Practical Methods for Proving Program Termination

Michael A. Colón and Henny B. Sipma [*]

Computer Science Department
Stanford University
Stanford, CA 94305-9045
{colon,sipma}@cs.stanford.edu

Abstract. We present two algorithms to prove termination of programs by synthesizing linear ranking functions. The first uses an invariant generator based on iterative forward propagation with widening and extracts ranking functions from the generated invariants by manipulating polyhedral cones. It is capable of finding subtle ranking functions which are linear combinations of many program variables, but is limited to programs with few variables.
The second, more heuristic, algorithm targets the class of structured programs with single-variable ranking functions. Its invariant generator uses a heuristic extrapolation operator to avoid iterative forward propagation over program loops. For the programs we have considered, this approach converges faster and the invariants it discovers are sufficiently strong to imply the existence of ranking functions.

1 Introduction

Proving total program correctness consists of two tasks: proving partial correctness, that is, proving that the relation between inputs and outputs satisfies its specification, and proving termination. While many techniques have been proposed to automatically establish termination of term rewriting systems [9], logic programs [15] and functional programs [12], the problem of demonstrating termination of imperative programs has received much less attention. Clearly, one possible strategy is to reduce the problem to one of the well-studied cases by program transformation, but such an approach often introduces additional complexities. For example, when translating an imperative program into a functional program, the recursive functions introduced to encode loops often fail to terminate for all inputs, necessitating the development of automatic methods to approximate the domains of these functions [2].

In [4], we presented a method for generating linear ranking functions to prove termination of program loops. The approach taken was to represent program invariants and transition relations as polyhedral cones and to construct

[*] This research was supported in part by NSF(ITR) grant CCR-01-21403, by NSF grant CCR-99-00984-001, by ARO grant DAAD19-01-1-0723, and by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014.

linear ranking functions by manipulating these cones. The method made the tacit assumption that the loops are unnested. As such, it could generate complex ranking functions for simple control structures, but was unable to construct even simple ranking functions for complex structures. Furthermore, it could not establish termination of unnested loops with multiple paths in which different paths require different ranking functions, i.e., loops which can be proved terminating using lexicographic ranking functions all of whose components are linear. The aim of the present work is to generalize the approach and address these shortcomings.

In this paper we propose two algorithms for proving termination of programs. The first is capable of finding complex linear ranking functions to prove termination, and uses iterative forward propagation with widening to derive invariants to justify the ranking functions. The second is more heuristic in nature, but is faster for structured programs and is capable of handling larger programs.

2 Preliminaries

2.1 Program representation and Computational model

We present programs using SPL (Simple Programming Language) [14], a Pascal-like programming language with well-defined semantics. As an underlying computational model we use transition systems, a flexible first-order representation of programs.

A program $P : \langle V, L, \mathcal{T}, L_0, \Theta \rangle$ consists of the following components:

- V, a finite set of program variables; a *state* σ is an assignment to all variables in V. An *assertion* is a first-order formula over V.
- L, a finite set of program *locations*.
- $\mathcal{T}$, a finite set of *transitions*, where each transition τ is represented by a triple $\langle \ell, \ell', \rho \rangle$ consisting of a *prelocation* ℓ, a *postlocation* ℓ', and an assertion ρ over V and V', where V' denote the variables in the next state. By $\mathrm{post}(\rho, \varphi)$ we denote the *postcondition* of τ with respect to an assertion φ, which is defined as the set of states reachable by taking transition τ from a state satisfying φ, that is, the set of states satisfying $\exists V_0 \,.\, \rho(V_0, V) \wedge \varphi(V_0)$.
- $L_0 \subseteq L$, the set of *initial locations*.
- Θ, an assertion characterizing the *initial states*.

A *configuration* is a pair $\langle \ell, \sigma \rangle$ consisting of a location and a state. A *computation* π is a potentially infinite sequence $\langle \ell_0, \sigma_0 \rangle, \langle \ell_1, \sigma_1 \rangle, \ldots$ of configurations such that ℓ_0 is an initial location, $\sigma_0 \models \Theta$, and for each adjacent pair of configurations $\langle \ell_i, \sigma_i \rangle$ and $\langle \ell_{i+1}, \sigma_{i+1} \rangle$, there exists a transition $\tau = \langle \ell_i, \ell_{i+1}, \rho \rangle$ such that $\sigma_i, \sigma_{i+1} \models \rho$.

2.2 Flow graphs

Flow graphs are convenient for representing the control structure of programs. Recall that a *directed graph* $G = \langle V, E \rangle$ consists of a finite set of *vertices* V and a finite set of *edges* E, where each edge of E is a pair $\langle v, v' \rangle$ of vertices of V. A *path* of G is a potentially infinite sequence $v_1, v_2, \ldots$ of vertices such that for all $i > 0$, $\langle v_i, v_{i+1} \rangle \in E$. A finite path $v_1, \ldots, v_n$ is a *cycle* if $v_1 = v_n$. A graph is *acyclic* if none of its paths are cycles and *strongly-connected* if there is a path between every pair of vertices. A *strongly-connected subgraph* (SCS) of G is a subgraph of G which is strongly connected. An SCS S is a *maximal* SCS (MSCS) if it is not a proper subgraph of an SCS of G. An MSCS consisting of a single vertex and no edges is said to be *trivial.*

A *flow graph* is a directed graph in which a subset of the vertices are distinguished as being initial. A path of a flow graph is said to be *proper* if its first vertex is initial. The *control flow graph* (CFG) of a program is the flow graph whose vertices are the control locations and which contains an edge $\langle \ell, \ell' \rangle$ for each transition $\tau = \langle \ell, \ell', \rho \rangle$. The CFG of a program can be viewed as an abstraction of the program in which all variables are ignored: each computation of the program induces a path in the flow graph. However, not all paths of the flow graph correspond to computations of the program. Invariants and ranking functions are needed to refine this abstraction.

2.3 Ranking functions

A binary relation $\prec$ is called *well-founded* over a domain $\mathcal{D}$ if there is no infinite descending chain, that is, no infinite sequence of elements $d_0, d_1, d_2, \ldots$ of $\mathcal{D}$ such that $d_i \succ d_{i+1}$ for all $i \geq 0$. The most commonly used well-founded domain is that of the natural numbers with the $>$ relation.

Well-founded relations can be used to show that a certain set of program locations cannot be visited infinitely often. Let π be an infinite computation of a program P. Since there are only finitely many locations, π must visit some subset of the locations infinitely often. This subset must be an SCS of the CFG of P. To prove termination of the program, then, it suffices to show that no SCS of the CFG can be visited infinitely often. To do so, we exhibit a ranking function for each SCS of the CFG.

A *ranking function* δ for an SCS S is a mapping from program states into a well-founded domain such that no transition associated with an edge of S increases the measure assigned by δ, and some transition decreases it. Thus the existence of a ranking function for S implies that any infinite computation can take the decreasing transitions only finitely many times, and therefore, if it remains within S, it must eventually confine itself to a proper sub-SCS of S which does not contain the decreasing transitions. If removal of these decreasing transitions from the SCS results in an acyclic graph, the SCS admits no infinite computations. If for each SCS of the CFG we can find a ranking function, we have shown that the program terminates.

2.4 Invariant assertions

An assertion $\mathcal{I}$ is said to be *invariant* at location ℓ of a program P if, for any computation π of P, $\mathcal{I}$ holds whenever π reaches ℓ. An *invariant map* μ is any assignment of assertions to the locations of P such that $\mu(\ell)$ is invariant at ℓ. An invariant map μ is said to be *inductive* if, for every transition $\tau = \langle \ell, \ell', \rho \rangle$, $\text{post}(\rho, \mu(\ell)) \models \mu(\ell')$. Thus inductive invariant maps can be verified locally, provided the assertion language is decidable.

It is often possible to compute non-trivial invariants for a program by iterative forward propagation in an abstract domain, a method known as *abstract interpretation* [6]. Given a program P and a map μ, the operator $\mathcal{F}$ is defined as follows:

$$\mathcal{F}(\mu, P) = \bigcup_{\ell \in L} \{\ell \mapsto \mathcal{G}(\mu, \ell)\}.$$

with

$$\mathcal{G}(\mu, \ell') = \begin{cases} \bigsqcup_{\langle \ell, \ell', \rho \rangle \in \mathcal{T}} \text{post}(\rho, \mu(\ell)) \sqcup \Theta & \text{if } \ell' \text{ is initial,} \\ \bigsqcup_{\langle \ell, \ell', \rho \rangle \in \mathcal{T}} \text{post}(\rho, \mu(\ell)) & \text{otherwise} \end{cases}$$

Iterative application of $\mathcal{F}$ to an initial map assigning *false* to each location yields the strongest map of inductive invariants expressible in the given assertion language, provided it converges. If the assertion language contains infinite ascending chains, a heuristic *widening* operator is employed to ensure convergence for programs containing loops. The use of abstract interpretation to generate invariants expressible as systems of linear constraints was first proposed in [7].

When reasoning about infinite computations, it is often necessary to make use of assertions that may not hold every time a location is reached, but are guaranteed to hold in the limit. Given an SCS S with location ℓ, an assertion φ is said to be *tail invariant* at ℓ if, for any infinite computation π that never leaves S, φ fails to hold when π reaches ℓ only finitely many times. In other words, φ is an invariant of a suffix (or tail) of π. Tail invariants allow us to ignore the program states in the first pass of iterative structures such as **repeat** – **until** loops, in which the loop condition is evaluated at the end of each iteration.

Given an SCS S and an invariant map μ, μ can be strengthened to a map of tail invariants of S by forward propagation restricted to S. With each iteration, $\mathcal{F}$ is applied to μ and the resulting map is conjoined to the invariants of μ. It is sound, and usually sufficient, to terminate the forward propagation before a fixed point is reached.

2.5 Polyhedral cones and systems of linear constraints

Our invariant generator and our algorithm for generating ranking functions are based on polyhedral cones and systems of linear constraints.

A vector w is a *linear combination* of vectors $v_1, \ldots, v_n$ if $w = \lambda_1 v_1 + \cdots + \lambda_n v_n$ and a *conic combination* if $\lambda_1, \ldots, \lambda_n \geq 0$. The set of linear combinations of a set V is denoted $\mathcal{L}in(V)$, while $\mathcal{C}on(V)$ denotes its conic combinations. A *cone* is any set of vectors closed under conic combination. A pair $\langle L, R \rangle$ of sets of vectors is a *generator* of the cone C if $C = \mathcal{L}in(L) + \mathcal{C}on(R)$. The vectors in

L are known as the *lines* of the generator, while the members of R are the *rays*. A cone is *polyhedral* if it possesses a finite generator. In this paper, we consider only polyhedral cones.

The *polar* C^* of a cone C is the set of vectors forming non-acute angles with every member of C, i.e., $C^* = \{w \mid w \cdot v \leq 0 \text{ for all } v \in C\}$. A cone is polyhedral iff its polar is polyhedral. A *double description* is a pair of cones $\langle C, D \rangle$ satisfying $D = C^*$, and the *double description method* is an algorithm for computing polars of polyhedral cones based on this dual representation [10].

A *linear constraint* is an assertion of the form $\alpha_1 x_1 + \ldots + \alpha_d x_d + \beta \ \rho \ 0$, where ρ is $=$ or $\leq$. A conjunction of linear constraints is known as a *system*. The *theory* of a system S is the set of constraints satisfied by every solution of S. It was proven by Farkas that the theory of a system S of linear constraints is the cone it generates, where the equalities are interpreted as lines and the inequalities are treated as rays.

The polar of a system S of linear constraints viewed as a cone admits two interpretations: It can be seen as the generator of solutions of S or as a homogeneous system of constraints on the coefficients of the consequences of S. This second interpretation allows us to impose syntactic restrictions on the consequences of a system. For example, a variable x_i can be eliminated from a system by adding the constraint $\alpha_i = 0$ to its polar. That is, $\alpha_i = 0$ is added to the cone D in the representation $\langle C, D \rangle$ of S using the double description method pair $\langle C', D' \rangle$ with C' representing precisely those consequences of S in which the variable x_i does not appear.[1] Given two systems S_1 and S_2, their *convex hull* $S_1 \sqcup S_2$, i.e., the intersection of their consequences, can be computed by adding the constraints of the polar of S_2 to the polar of S_1.

3 An algorithm for generating ranking functions

3.1 Algorithm

The algorithm, of which a schematic outline is shown in Figure 1, consists of two phases. The first phase prepares the program for computing the ranking functions: invariants of P are generated and its CFG G is extracted. Then G is pruned by eliminating vertices from G that have an invariant of *false* and edges that can never be taken due to the unsatisfiability of the enabling condition of the corresponding transition relation given the generated invariants.

The ranking functions are computed in the second phase by the mutually recursive procedures **rank1** and **rank2**. Given a flow graph G, **rank1** decomposes it into its MSCS's and invokes **rank2** on each non-trivial MSCS. If all MSCS's are trivial, **rank1** succeeds immediately.

Given an MSCS S, **rank2** first partitions the variables of the program into those that are modified by some transition of S and those that are preserved by all transitions of S. Next it computes the tail invariants. It then computes the set N of all linear expressions over the modified variables that do not increase under

[1] In essence, the double description method is used to simulate Fourier's elimination.

```
Input: program $P$
Output: ranking functions for each SCS or fail
1. Generate invariants $\mu$ for $P$; Extract the CFG $G$ of $P$; Prune $G$ using $\mu$
2. call rank1($G$)

procedure rank1($G$)
1. decompose $G$ into a list $L$ of MSCS's
2. for each non-trivial $S \in L$ call rank2($S$)

procedure rank2($S$)
1.  partition the variables for $S$
2.  generate tail invariants of $S$
3.  compute set $N$ of non-increasing expressions for $S$
4.  for each $\tau \in S$ do
      5. compute the set $D$ of bounded decreasing expressions for $\tau$
      6. if $N \cap D \neq \emptyset$ do
          7. output any expression of $N \cap D$ as a ranking function for $S$
          8. remove $\tau$ from $S$
9.  if no transitions were removed from $S$ then fail
10. call rank1($S$)
```

Fig. 1. General algorithm for generating ranking functions

any transition of S. To do so, it first computes, for each transition $\tau = \langle \ell, \ell', \rho \rangle$ of S, the set of expressions e over modified variables such that

$$\mu(\ell) \wedge \rho \models e \geq e',$$

where e' denotes e with unprimed variables replaced by their primed versions. The computation is performed by representing the systems of linear inequalities $\mu(\ell)$ and ρ as polyhedral cones, taking their union, then adding equations to the polar of the combined system, to eliminate the unmodified variables and to ensure that the primed and unprimed versions of the same modified variable appear with opposite sign. Projecting this cone onto the primed variables then yields the generator of the non-increasing linear expressions for τ. (The precise details of this construction are presented in [4].) Taking the intersection of these cones over all $\tau \in S$ yields the set of expressions that are non-increasing over the entire MSCS.

Then, **rank2** computes for each transition $\tau = \langle \ell, \ell', \rho \rangle$ the set of expressions that both decrease under and are bounded from below by τ. That is, it computes the set of expressions e over the modified variables for which there exists a positive constant β and an expression Λ over the unmodified variables such that

$$\mu(\ell) \wedge \rho \models e \geq e' + \beta \quad \text{and} \quad \mu(\ell) \wedge \rho \models e' \geq \Lambda.$$

The restriction of the lower bound Λ to unmodified variables is necessary to ensure that the range of the purported ranking function is in fact well-founded. Again, **rank2** performs this computation by manipulating systems of linear inequalities represented as polyhedral cones.

```
in $n$ : integer where $n \geq 0$
in $A$ : array $[1 \ldots n]$ of integer
local $i, j$ : integer

$\ell_0$: $i := n$
$\ell_1$: while $i \geq 0$ do
    $\ell_2$: $j := 0$
    $\ell_3$: while $j \leq i - 1$ do
        $\ell_4$: if $A[j] > A[j+1]$ then
            $\ell_5$: $\langle A[j], A[j+1] \rangle := \langle A[j+1], A[j] \rangle$
        $\ell_6$: $j := j + 1$
    $\ell_7$: $i := i - 1$
```

Fig. 2. Program BUBBLESORT

Finally, for each transition τ possessing an expression δ which is bounded and decreasing under τ and non-increasing over the entire MSCS S, **rank2** outputs δ as a ranking function for S and removes τ from S. If no such transition exists, **rank2** fails. Otherwise, **rank2** invokes **rank1** to find ranking functions for the MSCS's created by the removal of these transitions.

Notice that it is sound to make use of tail invariants of an SCS S when generating its ranking functions. If δ can be shown to be both non-increasing over S and decreasing under τ assuming tail invariants, then for any infinite computation which remains in S, after finitely many steps in which δ changes arbitrarily, δ will attain a maximum value and then decrease whenever τ is taken.

3.2 Examples

Consider the program BUBBLESORT shown in Figure 2. **rank1** decomposes the CFG G of this program into the non-trivial MSCS $S_1 : \{\ell_1, \ldots, \ell_7\}$, and the trivial MSCS ℓ_0. Given S_1, **rank2** finds that i is non-increasing over the MSCS and is bounded and decreasing under τ_7, using the invariant $i = j \ \wedge \ j \geq 0$ generated for ℓ_7.

Eliminating τ_7 and decomposing the resulting graph yields a single new non-trivial MSCS $S_2 : \ \{\ell_3 \ldots \ell_6\}$. For S_2, **rank2** finds that $-j$ is non-increasing over the MSCS and bounded and decreasing under τ_6. Note that partitioning the variables enables i to appear in the lower bound on $-j$. Otherwise, the ranking function $i - j$, involving more than one variable, would be needed to show termination of S_2.

Figure 3.2 shows a program with a slightly more complicated control structure which was derived from McCarthy's 91 function [13]. Given input x, the function returns $x - 10$ if $x > 100$ and 91 otherwise. For this program, **rank2** generates $-y_1$ for the MSCS $\{\ell_3, \ell_4\}$ and $-y_1 + 11 * y_2$ for the MSCS $\{\ell_5 \ldots \ell_{11}\}$.

Program	no. of variables	no. of statements	no. of loops	inv.gen. (msec)	rank gen. (msec)
BUBBLESORT	3	8	2	88	101
PERFECT	4	10	2	2328	149
MCCARTHY91	4	11	2	235	154
DETERMINANT	5	13	3	570	351
MATRIX-CHAIN	5	19	4	836	504
LUP-DECOMPOSITION	4	28	5	691	439

Table 1. Run times for general algorithm on a 1GHz Xeon Pentium III processor with 2GB RAM, running Linux and Java 1.3.1

```
in $x$ : integer
local $y_1, y_2, z$ : integer

$\ell_0$: $(y_1, y_2) := (x, 1)$;
$\ell_1$: if $(y_1 > 100)$ then $\ell_2$: $z := y_1 - 10$
   else
   [ $\ell_3$: while $y_1 \leq 100$ do $\ell_4$: $(y_1, y_2) := (y_1 + 11, y_2 + 1)$;
     $\ell_5$: while $y_2 > 1$ do
        [ $\ell_6$: $(y_1, y_2) := (y_1 - 10, y_2 - 1)$;
          $\ell_7$: if $y_1 > 100 \wedge y_2 = 1$ then $\ell_8$: $z := y_1 - 10$
          else
             [ $\ell_9$: if $y_1 > 100$ then $\ell_{10}$: $(y_1, y_2) := (y_1 - 10, y_2 - 1)$;
               $\ell_{11}$: $(y_1, y_2) := (y_1 + 11, y_2 + 1)$ ] ] ]
```

Fig. 3. Program derived from McCarthy's 91 function

3.3 Some experimental results

We have implemented our algorithm in Java using the invariant generator and polyhedral cone library of STeP [1] and have applied it to several programs taken from [13] and [5], obtaining the results presented in Table 1. Note that most of the execution time is spent in the invariant generator.

Encouraged by these results, we then applied the algorithm to a larger program – an implementation of mergesort taken from [16] and shown in Fig.6. For this example, the initial results were disappointing. The invariant generator failed to converge in a reasonable amount of time. We then re-ran the algorithm, restricting the invariant generator to consider only the variables i, m, n, p, q and r, which we knew *a priori* to be the only variables relevant to termination. With this restriction, the invariant generator converged in 5 seconds, generating invariants sufficiently strong to demonstrate termination in 2.5 sec. The generated ranking functions are $-p, m, r, q$, and $-i$ for the loops at $\ell_2, \ell_{14}, \ell_{33}, \ell_{38}$, and ℓ_{49}, respectively, and the ranking functions q and r for the loop at ℓ_{23}.

Unsatisfied with an algorithm that requires guidance in the form of a list of relevant variables to ensure convergence for a somewhat large, but not unduly

```
**Input:** program P
**Output:** invariant map $\mu$
1. construct CFG $G$ of P; initialize $\mu$ to $\emptyset$
2. call inv1($G$)

procedure inv1($G$)
1. decompose $G$ into an ordered list $L$ of MSCS's
2. for each $S \in L$ in order do
    3. if $S$ is trivial then propagate assertions to $S$'s vertex
    4. else call inv2($S$)

procedure inv2($S$)
1. let $v$ be the header of $S$
2. if $\mu(v)$ is undefined then
    3. forward propagate assertions to $v$
    4. extrapolate $\mu(v)$ over $S$
5. remove from $S$ all edges to $v$
6. call inv1($S$)
```

Fig. 4. Algorithm for generating invariants

complex program, we devised a more heuristic algorithm targeted at structured programs with simple ranking functions, which we present in the next section.

4 An alternate algorithm for generating ranking functions

Our heuristic algorithm is based on two observations concerning the programs we have considered. First, they are all written in structured programming languages and, therefore, have reducible CFG's. Recall that a vertex v *dominates* a vertex w if every proper path to w passes through v; a flow graph G is *reducible* if every SCS S contains a vertex v that dominates it, called its *header*. Reducible flow graphs are well-structured: Their loops are properly nested.

Based on this observation, we devised an invariant generator that takes advantage of reducible CFG's. The algorithm, shown in Figure 4, propagates assertions through the CFG. However, upon encountering a non-trivial SCS, rather than iterating with widening until convergence, the algorithm attempts to extrapolate an over-approximation of the fixed point, which it then propagates into and past the SCS.

The extrapolation algorithm used is simple. Given an assertion, represented as a system of linear inequalities, it first computes those consequences of the system involving only variables not modified in the SCS, that is, it eliminates the modified variables. Then an attempt is made to refine the (now invariant) assertion by preserving any bounds on the modified variables implied by the original system. The approach taken is to determine for each modified variable whether the original system implies a bound on that variable which is linear in the unmodified variables and which is preserved by each transition. If so, the approximation is strengthened by including this bound.

```
**Input:** program $P$
**Output:** ranking functions for each SCS
1. Generate invariants $\mu$ for $P$; Extract CFG $G$ of $P$; Prune $G$ using $\mu$
2. call **rank1**$(G)$
3. if any SCS does not have a ranking function, fail

**procedure rank1**$(G)$
1. decompose $G$ into a list $L$ of MSCS's
2. for each non-trivial $S \in L$ call **rank2**$(S)$

**procedure rank2**$(S)$
1. partition the variables for $S$
2. generate tail invariants of $S$
3. for each variable $v$ modified in $S$ do
   4. if $v$ is non-increasing/non-decreasing over $S$ then
      5. for each $\tau \in S$ do
         6. if $v$ is bounded and decreasing/increasing under $\tau$ then
            7. record $\tau$ and $v/-v$
8. remove from $S$ all edges to its header
9. call **rank1**$(S)$
```

Fig. 5. Algorithm for generating ranking functions

The second observation concerning the programs we consider is that, provided care is taken to distinguish between those variables that are modified in an SCS and those that are preserved, all SCS's of these programs possess single-variable ranking functions. Thus, a heuristic algorithm that restricts itself to ranking functions of this form is likely to prove termination of most loops while avoiding the overhead of manipulating polyhedral cones. An algorithm for generating ranking functions incorporating this idea is presented in Figure 5.

The algorithm visits the loops of the program and attempts to find single-variable ranking functions that justify the elimination of transitions of the loop that are not transitions of any inner loop. It records, for any ranking function discovered, both the function and the set of transitions whose removal is justified by the function. It then breaks all cycles of the loop from the loop header and invokes itself recursively. Note that, unlike the algorithm of Section 3, this algorithm continues to search for ranking functions of inner loops even if it fails to prove termination of the outer loops. The structure of reducible CFG's makes this possible. If, after visiting all loops, removing the decreasing transitions from the CFG results in an acyclic graph, the algorithm reports success.

We have implemented this second algorithm and applied it to the programs considered in Section 3, obtaining the results shown in Table 2. As expected, this algorithm fails on MCCARTHY91, since no single-variable ranking function exists for the second MSCS. However, invariant generation is much faster than the iterative forward propagation with widening used in the previous section.

```
in $n$ : integer where $n > 0$
local $i, j, k, l, t$ : integer
local $h, m, p, q, r$ : integer
local $up$ : boolean
local $a$ : array $[1..2 * n]$ of integer

$\ell_0$: $up := \mathrm{T}$;
$\ell_1$: $p := 1$;

$\ell_2$: repeat
    [$\ell_3$: $h := 1$;
     $\ell_4$: $m := n$;
     $\ell_5$: if $up$ then
         [$\ell_6$: $i := 1$;  $\ell_7$: $j := n$;  $\ell_8$: $k := n + 1$;  $\ell_9$: $l := 2 * n$]
       else
         [$\ell_{10}$: $k := 1$;  $\ell_{11}$: $l := n$;  $\ell_{12}$: $i := n + 1$;  $\ell_{13}$: $j := 2 * n$] ;

     $\ell_{14}$: repeat
         [$\ell_{15}$: if $m \geq p$ then $\ell_{16}$: $q := p$ else  $\ell_{17}$: $q := m$;
          $\ell_{18}$: $m := m - q$;
          $\ell_{19}$: if $m \geq p$ then $\ell_{20}$: $r := p$ else  $\ell_{21}$: $r := m$;
          $\ell_{22}$: $m := m - r$;

          $\ell_{23}$: while $(q > 0 \ \wedge \ r > 0)$ do
              [$\ell_{24}$: if $a[i] < a[j]$ then
                   [$\ell_{25}$: $a[k] := a[i]$;
                    $\ell_{26}$: $k := k + h$;   $\ell_{27}$: $i := i + 1$;   $\ell_{28}$: $q := q - 1$]
                 else
                   [$\ell_{29}$: $a[k] := a[j]$;
                    $\ell_{30}$: $k := k + h$;   $\ell_{31}$: $j := j - 1$;   $\ell_{32}$: $r := r - 1$]] ;

          $\ell_{33}$: while $(r > 0)$ do
              [$\ell_{34}$: $a[k] := a[j]$;
               $\ell_{35}$: $k := k + h$;   $\ell_{36}$: $j := j - 1$;   $\ell_{37}$: $r := r - 1$] ;

          $\ell_{38}$: while $(q > 0)$ do
              [$\ell_{39}$: $a[k] := a[i]$;
               $\ell_{40}$: $k := k + h$;   $\ell_{41}$: $i := i + 1$;   $\ell_{42}$: $q := q - 1$] ;

          $\ell_{43}$: $h := -h$;   $\ell_{44}$: $t := k$;   $\ell_{45}$: $k := l$;   $\ell_{46}$: $l := t$]
       until $m \leq 0$;

     $\ell_{47}$: $up :=!up$;   $\ell_{48}$: $p := 2 * p$;]
  until $p \geq n$;

$\ell_{49}$: if $!up$ then
    [$\ell_{50}$: $i := 1$;
     $\ell_{51}$: while $i \leq n$ do
         [$\ell_{52}$: $a[i] := a[i + n]$;   $\ell_{53}$: $i := i + 1$]]
```

Fig. 6. Program MERGESORT

Program	no. of variables	no. of statements	no. of loops	inv.gen. (msec)	rank gen. (msec)
BUBBLESORT	3	8	2	67	72
PERFECT	4	10	2	76	75
MCCARTHY91	4	11	2	-	-
DETERMINANT	5	13	3	169	186
MATRIX-CHAIN	5	19	4	216	233
LUP-DECOMPOSITION	4	28	5	186	208
MERGESORT	11	54	6	2665	3781

Table 2. Run times for heuristic algorithm on a 1GHz Xeon Pentium III processor with 2GB RAM, running Linux and Java 1.3.1

5 Conclusions

We present two algorithms to prove termination of programs. The first, more powerful algorithm is capable of finding subtle ranking functions which are linear combinations of many program variables, but is limited to short programs with few variables. The second, more heuristic algorithm, finds single-variable ranking functions in structured programs of larger size.

The approach taken by our first method bears resemblance to methods for automatically synthesizing *polynomial interpretations* [9] for establishing termination of term rewriting systems. For example, [11] extracts a system of non-linear constraints on the coefficients of low-degree polynomials which guarantee correctness, then uses a combination of heuristic instantiation and a variant of *cylindrical algebraic decomposition* [3] to solve these constraints. Our algorithm can be seen as employing the double description method to solve a system of linear constraints on the coefficients of a linear expression which guarantee monotonicity of the defined function and the well-foundedness of its range. Our second method is similar to the heuristic approach proposed in [8], which identifies candidate single-variable ranking functions based on bounds appearing in the program, then verifies them using decision procedures. Our heuristic algorithm combines these two steps, using the double description method as a decision procedure. In addition, by making use of an invariant generator, our algorithm is able to discover ranking functions whose bounds do not appear explicitly in the program, but are implicit.

We see the utility of our algorithms in their potential to be incorporated into light-weight static analysis tools to identify potentially nonterminating loops. Preliminary analysis of a Web server, implemented by some 30,000 lines of Java code, indicated that one third of the loops could be proved terminating directly with our methods. Combined with static analysis to identify loop invariants our methods would be able to handle about half of the loops. If in addition we augmented our methods with some simple mechanisms to keep track of the size of collections, more than 80% of the loops could be handled. Thus, incorporated in an analysis tool, our methods could potentially relieve the programmer from checking 80% of the loops manually, if termination was a critical requirement.

References

1. Nikolaj S. Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design*, 16(3):227–270, June 2000.
2. J. Brauburger and J. Giesl. Approximating the domains of functional and imperative programs. *Science of Computer Programming*, 35:113–136, 1999.
3. G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Brakhage, editor, *Proc. Second GI Conf. Autamata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183, 1975.
4. Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 67–81. Springer Verlag, April 2001.
5. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, 1990.
6. Patrick Cousot and Rhadia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 238–252. ACM Press, 1977.
7. Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among the variables of a program. In 5^{th} *ACM Symp. Princ. of Prog. Lang.*, pages 84–97, January 1978.
8. 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.
9. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
10. K. Fukuda and A. Prodon. Double description method revisited. In *Combinatorics and Computer Science*, volume 1120 of *Lecture Notes in Computer Science*, pages 91–111. Springer-Verlag, 1996.
11. J. Giesl. Generating polynomial orderings for termination proofs. In J. Hsiang, editor, *Proc. 6th Intl. Conf. Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 426–431. Springer-Verlag, 1995.
12. J. Giesl, C. Walther, and J. Brauburger. Termination analysis for functional programs. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction – A Basis for Applications, Volume III: Applications*, chapter 6, pages 135–164. Kluwer Academic, 1998.
13. Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.
14. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.
15. D. de Schreye and S. Decorte. Termination of logic programs: The never ending story. *Journal of Logic Programming*, 19, 20:199–260, 1994.
16. Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.