithms dealing with matrices).

4.1 Sorting Program

This example illustrates the approach from [PR03] and the role of simple while programs.

We consider the program shown in Figure 2 implementing a sorting algorithm. For legibility, we concentrate on the skeleton shown on the right, which



```
    int n,i,j,A[n];
    i=n;
l1: while (i>=0) {
      j=0;
l2:   while (j<=i-1) {
        if (A[j]>=A[j+1])
          swap(A[j],A[j+1]);
        j=j+1;
      }
      i=i-1;
    }
```

```
l1: if (i>=0) j=0;

l2: if (i-j>=1) {
      j=j+1;
      goto l2;
    } else {
      i=i-1;
      goto l1;
    }
```

Fig. 2. Sorting program and its skeleton.

consists of the statements `st1`, `st2`, `st3`.

```
l1: if (i>=0)   { (i,j):=(i,0);   goto l2; }    /* st1 */
l2: if (i-j>=1) { (i,j):=(i,j+1); goto l2; }    /* st2 */
l2: if (i-j<1)  { (i,j):=(i-1,j); goto l1; }    /* st3 */
```

We read, for example, the first program statement as: if the current program location is labeled by `l1` and the "`if`" condition is satisfied then update the variables according to the update expressions and change the current label label to `l2`. Note that the updates are performed simultaneously ("concurrent" assignments in [Dij76]).

Each of the 'simple' programs below must be read as a one-line program.

```
l1: if (true)   { (i,j):=(Any,Any);      goto l2; }   /* a1 */
l2: if (true)   { (i,j):=(Any,Any);      goto l1; }   /* a2 */
l1: if (i>=0)   { (i,j):=(i-Pos,Any);    goto l1; }   /* a3 */
l2: if (i>=0)   { (i,j):=(i-Pos,Any);    goto l2; }   /* a4 */
l2: if (i-j>=1) { (i,j):=(i-Nat,j+Pos); goto l2; }   /* a5 */
```

Note the nondeterministic update expressions, e.g., after execution of `i:=Any` the value of variable `i` could by any integer, the update `i:=i-Pos` decrements the value of `i` by at least one.

We notice that `st1` is approximated by `a1`, `st2` by `a5` and `st3` by `a2`. This means that every transition induced by execution of the statement `st1` can also be achieved executing a single step of `a1`. In fact, every sequence of program statements is approximated by one of `a1`, ..., `a5`. We say that the set {`a1`, ..., `a5`} is a transition invariant in our terminology.

For example, every sequence of program statements that leads from `l2` to `l2` is approximated by `a4` if it passes through `l1`, and by `a5` otherwise. The following table assigns to each 'simple' program the set of sequences of program statements that it approximates. All non-assigned sequences are not feasible.

a1	st1(st2|st3st1)*
a2	(st2|st3st1)*st3
a3	st1(st2|st3st1)*st3
a4	(st2|st3st1)*st3st1(st2|st3st1)*
a5	st2+

According to the formal development in [PR03], the transition invariant above is 'strong enough' to prove termination, which means: each of its 'simple' programs, viewed in isolation, is terminating.

Termination is obvious for 'simple' programs that do not refer to a loop in the control flow graph, like the 'simple' programs `a1` and `a2`. The 'simple' programs of the form

```
ln: if (cond) { updates;  goto ln; }
```

translate to a while loop

```
ln: while (cond) { updates; }.
```

The 'simple' programs that translate to a while loop are in fact simple while programs whose termination proofs we study in this paper.

Next, we describe an application of the transition invariants method with termination test in Figure 1 as a subroutine for checking whether a transition invariant is strong enough. We prove termination of a program implementing singular value decomposition algorithm.

4.2 Program with Unbounded Nondeterminism

We consider the program shown in Figure 3. It has a nondeterministic choice at the location labeled by `l`. The value of the variable y is chosen nondeterministically in the first branch. Termination proof for this program requires a lexicographic ranking function. The program translates to the statements `st1` and `st2`:

```
l: if (x>=0)   { (x,y):=(x-1,Any);   goto l; }   /* st1 */
l: if (y>=0)   { (x,y):=(x,y-1);     goto l; }   /* st2 */
```

The transition invariant computed by our tool consists of the following 'simple' programs.

```
l: if (x>=0)   { (x,y):=(x-Pos,Any);   goto l; }   /* a1 */
l: if (y>=0)   { (x,y):=(Any,y-Pos);   goto l; }   /* a2 */
```

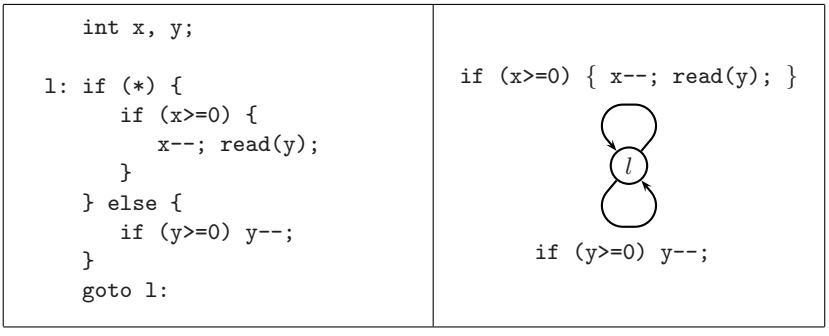


Fig. 3. Program with unbounded nondeterminism.

Both 'simple' programs, viewed in isolation, are terminating. Hence, the program with unbounded nondeterminism, shown in Figure 3 is terminating.

4.3 Singular Value Decomposition Program

We considered an algorithm for constructing the singular value decomposition (SVD) of a matrix. SVD is a set of techniques for dealing with sets of equations or matrices that are either singular or numerically very close to singular [PTVF92]. A matrix A is singular if it does not have a matrix inverse A^{-1} such that $AA^{-1} = I$, where I is the identity matrix.

Singular value decomposition of the matrix A whose number of rows m is greater or equal to its number of columns n is of the form

$$A = UWV^T,$$

where U is an $m \times n$ column-orthogonal matrix, W is an $n \times n$ diagonal matrix with positive or zero elements (called singular values), and the transpose matrix of an $n \times n$ orthogonal matrix V. Orthogonality of the matrices U and V means that their columns are orthogonal, i.e.,

$$U^T U = VV^T = I.$$

The SVD decomposition always exists, and is unique up to permutation of the columns of U, elements of W and columns of V, or taking linear combinations of any columns of U and V whose corresponding elements of W are exactly equal.

SVD can be used in numerically difficult cases for solving sets of equations, constructing an orthogonal basis of a vector space, or for matrix approximation [PTVF92].

We proved termination of a program implementing the SVD algorithm based on a routine described in [GVL96]. The program was taken from [PTVF92]. It is written in C and contains 163 lines of code with 42 loops in the control-flow graph, nested up to 4 levels.

We used our transition invariant generator to compute a transition invariant for the SVD program. Proving the transition invariant to be strong enough required testing termination of 219 LASW programs.

We applied our implementation of the algorithm on Figure 1, which was done in SICStus Prolog [Lab01] using the built-in constraint solver for linear arithmetic [Hol95]. Proving termination required 800 ms on a 2.6 GHz Xeon computer running Linux, which is in average 3.6 ms per each LASW program.

5 Related Work

The verification of termination and other liveness properties of programs requires the discovery of *invariants* as well as of *ranking functions* to prove the termination of program loops. Here, we relate our work not to methods for the automated discovery of invariants (see e.g. [Kar76,CH78,BBM97]), but to the more closely related topic of methods for the automated synthesis of ranking functions, a topic that has received increasing attention in the last years [GCGL02,CS01, DGG00,Mes96,MN01,CS02,SG91].

As a first general remark, a major difference between our work and all the others lies in the fact that we obtain a completeness result.

A heuristic-based approach for discovery of ranking functions is described in [DGG00]. It inspects the program source code for ranking function candidates. This method restricted to programs where the ranking function is exhibited already in the source code.

The algorithm in [CS01] extracts a linear ranking function of an unnested program loop by manipulating polyhedral cones; these represent the transition relation of the loop and the loop invariant. Their approach depends on the strength of the invariant generator, which they call in a subroutine to propose bounded linear arithmetic expression. The algorithm requires exponential space in the worst case. A generalization of that algorithm described in [CS02] for programs with complex control structures detects linear ranking functions for strongly connected components in the control-flow graph of more general programs. In both cases the algorithm is restricted to bounded nondeterminism. Moreover, it cannot handle loops with non-monotonic decrease, such as in `while (x>=0) {x=x+1; x=x-1;}`.

The method for discovery of nonnegative linear combinations of bound argument sizes for proving termination of logic programs in [SG91] relies on automatically inferred inter-argument constraints. The duality theory of linear programming is applied to discover combinations that decrease during top-down execution of recursive rules; the determined combinations are bounded from below since argument sizes are always positive. This method was applied for inferring termination of constraint logic programs [Mes96], and in systems for inferring termination of logic programs [MN01,GCGL02]. Carried over into the context of imperative program loops, the inference of inter-argument constraints corresponds to calls to the invariant generator, as in [CS01]; the same restrictions as mentioned above apply.

6 Conclusion

We have presented the to our knowledge first complete algorithm for the synthesis of linear ranking functions for a small but natural and well-motivated class of programs, namely, unnested program loops built using linear arithmetic expressions (LASW programs). The method is guaranteed to find a linear ranking function, and therefore to prove termination, if a linear ranking function exists. The existence of a linear ranking function for an LASW program is equivalent to the satisfiability of the system of linear inequalities derived from the program.

The termination check for LASW programs is a subroutine in the automated method for the verification of termination and other liveness properties of general-purpose programs via the computation of transition invariants [PR03].

We have implemented the proposed algorithm using an efficient implementation of a solver for linear programming over rationals [Hol95]. We applied our implementation to prove termination of a singular value decomposition program, which required termination proofs for 219 LASW programs. This and other experiments indicate the practical potention of the algorithm.

Considering future work, we would like to find a characterization of LASW programs that do always have linear ranking functions, i.e., for which our algorithm decides termination. Another direction of work is to handle unnested program loops built using expressions other than linear arithmetic.

Acknowledgments. We thank Bernd Finkbeiner, Konstantin Korovin, and Uwe Waldmann for comments on this paper. We thank Ramesh Kumar for his help with the SVD example.

References

[BBM97] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BS99] Olaf Burkart and Bernhard Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL 1978: Symp. on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.

[CS01] Michael Colon and Henny Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *Proc. of TACAS 2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81. Springer-Verlag, 2001.

[CS02] Michael Colon and Henny Sipma. Practical methods for proving program termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. of CAV 2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 442–454. Springer-Verlag, 2002.

[DGG00] Dennis Dams, Rob Gerth, and Orna Grumberg. A heuristic for the automatic generation of ranking functions. In *Workshop on Advances in Verification (WAVe'00)*, pages 1–8, 2000.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, Englewood Cliffs, NJ, 1976.

[GCGL02] Samir Genaim, Michael Codish, John P. Gallagher, and Vitaly Lagoon. Combining norms to prove termination. In Agostino Cortesi, editor, *Proc. of VMCAI 2002: Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *LNCS*, pages 126–138. Springer-Verlag, 2002.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations; 3rd edition*. Johns Hopkins Univ Press, 3rd edition, 1996.

[Hol95] Christian Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[Lab01] The Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, PO Box 1263 SE-164 29 Kista, Sweden, October 2001. Release 3.8.7.

[Mes96] Fred Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In Michael J. Maher, editor, *Proc. of JICSLP 1996: Joint Int. Conf. and Symp. on Logic Programming*, pages 7–21. MIT Press, 1996.

[MN01] Fred Mesnard and Ulrich Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In Patrick Cousot, editor, *Proc. of SAS 2001: Symp. on Static Analysis*, volume 2126 of *LNCS*, pages 93–110. Springer-Verlag, 2001.

[PR03] Andreas Podelski and Andrey Rybalchenko. Software model checking of liveness properties via transition invariants. Technical report, Max-Plank-Institut für Informatik, 2003.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P . Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., 1986.

[SG91] Kirack Sohn and Allen Van Gelder. Termination detection in logic programs using argument sizes. In *Proc. of PODS 1991: Symp. on Principles of Database Systems*, pages 216–226. ACM Press, 1991.