Bounding the Computational Complexity of Flowchart Programs with Multi-dimensional Rankings

Christophe Alias, Alain Darte, Paul Feautrier, Laure Gonnord

▶ **To cite this version:**

Bounding the Computational Complexity of Flowchart Programs with Multi-dimensional Rankings

Christophe Alias — Alain Darte — Paul Feautrier — Laure Gonnord

Bounding the Computational Complexity of Flowchart Programs with Multi-dimensional Rankings

Christophe Alias[*], Alain Darte [†], Paul Feautrier [‡], Laure Gonnord [§]

Abstract: Proving the termination of a flowchart program can be done by exhibiting a ranking function, i.e., a function from the program states to a well-founded set, which strictly decreases at each program step. A standard method to automatically generate such a function is to compute invariants for each program point and to search for a ranking in a restricted class of functions that can be handled with linear programming techniques. Previous algorithms based on affine rankings either are applicable only to simple loops (i.e., single-node flowcharts) and rely on enumeration, or are not complete in the sense that they are not guaranteed to find a ranking in the class of functions they consider, if one exists. Our first contribution is to propose an efficient algorithm to compute ranking functions: It can handle flowcharts of arbitrary structure, the class of candidate rankings it explores is larger, and our method, although greedy, is provably complete. Our second contribution is to show how to use the ranking functions we generate to get upper bounds for the computational complexity (number of transitions) of the source program, again for flowcharts of arbitrary structure. This estimate is a polynomial, which means that we can handle programs with more than linear complexity. We applied the method on a collection of test cases from the literature. We also point out important extensions, mainly to do with the scalability of the algorithm and, in particular, the integration of techniques based on cutpoints.

Key-words: static analysis, termination proof, multidimensional affine ranking functions, worst-case time complexity estimation

[*] INRIA Researcher
[†] CNRS Researcher
[‡] Professor at ENS Lyon
[§] Research Assistant at Lille University

Estimation automatique de la complexité au pire avec des fonctions de terminaison multidimensionnelles

Résumé : On peut prouver la terminaison d'un programme en exhibant une fonction de ranking, i.e., une application des états du programme vers un ensemble bien fondé, qui décroit strictement à chaque étape du calcul. Une méthode standard pour générer une telle fonction est de calculer les invariants pour chaque point du programme et de chercher dans une classe de fonctions restreinte en utilisant des techniques de programmation linéaire. Les approches précédentes sont soit applicables uniquement à des boucles simples et utilisent de l'enumération, ou bien ne sont pas complètes dans le sens où il n'est pas garanti qu'elles trouvent une fonction de ranking quand elle existe dans la classe considérée. Notre première contribution est un algorithme efficace pour calculer une fonction de ranking. Les graphes de flot de contrôle arbitraires sont traités, la classe de fonctions de ranking gérée est plus grande et notre méthode, bien que gloutonne, est complète. Notre seconde contribution est de montrer comment calculer une borne supérieure sur la complexité en temps de l'algorithme (nombre de transitions franchies) à partir d'une fonction de ranking. Le résultat est une fonction polynomiale par morceaux, ce qui signifie que nous pouvons traiter des programmes avec une complexité plus que linéaire. La méthode a été appliquée avec succès sur une collection d'exemples de la littérature. Nous montrons aussi plusieurs extensions importantes de l'algorithme permettant de traiter des exemples de grande taille.

Mots-clés : analyse statique, preuve de terminaison, génération de fonctions de terminaison multidimensionnelles affines, estimation de la complexité au pire

1 Introduction

The problem of proving program correctness has been with us since the early days of Computer Science. In a seminal paper [20], R. W. Floyd proposed what has become one of the standard approaches: affix assertions to each program point and prove that they are consequences of the assertions of its predecessors in the program control graph. The assertions at the entry point of the program are its *preconditions*, the assertions at loop entry points are *invariants*, while the assertions at its exit point must entail correctness, according to some set of requirements. Constructing the required set of assertions is a tedious and error-prone task. The automatic construction of invariants has been proved to be untractable in the general case [5]. However, partial or conservative solutions can be obtained by abstract interpretation methods [14].

At the same time, it was soon realized that this method proves only partial correctness, i.e., that the program gives the correct result if and when it terminates. To prove termination, one needs a variant or ranking function (a W-function in Floyd's terminology), i.e., a function from the states of the program to some well-founded set, which strictly decreases at each program step. Of course, designing an algorithm for building ranking functions in all cases is not possible since it would give a solution to the undecidable halting problem. However, this does not preclude the existence of partial solutions, which, e.g., handle only programs (or approximated models) of a restricted shape, or look for rankings in a restricted class of functions. For example, the techniques of Podelski and Rybalchenko [29] and of Bradley, Manna, and Sipma [8] are designed to handle "single-path linear loops" (basically a single basic block with multiple self-loops) or programs approximated into such simple loops thanks to precomputations similar to transitive closure on control-flow paths. The class of ranking functions they consider is the class of affine functions, either one-dimensional [29] or multi-dimensional [8]. In addition, the technique of [8] tries to compute the invariants and the ranking functions simultaneously. Also, as it is based on an exhaustive search to map each transition on a given ranking dimension, it is complete in the sense that it is guaranteed to find a ranking in the class of functions considered, if one exists. Unlike these two methods, the technique of Colón and Sipma [13] can handle flowchart programs of arbitrary structure. The class of functions considered is larger (one affine expression for each strongly connected component at a given level of the algorithm) but the technique is not proved to be complete. Our first contribution is to generalize these previous work for generating ranking functions. We design an algorithm with the following features:

- It can handle flowcharts of arbitrary structure.

- The class of ranking functions we consider is much larger: in the global ranking function we generate, each program point can have its own multi-dimensional affine expression.

- Our algorithm is based on a greedy mechanism. Nevertheless, our technique is provably complete, even for our larger class of ranking functions.

There are many variations on the above theme. For instance, as in [28], one may select a set of *cutpoints*, with the property that if the cutpoints are removed, the flowchart becomes acyclic. It is then enough to exhibit a function that is non increasing everywhere, and that decreases and is well-founded at each cutpoint. One may even proceed each cycle of the flowchart at a time. Another feature of our algorithm is that it can easily be extended to exploit cutpoints.

Our second contribution is to show that the global ranking functions we generate can be used to give upper bounds on the worst-case computational complexity (WCCC) of the program execution, i.e., the number of transitions that can be made in an execution trace. Obviously, if a program does not terminate, its WCCC is infinite. If the program terminates and a one-dimensional ranking function exists, its value at program start is an upper bound on the number of steps before termination since it decreases at least by one at each program step. The situation is more complicated in the case of multi-dimensional ranking functions but we show how it can be computed thanks to counting techniques in polyhedra. Furthermore, our ranking algorithm has an additional important feature:

- It generates a multi-dimensional affine ranking function whose dimension is minimal. This minimality is important to get an accurate upper bound on the WCCC of the flowchart program.

To the best of our knowledge, our technique is the first one that uses ranking functions to compute upper bounds on the number of iterations of arbitrary loops (a particular case of the WCCC).

The rest of the paper is organized as follows. Section 2 gives some basic notations and concepts: the abstraction of programs we use (*integer interpreted automata*), the class of ranking functions we consider, and the WCCC. Section 3 presents our method for constructing multi-dimensional affine ranking functions and to infer the computational complexity of the source program. Section 4 details the different features of our method, in particular its completeness. In Section 5, we report on our implementation through a collection of benchmarks from the literature. We next describe other approaches to the termination problem and conclude, pointing to some unsolved problems and outlining future work.

2 Notations and definitions

We write matrices with capital letters (as A) and column vectors with a top arrow (as $\vec{x}$). If $\vec{x}$ has dimension d, its components are denoted $\vec{x}[i]$, with $0 \leq i < d$. Thus, its i-th component is $\vec{x}[i-1]$. Sets are represented with calligraphic letters such as $\mathcal{W}$, $\mathcal{K}$, etc.

2.1 Integer interpreted automata

In the tradition of most previous work on program termination and static program analysis, we do not start from the program itself but from an abstraction: the associated *integer interpreted automaton*. This is similar to the flowcharts used a long time ago to express programs (see, e.g., Manna's book [28]) until the advent of structured programming. However, when one looks at real-life programs, many deviations from the strict structured model can occur, including premature loop termination, exceptions, and even the occasional goto. Starting from a flowchart allows us not to depend of the details of the syntax and semantics of the source language, which can be dealt with by an appropriate preprocessor.

In our work, a program is represented by an *extended relational (integer) interpreted automaton* $(\mathcal{K}, n, k_{init}, \mathcal{T})$ defined by:

- a finite set $\mathcal{K}$ of *control points*;
- n integer variables represented by a vector $\vec{x}$ of size n;
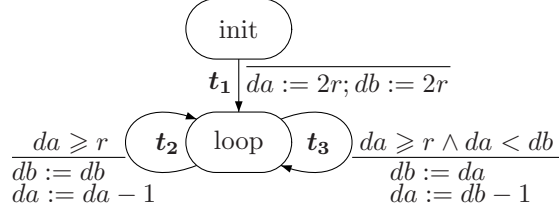- an initial control point $k_{init} \in \mathcal{K}$;

Figure 1: Affine interpreted automaton

- a finite set $\mathcal{T}$ of 4-tuples (k, g, a, k'), called *transitions*, where $k \in \mathcal{K}$ (resp. $k' \in \mathcal{K}$) is the source (resp. target) control point, $g : \mathbb{Z}^n \mapsto \mathbb{B} = \{\text{true}, \text{false}\}$ is a guard (function from variable valuations to Booleans), and $a : \mathbb{Z}^n \mapsto (\mathbb{Z} \cup \{?\})^n$ is an action that assigns, to each variable valuation $\vec{x}$, a vector with possibly some unspecified components (denoted by the symbol "?").

Semantics The set of states is $\mathcal{K} \times \mathbb{Z}^n$. A *trace* from $(k_0, \vec{x}_0)$ to $(k, \vec{x})$ is a sequence $(k_0, \vec{x}_0), (k_1, \vec{x}_1), \ldots, (k_p, \vec{x}_p)$ such that $k_p = k$, $\vec{x}_p = \vec{x}$ and for each i, $0 \le i < p$, there exists in $\mathcal{T}$ a transition (k_i, g_i, a_i, k_{i+1}) such that $g_i(\vec{x}_i) = \text{true}$ and $\vec{x}_{i+1} = a_i(\vec{x}_i)$. If $a_i(\vec{x}_i)$ has a "?" component, the corresponding component of $\vec{x}_{i+1}$ may take an arbitrary value in $\mathbb{Z}$. Given an initial valuation $\vec{v}$, a state $(k, \vec{x})$ is *reachable from* $\vec{v}$ iff (if and only if) there is a trace from $(k_{init}, \vec{v})$ to $(k, \vec{x})$. A state $(k, \vec{x})$ is *reachable* if there exists $\vec{v} \in \mathbb{Z}^n$ such that $(k, \vec{x})$ is reachable from $\vec{v}$. The set of reachable states is denoted by $\mathcal{R}$.

In this paper, we consider *affine interpreted automata*, i.e., integer interpreted automata with *extended affine guarded transitions*.

Definition 1 (Extended affine guarded transition) *A transition* $t = (k, g, a, k')$ *is an* extended affine guarded transition *when:*
- *$g(\vec{x})$ is true iff $G\vec{x} + \vec{g} \ge \vec{0}$ (component-wise) where $\vec{g}$ is an integer vector and G an integer matrix with n columns and as many rows as the size of $\vec{g}$;*
- *there exists an integer matrix A and an integer vector $\vec{a}$ such that the i-th component of $\vec{x}' = a(\vec{x})$ is the i-th component of $A\vec{x} + \vec{a}$ or is equal to "?".*

An example of such an automaton is given in Figure 1. The control points are labelled for convenience, and transitions are represented with arrows indexed by $\frac{g}{a}$ (g is omitted when $g = \text{true}$).

Example 1 The C code below is an abstraction of a real C code computing the greatest common divisor (gcd) of two polynomials.

```
// expr is an expression, A is an array,
// r is a constant positive integer parameter.
  da = 2r; db = 2r;
  while (da >= r) {
    cond = ( da >= db || A[expr] == 0 );
    if (!cond) { tmp = db; db = da; da = tmp - 1; }
    else da = da - 1;
  }
```

This simplified code is itself abstracted by the automaton of Figure 1, where `init` is the initial control point and `loop` corresponds to the while loop. The fact that the condition `A[expr]==0` cannot be statically evaluated introduces some non-determinism

in the automaton. To enter the `then` part of the test, the condition `da < db` must be true (transition t_3) while the `else` part can always be traversed (transition t_2) as long as the termination test of the while loop is false (additional condition `da >= r` for both transitions). □
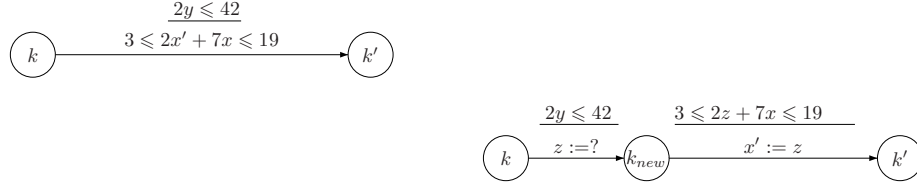
Another popular model of integer automaton represents the effect of transitions, changing the variables $\vec{x}$ into $\vec{x}'$, as an affine relation $A'\vec{x}' + A\vec{x} + a \geq 0$ of $\vec{x}$ and $\vec{x}'$, i.e., describe the set of possible pairs $(\vec{x}, \vec{x}')$ as a polyhedron Q_t. Lemma 1 below shows that these two representations are equivalent. We use both notations in our work: our theory is simpler to describe with affine relations, while the external tool we use for computing invariants relies on extended affine functions.

Lemma 1 *A general transition with affine relations can be encoded as the sequence of two extended affine transitions.*

Proof Let $t = (k, Q_t, k')$ be a general affine transition. Let $\vec{z}$ be a vector of n fresh variables and k_{new} a new control point. Then, the transition t is equivalent to the combination of t_1 and t_2, where:

- $t_1 = (k, true, a_1, k_{new})$ with $a_1(\vec{x}, \vec{z}) = (\vec{x}, (?)^n)$;
- $t_2 = (k_{new}, g_2, a_2, k')$ where $g_2(\vec{x}, \vec{z}) = true$ iff $(\vec{x}, \vec{z}) \in Q_t$ (affine guard) and $a_2(\vec{x}, \vec{z}) = (\vec{z}, \vec{z})$ (projection).

In other words, the affine relation is encoded in the guard of the second transition. As an illustration, the transition on the left below is equivalent to the pair of transitions on the right. ∎

Invariants   The guard g in a transition $t = (k, g, a, k')$ gives a necessary condition on variables $\vec{x}$ to traverse the transition t and to apply its corresponding action a. To get the exact valuations $\vec{x}$ of variables for which the action a can be performed, one would need to take into account the initial valuations and the successive conditions that led to the control point k. We denote by $\mathcal{R}_k$ the set of possible valuations $\vec{x}$ of variables when the control is in k:

$$\mathcal{R}_k = \{\vec{x} \in \mathbb{Z}^n \mid (k, \vec{x}) \in \mathcal{R}\}.$$

Then, there exists a trace containing the transition (k, g, a, k') iff $\vec{x} \in \mathcal{R}_k$ and $g(\vec{x})$ is true. Note that $\mathcal{R}_k$ does not depend on any initial valuation. More precisely, it is the union, for all initial valuations $\vec{v}$, of the set of vectors $\vec{x}$ such that $(k, \vec{x})$ is reachable from $\vec{v}$.

In practice, it is difficult to determine the set $\mathcal{R}_k$ exactly but it is possible to give over-approximations, thanks to the notion of *invariants*. An invariant on a control point k is a formula $\phi_k(\vec{x})$ that is true for all reachable states $(k, \vec{x})$. An invariant is *affine* if it is the conjunction of a finite number of affine conditions on program variables. The set $\mathcal{R}_k$ is then over-approximated by the integer points within a polyhedron $\mathcal{P}_k$. Since the seminal paper [14], the problem of constructing invariants has been widely studied. Here, instead of coupling the invariant construction and the termination proof as in [8], we rely on Aspic, a public domain implementation of *abstract acceleration* [21]. Compared to the standard widening approach, this method computes a more precise

reachability set for "accelerable" loops, which locally avoids the use of widening and globally increases precision. Aspic takes as input an affine interpreted automaton and outputs an affine invariant for each control point.

Example 1 (Cont'd)   With the initial condition $r \geq 1$ and loop exit to end with guard $r > da$, Aspic produces the following invariants:

- $P_{init} = \{1 \leq r\}$;
- $P_{loop} = \{da \leq 2r, db \leq 2r, 1 \leq r, r \leq db, r - 1 \leq da\}$;
- $P_{end} = \{da < r, db \leq 2r, 1 \leq r, r \leq db, r - 1 \leq da\}$.   □

2.2   Termination and ranking functions

Invariants can only prove partial correctness of a program. The standard technique for proving termination is to consider ranking functions and well-founded sets. A well-founded set $\mathcal{W}$ is a set with a (total or partial) order $\preceq$ (we write $a \prec b$ if $a \preceq b$ and $a \neq b$) such that there is no infinite descending chain, i.e., no infinite sequence $(x_i)_{i \in \mathbb{N}}$ with $x_i \in \mathcal{W}$ and $x_{i+1} \prec x_i$ for all $i \in \mathbb{N}$.

Definition 2 (Ranking function) *A ranking function is a function $\rho : \mathcal{K} \times \mathbb{Z}^n \to \mathcal{W}$, from the automaton states to a well-founded set $(\mathcal{W}, \preceq)$, whose values decrease at each transition $t = (k, g, a, k')$:*

$$\vec{x} \in \mathcal{R}_k \wedge g(\vec{x}) = \text{true} \wedge \vec{x}' \in a(\vec{x}) \Rightarrow \rho(k', \vec{x}') \prec \rho(k, \vec{x}) \tag{1}$$

It is said affine *if it is affine in the second parameter (the variables).*

Definition 3 (1D & kD ranking) *A ranking function ρ is one-dimensional if its co-domain is $(\mathbb{N}, \leq)$. It is k-dimensional (or multi-dimensional of dimension k) if its co-domain is $(\mathbb{N}^k, \leq_k)$, where the order $\leq_k$ is the standard lexicographic order on integer vectors.*

Obviously, the existence of a ranking function implies program termination for any valuation $\vec{v}$ at the initial control point k_{init}. A well-known property is that an integer interpreted automaton terminates for any initial valuation if and only if it has a ranking function. Furthermore, if it terminates and has bounded non-determinism, there is a one-dimensional ranking function. For the proofs, see our previous research report on the subject [1].

In general, algorithms for termination look for special sets $\mathcal{W}$ and ranking functions of a certain class. For example, in [26], Lee studies ranking functions for a particular approximation of recursive programs, those with the so-called SCT property (size-change termination). According to this study, for SCT programs, it is sufficient to consider "minimums and maximums over lexicographic tuples", i.e., simple functions into $(N^d, \leq_d)$. In the context of affine transitions, all previous work restrict to special class of affine ranking functions. The techniques developed in [29, 8] consider only simple loops, i.e., $\mathcal{K}$ is a singleton. Ranking functions are then restricted to affine functions from $\mathbb{Z}^n$ to $\mathcal{W}$, where $\mathcal{W} = (N, \leq)$ for [29] and $\mathcal{W} = (N^d, \leq_d)$ for [8] where d is, typically, equal to the number of transitions. The recursive algorithm of [13] handles general flowchart programs (thus $\mathcal{K}$ is arbitrary) and builds a ranking function from $(\mathcal{W}, \mathbb{Z}^n)$ to $(\mathbb{N}^d, \leq_d)$ where d is given by the depth of the recursion. For each recursive call, the algorithm considers each strongly connected component (SCC) formed by the transitions "not yet satisfied" and looks for an affine expression of the variables. In

other words, in each SCC, the ranking function does not depend on the control points k. Because of this restriction, the algorithm is not complete for the class of general multi-dimensional affine rankings. In contrast, we are able to capture all such rankings and our algorithm is complete. The reader is referred to [17] for relations between the present problem and scheduling methods.

3   Computing affine ranking functions

This section presents an algorithm to construct multi-dimensional affine ranking functions, i.e., ranking functions $\rho : \mathcal{K} \times \mathbb{Z} \to \mathbb{N}^d$, affine for the second parameter. The integer d is the dimension of the ranking. For the sake of clarity, we first recall in Section 3.1 the well-known method to get a *one-dimensional* affine ranking function. The method is then generalized in Section 3.2 to compute an affine ranking function of dimension $d > 1$. In our implementation, both cases are handled by the same algorithm.

Note: as we use linear programming (but not integer linear programming), we may end up with functions with rational components. Then, we can always multiply such a function by a suitable integer to get a ranking function with integer values. Thus, in the rest of the paper, we will not insist on this subtlety any longer.

3.1   One-dimensional affine ranking functions

As explained in Section 2.1, in practice, the exact sets $\mathcal{R}_k$ are not necessarily available. They are over-approximated by invariants $\mathcal{P}_k$, with $\mathcal{R}_k \subseteq \mathcal{P}_k$, which are, in our case, described by polyhedra. The conditions that a ranking function must satisfy are then related to these invariants and not to the exact sets of reachable states.

A one-dimensional ranking function ρ has co-domain $\mathbb{N}$, i.e., it assigns a nonnegative integer to each relevant state:

$$\vec{x} \in \mathcal{P}_k \Rightarrow \rho(k, \vec{x}) \geq 0 \tag{2}$$

Consider Inequality (1), which specifies that the ranking decreases on transitions. Let Q_t be the polyhedron described by the constraints of a transition $t = (k, g, a, k')$, i.e., $\vec{x} \in \mathcal{P}_k$, $g(\vec{x})$ is true, and $\vec{x}' = a(\vec{x})$. For an extended affine interpreted automaton, Q_t is built from matrices A and G, and vectors $\vec{a}$ and $\vec{g}$ (see Definition 1). For an automaton whose actions are general affine *relations*, Q_t is directly given by the actions definition. Inequality (1) then becomes:

$$\vec{y} = (\vec{x}, \vec{x}') \in Q_t \Rightarrow \rho(k, \vec{x}) - \rho(k', \vec{x}') \geq 1 \tag{3}$$

It remains to linearize Inequalities (2) and (3) and to get a ranking function by means of a linear solver. The standard method (used in [18, 29, 8]) is to rely on the affine form of Farkas lemma [31]:

Lemma 2 (Farkas lemma, affine form) *An affine form $\phi : \mathbb{R}^n \to \mathbb{R}$ with $\phi(\vec{x}) = \vec{c}.\vec{x} + c_0$ is nonnegative everywhere in a non-empty polyhedron $\{\vec{x} \mid A\vec{x} + \vec{a} \geq \vec{0}\}$ iff:*

$$\exists \vec{\lambda} \in (\mathbb{R}^+)^n, \lambda_0 \in \mathbb{R}^+ \text{ such that } \phi(\vec{x}) \equiv \vec{\lambda}.(A\vec{x} + \vec{a}) + \lambda_0$$

The notation $\equiv$ is a formal equality, which means that $\vec{x}$ can be eliminated and coefficients identified. In other words:

$$\exists \vec{\lambda} \in (\mathbb{R}^+)^n, \lambda_0 \in \mathbb{R}^+ \text{ such that } \vec{c} = \vec{\lambda}.A \text{ and } c_0 = \vec{\lambda}.\vec{a} + \lambda_0$$

For all $\vec{x}$, $\phi(\vec{x})$ is thus expressed as the dot product of $\vec{x}$ and a nonnegative combination of the normals of the facets of $\mathcal{P}$, plus a nonnegative constant. We can now apply the affine form of Farkas lemma to Inequalities (2) and (3). Write $P_k\vec{x} + \vec{p}_k \geq \vec{0}$ the constraints that define $\mathcal{P}_k$. For Inequality (2), we get:

$$\exists \vec{\lambda}_k \in (\mathbb{R}^+)^n, \lambda_k^0 \in \mathbb{R}^+ \text{ such that } \rho(k, \vec{x}) \equiv \vec{\lambda}_k.(P_k\vec{x} + \vec{p}_k) + \lambda_k^0 \qquad (4)$$

Similarly, let us write $Q_t = \{\vec{y} = (\vec{x}, \vec{x}') \mid Q_t\vec{y} + \vec{q}_t \geq \vec{0}\}$. We call $\Delta_t(\rho, \vec{x}, \vec{x}') = \rho(k, \vec{x}) - \rho(k', \vec{x}')$ the *delay* of transition t. Inequality (3) states that $\Delta_t(\rho, \vec{x}, \vec{x}') \geq 1$, which means:

$$\exists \vec{\mu}_t \in (\mathbb{R}^+)^n, \mu_t^0 \in \mathbb{R}^+ \text{ s.t. } \Delta_t(\rho, \vec{x}, \vec{x}') - 1 \equiv \vec{\mu}_t.(Q_t\vec{y} + \vec{q}_t) + \mu_t^0 \qquad (5)$$

A substitution of (4) in (5) and an identification on each dimension of $\vec{y}$ leads to a linear system $\mathcal{S}_t$ with nonnegative unknowns $\vec{\lambda}_k$, λ_k^0 (from $\rho(k, \vec{x})$), $\vec{\lambda}_{k'}$, $\lambda_{k'}^0$ (from $\rho(k', \vec{x}')$), $\vec{\mu}_t$, and μ_t^0).

Finally, concatenating all $\mathcal{S}_t$ for all transitions $t \in \mathcal{T}$ gives a linear system $\mathcal{S}$, with nonnegative unknowns $(\vec{\lambda}_k, \lambda_k^0)_{k \in \mathcal{K}}$ and $(\vec{\mu}_t, \mu_t^0)_{t \in \mathcal{T}}$, that characterizes all one-dimensional affine ranking functions for the automaton.

Example 1 (Cont'd) As explained before, Aspic provides the following invariants (the initial values are denoted by r_0, da_0, db_0):

- $P_{init} = \{1 \leq r_0 = r, db = db_0, da = da_0\}$
- $P_{loop} = \{da \leq 2r, db \leq 2r, r \leq db, r - 1 \leq da, 1 \leq r = r_0\}$
- $P_{end} = \{da < r, db \leq 2r, r \leq db, r - 1 \leq da, 1 \leq r = r_0\}$

For readability in the following systems, for $i \geq 1$ and $k \in \mathcal{K}$, we write λ_k^i instead of $\vec{\lambda}_k[i - 1]$, same for $\vec{\mu}$. Also $\vec{x}$ stands for the vector $(da, da_0, db, db_0, r, r_0)$. The subsystem (4) is obtained by applying Farkas lemma to P_{init}, P_{loop}, and P_{end} (here, with no simplification):

(4i) $\rho(init, \vec{x}) = \lambda_{init}^0 + \lambda_{init}^1(r-1) + \lambda_{init}^2(r_0-1) + \lambda_{init}^3(r_0-r) + \lambda_{init}^4(r-r_0) + \lambda_{init}^5(db_0 - db) + \lambda_{init}^6(db - db_0) + \lambda_{init}^7(da_0 - da) + \lambda_{init}^8(da - da_0)$ with $\lambda_{init}^i \geq 0$ for all $i \in [0..8]$;

(4ii) $\rho(loop, \vec{x}) = \lambda_{loop}^0 + \lambda_{loop}^1(2r-da) + \lambda_{loop}^2(2r-db) + \lambda_{loop}^3(r-1) + \lambda_{loop}^4(db-r) + \lambda_{loop}^5(da-r-1) + \lambda_{loop}^6(r_0-r) + \lambda_{loop}^7(r-r_0)$ with $\lambda_{loop}^i \geq 0$ for all $i \in [0..7]$;

(4iii) and a similar formula for the control point end.

To get the subsystem (5), we first compute Q_t (here in logical form):

- $Q_{t_1} = \vec{x} \in P_{init} \wedge da' = 2r \wedge db' = 2r \wedge r' = r$;
- $Q_{t_2} = \vec{x} \in P_{loop} \wedge r \leq da \wedge da' = da - 1 \wedge db' = db \wedge r' = r$;
- $Q_{t_2} = \vec{x} \in P_{loop} \wedge r \leq da < db \wedge da' = db - 1 \wedge db' = da \wedge r' = r$.

And, finally, we obtain:

(5i) $\rho(init, \vec{x}) - \rho(loop, \vec{x}') - 1 = \mu_{t_1}^0 + \mu_{t_1}^1(r_0 - 1) + \mu_{t_1}^2(r - r_0) + \mu_{t_1}^3(r_0 - r) + \ldots + \mu_{t_1}^7(da' - 2r) + \mu_{t_1}^8(2r - da') + \mu_{t_1}^9(db' - 2r) + \mu_{t_1}^{10}(2r - db') + \ldots$ with $\mu_{t_1}^i \geq 0$ for all i;

(5ii) and similar expressions for transitions t_2 and t_3.

In (5i), we replace the expressions involving ρ by the values obtained in (4i) and (4ii). We do the same in the other expressions coming from (5). We obtain a conjunction of conditions involving the different $\vec{\lambda}_k$ and $\vec{\mu}_k$. The solver finally produces:

- $\rho(init, \vec{x}) = 2r_0 + 3$;
- $\rho(loop, \vec{x}) = 2 + da + db - 2r$;
- $\rho(end, \vec{x}) = 0$.

which means that the loop terminates in at most $2r_0 + 3$ steps (including at most $2r_0 + 1$ iterations of the while loop). The quantity $i = 2 + da + db - 2r$, which is automatically extracted, can be viewed as a kind of counter for the while loop.   □

Solving the system $\mathcal{S}$ gives a termination test, providing the invariants are accurate enough. Note however that, since ρ is, so far, affine and one-dimensional, there is no hope, with this method, to be able to determine the termination of an automaton whose *worst-case computational complexity* (WCCC), i.e., maximal trace length, is more than linear in the input variables. The extension in the next section can determine the termination of some programs with a non-linear (but still polynomially-bounded) WCCC.

3.2 Multi-dimensional affine ranking functions

Using multi-dimensional affine ranking functions, i.e., with co-domain $(\mathbb{N}^d, \leq_d)$ for some $d \in \mathbb{N}$, extends the set of programs whose termination can be determined. Furthermore, when it exists, a polynomial WCCC can be derived from the ranking, with a simpler method than by manipulating directly polynomials of degree d.

For a d-dimensional ranking function ρ, the decreasing constraint expressed by Inequality (1) becomes:

$$(\vec{x}, \vec{x}') \in Q_t \Rightarrow \Delta_t(\rho, \vec{x}, \vec{x}') >_d \vec{0} \tag{6}$$

which means that $\Delta_t(\rho, \vec{x}, \vec{x}') \neq \vec{0}$ and its first nonzero component is positive. The difficulty is then to decide, for each transition, at which dimension the first nonzero component occurs. For that, the multi-dimensional technique of [8] relies on an exponential technique, potentially exploring all possible mappings of transitions to dimensions. We proceed differently, using a greedy mechanism as in [25, 18, 13]. Our technique is provably complete (Theorem 1).

Unlike for a one-dimensional ranking function where we look for a function ρ such that $\rho(k, \vec{x}) - \rho(k', \vec{x}') \geq 1$, here we consider each dimension $\sigma = \rho[i]$ of ρ with the relaxed constraint:

$$(\vec{x}, \vec{x}') \in Q_t \Rightarrow \sigma(k, \vec{x}) - \sigma(k', \vec{x}') \geq \varepsilon_t \tag{7}$$

with $0 \leq \varepsilon_t \leq 1$. As we did in Section 3.1, we obtain a linear system $\mathcal{S}_t(\vec{\lambda}_k, \lambda_k^0, \vec{\lambda}_{k'}, \lambda_{k'}^0, \vec{\mu}_t, \mu_t^0, \varepsilon_t)$ for each transition, this time with the additional unknown ε_t. Then, considering all transitions, we get a global system $\mathcal{S}$. For a solution σ of $\mathcal{S}$, we say that a transition t is *satisfied* if $\varepsilon_t = 1$, otherwise, it is *respected*.

To build a multi-dimensional ranking ρ, we use a greedy algorithm, similar to the scheduling algorithm of [18], that builds the different dimensions of ρ, one after the other, from dimension 0 to $d - 1$, respecting unsatisfied transitions until all are satisfied for some dimension. Furthermore, for each dimension, we try to satisfy as many transitions as possible by maximizing the number of ε_t equal to 1. This boils down to solving the following optimization:

$$m = \max \left\{ \sum_{t \in \mathcal{T}} \varepsilon_t \mid \bigwedge_{t=(k,g,a,k') \in \mathcal{T}} \mathcal{S}_t(\vec{\lambda}_k, \lambda_k^0, \vec{\lambda}_{k'}, \lambda_{k'}^0, \vec{\mu}_t, \mu_t^0, \varepsilon_t) \right\} \tag{8}$$

Any solution σ leading to $\varepsilon_t > 0$ can be multiplied by a suitable positive constant to get a solution with $\varepsilon_t = 1$. Thus, for any *optimal* solution of (8), a transition t has
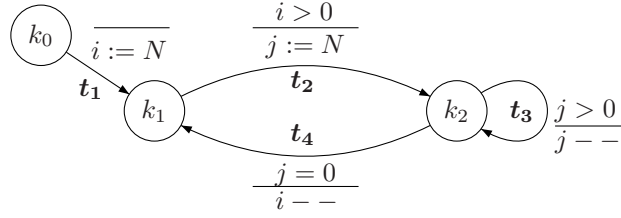
Figure 2: Automaton with a multi-dimensional ranking

either $\varepsilon_t = 0$ or $\varepsilon_t = 1$. Similarly, we can build an integer optimal solution from any rational optimal solution. The variables $(\vec{\lambda}_k)_{k \in \mathcal{K}}$ define an affine function that we use as the first dimension of ρ. By construction, $\rho[0]$ satisfies (3) for every transition t such that $\varepsilon_t = 1$. Other transitions are simply respected ($\varepsilon_t = 0$). We process them in the subsequent dimensions.

For that, we build a new system $\mathcal{S}$, obtained by concatenating the different $\mathcal{S}_t$, but considering only the transitions t for which $\varepsilon_t = 0$ in the previous dimensions, and we iterate the process. This way, we define $\rho[1]$, $\rho[2]$, and so on. When none of the remaining transitions can be satisfied, i.e., $m = 0$ in the system (8), the algorithm stops and no multi-dimensional ranking function is found. Otherwise, as the number of transitions is finite, all of them are eventually satisfied, after a finite number of steps, by some dimension of ρ. In this case, the algorithm outputs, for each $k \in \mathcal{K}$, an affine function $\rho(k, .) : \mathbb{Z}^n \to \mathbb{N}^{d_k}$, where d_k is the number of successive systems, involving the control point k, that were solved. To simplify the notations and the discussion, we can complete each $\rho(k, .)$, with an arbitrary constant value in the remaining dimensions, so that they all have co-domain $\mathbb{N}^d$, with $d = \max_k d_k$. This defines an affine ranking function ρ of dimension d.

Example 2 Consider the automaton of Figure 2 with the context $N \geq 0, N = N_0, i = i_0, j = j_0$ in the unique entry control point k_0. Aspic finds the following invariants:

- $P_{k_0} = \{0 \leq N = N_0, j = j_0, i = i_0\}$;
- $P_{k_1} = \{0 \leq N = N_0, 0 \leq i \leq N\}$;
- $P_{k_2} = \{0 \leq N = N_0, 1 \leq i \leq N, 0 \leq j \leq N\}$.

Solving the system while maximizing $\varepsilon_{t_1} + \varepsilon_{t_2} + \varepsilon_{t_3} + \varepsilon_{t_4}$ leads to $\varepsilon_{t_1} = \varepsilon_{t_2} = \varepsilon_{t_4} = 1$ and $\varepsilon_{t_3} = 0$. The corresponding function (first dimension of the ranking) is $\rho(k_0, \vec{x})[0] = 2N + 1$, $\rho(k_1, \vec{x})[0] = 2i$, $\rho(k_2, \vec{x})[0] = 2i - 1$. We keep the transition t_3 for the next dimension, and we get $\varepsilon_{t_3} = 1$ with $\rho(k_2, \vec{x})[1] = j$. The complete ranking function is thus $\rho(k_0, \vec{x}) = 2N + 1$, $\rho(k_1, \vec{x}) = 2i$, $\rho(k_2, \vec{x}) = (2i - 1, j)$. Note that k_0 and k_1 have a 1D-ranking. It can be extended arbitrarily, for example as $\rho(k_0, \vec{x}) = (2N + 1, 0)$ and $\rho(k_1, \vec{x}) = (2i, 0)$, if a globally 2D ranking function is desired. $\qquad \square$

4 Properties and extensions

In the previous section, we presented the general principles of our ranking algorithm. We now discuss some of its features, in particular its completeness for the class of multi-dimensional affine rankings (see Definition 3), how it can be adapted to exploit the notion of cutpoints, how it can be optimized in terms of implementation, and how it can be used to compute upper-bounds for the maximal trace lengths (worst-case computational complexity WCCC).

4.1   Completeness

Since non-terminating programs exist, there is no hope of proving that a ranking function always exists. Moreover, there are terminating affine interpreted automata with no multi-dimensional affine ranking. Thus, all we can prove is that, if a multi-dimensional affine ranking exists, our algorithm finds one, i.e., it is complete for the class of multi-dimensional affine rankings. Also, as the sets $\mathcal{R}_k$ are over-approximated by invariants $\mathcal{P}_k$, completeness has to be understood with respect to these invariants. In this section, we just sketch the completeness proof. The proof itself, quite long and technical, can be found in [1].

The knowledgeable reader may have noticed a similarity with the algorithm of [13]. However, as pointed out in Section 1, the class of ranking functions we consider is larger. At each level of the decomposition, each control point k can have its own affine ranking function $\vec{x} \to \rho(k, \vec{x})$ while the algorithm in [13] looks for a single affine expression that should be non-increasing for all transitions of the strongly connected component being considered. Thus, our algorithm is, in principle, more powerful unless we get lost in a too large search space. Its completeness guarantees this is not the case.

Theorem 1 *If an affine interpreted automaton, with associated invariants, has an affine ranking function, then the algorithm of Section 3.2 finds one and its dimension is minimal.*

To summarize the proof, we start from an affine ranking of dimension d. We show that there is an affine ranking of dimension d that fully satisfies at least one transition. This proves that our algorithm does not abort and generates a one-dimensional ranking σ. Then, we show that there is an affine ranking of dimension d whose first dimension is σ. Finally, we show that there is an affine ranking of dimension d, whose first dimension is σ, and such that the $d - 1$ last dimensions satisfy all transitions not fully satisfied by σ. Iterating the process, this shows our algorithm terminates and generates an affine ranking of dimension $\leq d$, for any possible dimension d.

4.2   Cut points

Floyd [28] uses a condition weaker than Condition (1): let $\mathcal{H}$ be a subset of $\mathcal{K}$ such that any cycle of the automaton contains at least one control point in $\mathcal{H}$ (a cutpoint set or cutset). Then to prove termination, it is enough to exhibit a function into a well-founded set, which decreases on all paths from one point in $\mathcal{H}$ to another. The existence of such a function implies that after a finite time, no cutpoint can be visited again. Hence cutpoints can be removed from the flowgraph, which becomes acyclic and hence terminates.

The fact that a ranking satisfying Condition (1) can also serve in this context is obvious. Conversely, one can prove that given a ranking function satisfying Floyd condition, one can build another ranking satisfying Condition (1). However, this ranking may not be affine, but only piecewise affine (it will use the max operator).

Another point is that the two methods can be subsumed into one algorithm by transforming the source flowchart. Once the set of cutpoints is known, we can eliminate all other control points with the exception of the start and stop points, by compressing the effects of paths between cutpoints. The method is best explained by reference to Figure 3(a). Assume we want to eliminate the control point C. The result is Figure 3(b). If, for example, the transition A to C has guard g and action a, and the transition from C
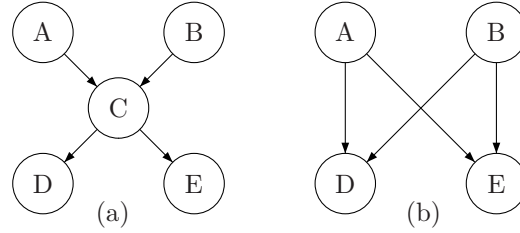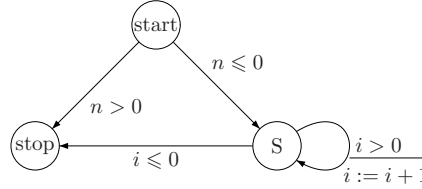
Figure 3: State elimination

Figure 4: A cutpoint construction

to D has guard g' and action a', then the new transition A to D has guard $g \wedge (g' \circ a)$, and action $a' \circ a$. A ranking satisfying Condition (1) for the compressed automaton satisfies the Floyd condition for the original automaton. Note that in our model, a transition with an undefined action (a question mark "?") cannot be eliminated. Hence, its target control point must be included in the cutset.

The compression of paths (combining guards and actions as we just explain) to keep only cutpoints has an effect on the complexity of the ranking algorithm. On one hand, it reduces the number of control points and thus the number of variables in the ranking algorithm (remember that each control point has its own function). On the other hand, the number of paths to compress can grow fast and so does the number of transitions in the compressed automaton. Thus, in terms of complexity, path compression must be used with care. However, in terms of quality of results, more affine rankings can be found for the compressed automaton (if no approximation was done during path compression) than for the original automaton. One of the reasons is that imposing the ranking to be nonnegative everywhere is too strong a constraint. We have indeed found that it precludes solving codes like:

```
for(i=n; i>0; --i) S;
```

in the absence of information on n. In fact, the natural ranking would be $n + 1$ at the start of the program, i inside the loop, and 0 at the exit point, but it is not acceptable if n is negative. However, this program terminates even if $n \leq 0$, and we should be able to prove it without any modification of the program text. Other scholars either need to insert $n > 0$ as a precondition, or use tail invariants [13]. Another solution is to use a cutpoint, here the point corresponding to statement S. The resulting automaton is depicted in Figure 4. Our algorithm then easily finds a two-dimensional ranking:

$$\rho_{\text{start}} = (2, 0), \ \rho_{\text{S}}(i) = (1, i), \ \rho_{\text{stop}} = (0, 0).$$

So far, we explained how cutpoints can be exploiting by modifying the automaton, before searching for affine rankings. Another possibility is to try to exploit the notion of cutpoints in the ranking algorithm itself. First, as we just mentioned, it is enough to make sure that the ranking function is nonnegative for the invariants that correspond

to cutpoints. Furthermore, there is also no need to impose Inequality (6) to be strict for all transitions. If at least one transition is strictly decreasing for each cycle, this is enough to guarantee termination. Unlike the first modification, this second modification does not enable to prove the termination of more programs but the resulting ranking functions are more likely to be "simpler" and to give more accurate results for WCCC computations. To integrate the second modification in the ranking algorithm, a possibility is to proceed as follows. At each level of the decomposition, we first identity the set $\mathcal{T}_1$ of transitions t for which $\varepsilon_t = 1$. We then select a subset $\mathcal{T}'_1$ of $\mathcal{T}_1$ that cover all cycles formed by the transitions in $\mathcal{T}_1$. We then solve again the inequalities of (8) (optimizing some other criterion than m), fixing $\varepsilon_t = 1$ if $t \in \mathcal{T}'_1$ and $\varepsilon_t = 0$ otherwise. We can also rely on the computation of strongly connected components to remove transitions in acyclic parts of the automaton.

4.3   Scalability

The size of the system (8) is roughly proportional to the number of transitions in the automaton. This may be too much for the underlying linear solver. However, the situation can be improved along the lines of [19]. Notice first that each transition has its own set of Farkas multipliers, the $\vec{\mu}_t$ of (5). These multipliers can be eliminated one transition at a time, thus leaving only the $\vec{\lambda}_k$ as unknowns. Furthermore, the unknowns in the constraint system for a transition pertain only to the source and sink of the transition. Hence, the grand constraint system (8) is a block representation of the incidence graph of the automaton: each system (5) generates a block of rows in which only the columns corresponding to the unknowns for the source and sink of transition t are nonzero. These unknowns can be successively eliminated using various heuristics (e.g., eliminate first the unknowns of a control point of minimum degree), leaving only a system of constraints on the ε_t, which is then solved for the maximum σ. All $\vec{\lambda}$ are then recovered by a process of back substitution, with no need to compute the $\vec{\mu}_t$.

4.4   Worst-case computational complexity (WCCC)

As shown in the survey by Wilhelm et al. [33], the computation of a worst-case execution time (WCET) is a highly-complex affair, as it has to take into account the program, its data, and the processor on which it is run. Handling all these complexities is beyond the scope of this paper. Our aim is to evaluate an *abstract* WCET, as would be observed on a processor with a perfectly additive timing model, executing one automaton transition in unit time. We call this quantity the *worst-case computational complexity* of the program (WCCC). Such an estimate can be useful, for example as a template with unknown coefficients, to be fitted to actual measurements by a process of regression. It is also standard in high-level synthesis to need an upper-bound on the number of loop iterations (`do` loops as well as `while` loops), to enable scheduling optimization at higher level. We thus define the WCCC as an upper bound on the number of transitions executed, given an initial value of the counter variables. Note that the WCCC is significant only up to a constant factor. For example, if we eliminate a state as shown in Figure 3, the semantics of the flowchart will not be materially changed, but the WCCC may decrease.

   With this definition, one could over-approximate the WCCC by the total number of reachable states, i.e., WCCC $\leq \sum_k \#\mathcal{R}_k$ or even more conservative WCCC $\leq \sum_k \#\mathcal{P}_k$ as $\mathcal{R}_k$ is itself over-approximated by $\mathcal{P}_k$. (Here, the notation $\#S$ means the number of integer points in a set S.) This is a very rough over-approximation but, even worse,

this technique can lead to an infinite WCCC, even for a terminating automaton, if some $\mathcal{P}_k$ is unbounded. Rather, we can use the ranking function itself to prune the invariant sets. Indeed, consider a trace $(k_0, \vec{x}_0), \ldots, (k_p, \vec{x}_p)$ in the execution of the automaton. By definition of a ranking function, $\rho(k_{i+1}, \vec{x}_{i+1}) \prec \rho(k_i, \vec{x}_i)$. Since $\prec$ is a strict order, it follows by transitivity that all $\rho(k_i, \vec{x}_i)$ are distinct in $\mathcal{W}$. Hence, the length of the trace is bounded by the cardinal of the co-domain of ρ:

$$\text{WCCC} \leq \# \bigcup_k \rho(k, \mathcal{P}_k) \leq \sum_k \#\rho(k, \mathcal{P}_k) \tag{9}$$

The first inequality is more accurate but harder to compute as it involves a union of sets. Let us see how we can compute $\#\rho(k, \mathcal{P}_k)$ for a given control point k. To make notations simpler, we drop the index k: we let $\rho(k, \vec{x}) = \rho(\vec{x}) = R\vec{x} + \vec{r}$ and $\mathcal{P} = \mathcal{P}_k$. To compute $\#\rho(\mathcal{P})$, we can ignore the constant vector $\vec{r}$. The number of different values in $\rho(\mathcal{P})$ is then the number of points in the image of a $\mathbb{Z}$-polyhedron (intersection of an integer lattice, here $\mathbb{Z}^n$, and a polyhedron, here $\mathcal{P}$) by an affine function. Such problems have already been studied in the literature using various techniques related to Ehrhart polynomials [11, 32]. To make things simpler, we prefer to over-approximate $\#\rho(\mathcal{P})$ as follows. We compute $R = USV$, the Smith normal form of R, where U and V are unimodular, $S = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$, and D is a diagonal positive matrix of rank d (the rank of R). Let $\mathcal{V}$ be the polyhedron obtained by projecting the polyhedron $V\mathcal{P}$ on its d first coordinates. Then $\#\mathcal{V}$ is a slight over-approximation of $\#\rho(\mathcal{P})$. Indeed, two vectors $\vec{x}$ and $\vec{y}$ in $\mathcal{P}$ have the same image by ρ if and only if $V\vec{x}$ and $V\vec{y}$ have the same d first components. The over-approximation comes from the fact that, in very specific cases, not all integral vectors in $\mathcal{V}$ are obtained by projection of an integral vector in $V\mathcal{P}$. The number of integral vectors in $\mathcal{V}$ is then computed using Ehrhart polynomials.

From this algorithm, interesting properties of the WCCC follow. For instance, multiplying a ranking by a positive constant only multiply the diagonal elements of D by the same constant, with no influence on the WCCC. However, it is important to minimize the rank of D because the WCCC will tend to be smaller if the dimension of $\mathcal{V}$ is smaller. This is why it is important to generate rankings of minimal dimension as our algorithm does (Theorem 1). However, adding linearly dependent components to the ranking will simply add null rows at the bottom of S. From this follows that the WCCC will at most be $O(size^n)$ since it is impossible to build more than n linear forms on n variables. This bound cannot be improved, since with n variables, one can write a system of n perfectly nested loops, which achieves the required complexity.

We point out that, so far, we compute the WCCC according to the rightmost expression in (9). A more precise evaluation can be obtained by first expanding the union in (9) into a disjoint union before computing Ehrhart polynomials. This is left for future work.

Example 2 (Cont'd) In this very simple, fully-deterministic, example, the two inequalities of (9) lead to the same upper bound $\text{WCCC} \leq 1 + (N_0 + 1) + N_0(N_0 + 1) = 1 + (N_0 + 1)^2$. □

The factors affecting the precision of the WCCC, beside the union computation, are the presence of non affine guards and of non affine domains. Consider first the code:

```
for(i=0; i<n && f()); i++) S;
```

Our tool will find a WCCC of $O(n)$, but the real value may be much less, depending on the properties of the function f. Consider now:

```
for(i=0, j=n; j>2; ++i, j/=2);
```

The invariant is $2 \leq j \leq n$ and the ranking is j, which gives a WCCC of n instead of the correct value $\log_2 n$. Here, the domain of j is grossly over-estimated by a polyhedron.

5 Implementation and experimental results

We have built a tool suite that converts a C program into an integer interpreted automaton, constructs its invariants, tests its termination and, if successful, computes an upper bound for its WCCC.

The first tool, the preprocessor c2fsm, is just an exercise in parsing and control graph construction. This tool also implements dead code elimination, useless variables elimination, and, as an option, the selection of cutpoints and the elimination of other control points. The main difficulties come from the complexity of the C syntax and semantics, and mainly from the convention that an assignment can occur at any depth in an expression. Our guidelines have been to consider only assignments to integer variables, and to give a variable the bottom value unless it is assigned an affine form in other integer variables. The fact that C has no Booleans is both a simplification and a hindrance, as it forbids the use of some of the techniques that were developed for synchronous languages compilation [6]. We plan to extend c2fsm by using gcc as a front end. It may even be possible to extract flowcharts from binaries or assembly code, thus greatly extending the scope of the method.

The result of the analysis performed by the C preprocessor is then presented as a system of control points and transitions in the input format of Aspic, which is responsible for the computation of invariants. The reader is referred to [21] for a description of Aspic.

Finally, the ranking algorithm and the WCCC computations are implemented in a tool called Rank. The minor premise of the affine form of Farkas lemma (the fact that the polyhedron is non-empty) and the linear program (8) are solved by means of the PIP tool (Parametric Integer Programming), now wrapped in the Piplib library [1]. The Ehrhart polynomials part of the Polylib library is then used for upper-bounding the computational complexities (WCCC).

This tool chain has been tested on a set of benchmarks from the literature. Most of the examples were collected in [10] from many other papers dealing with termination analysis. They can be found at: `http://www.dcs.qmul.ac.uk/~aziem/esop/`. The source code for all the examples (including ours) can be found in the extended version of this paper. The results are summarized in Table 1.

The first two columns identify the test cases. The symbol ♣ indicates a test case we developed to check our algorithm. Columns 3 to 5 give statistics about the (generated) automaton: number of relevant variables, of control points, of transitions. The next column gives the dimension of the ranking function found by our algorithm. The next column gives the timing measurements on a 2 GHz Pentium with 1 GByte of memory running Debian 2.6. The "Analysis" measures include the invariants computation time from the Aspic file, the computation of the ranking function, and the evaluation of the WCCC. In general, the WCCC is the maximum of several parametric expressions valid on different domains of the program imputs. To make the table simpler, the last column gives only the expression that can reach the maximum value.

---

[1]`piplib.org`

Name	Ref	Vars	States	Trans	$\dim \rho$	Time (s)	WCCC
easy1	[10]	3	4	5	1	0.2	43
easy2	[10]	3	3	3	1	0.07	$z_0 + 3$
ackermann	[4]	2	7	7	1	0.07	$4m_0 + 5$
terminate	[12]	3	1	1	1	0.08	$102 + k_0 + j_0 + i_0$
gcd	[7]	2	5	1	1	0.2	$y_0 + x_0 + 2$
rsd	♣	3	3	4	1	0.2	$5 + \dfrac{5r_0}{2} + \dfrac{r_o^2}{2}$
nd_loop	♣	2	4	6	1	0.05	22
wcet2	♣	2	3	5	1	0.15	$62 - 12i_0$
relation1	♣	2	4	4	1	0.14	4
ndecr	♣	2	4	4	1	0.08	$i_0 + 2$
perfect	[9]	4	5	12	3	0.35	$2 + \dfrac{3x_0}{2} + \dfrac{x_0^2}{2}$
cousot16	[15]	2	3	4	1	0.05	106
random2d	[10]	5	10	21	1	1.1	$6N_0 + 3$
random1d	[10]	3	4	6	2	0.1	$max + 3$
wise	♣	2	6	10	2	0.11	$1 + |x_0 - y_0|$
wcet1	♣	3	6	8	2	0.25	$n_0 + 2$
complex	[23]	2	4	11	2	0.28	$1560 - 9b_0 - 45a_0$
nestedLoop	[23]	6	5	12	3	1.3	$n_0 m_0 + 2N_0 + n_0 + 3$
exmini	♣	4	3	6	2	0.1	$104 + k_0 - j_0 - x_0$
aaron2	[10]	3	6	10	2	0.2	$2(x_0 - y_0) + 5$
while2	♣	3	3	4	2	0.1	$3 + N + N^2$
cousot9	[15]	3	4	5	2	0.2	$j_0 + 3$
ax	♣	4	3	6	3	0.16	$n_0^2 + n_0 + 3$
loops	[29]	3	4	5	2	0.15	$N^2 - N + 5$
counterex1	♣	4	5	13	3	1.1	$x_0 + 2$
determinant	[9]	4	6	7	4	0.15	$\dfrac{N^3}{3} + \dfrac{N^2}{2} + \dfrac{N}{6} + 3$
maccarthy91	[13]	4	5	18	2	1.2	13773/11 - 1363/110 x
Sorting programs							
realselect	♣	7	5	10	3	0.4	$\dfrac{N^2}{2} + \dfrac{3N}{2} + 1$
insertsort	♣	3	6	7	2	0.22	$\dfrac{N^2}{2} + \dfrac{3N}{2} + 1$
sipmabubble	[10]	4	10	17	3	0.33	$N^2 + 2N + 3$
realbubble	♣	6	5	11	3	0.4	$N^2 + 2$
realshellsort	♣	8	8	13	4	1.1	$\dfrac{N^3}{6} - \dfrac{N}{6}$
realheapsort	♣	10	11	31	3	2.8	$4N^2 - 11N + 9$

Table 1: Experimental results

In all test cases of Table 1, we were able to prove termination, even for nondeterministic examples like `random2d`. Nested loops are correctly handled, and we find multi-dimensional rankings for them. The case of recursion is often handled by making assumptions about (the values of) the variables after a recursive call (for instance we assume the result of `ackermann` is positive).

We were also able to prove the termination of some classical sortings algorithms. The rankings for these codes may seem of the wrong dimensions, but the additional dimensions have constant values and the order of magnitude of the WCCC are still as expected, e.g., $O(N^2)$ for `bubblesort`. The WCCC are given as an piecewise function depending on the initial values of the variables. For `wise`, we get $1 + x_0 - y_0$ if $x_0 \geq y_0$ and $1 + y_0 - x_0$ otherwise, that we simplified as $1 + |x_0 - y_0|$ for the sake of clarty.

For `heapsort`, our algorithm finds a WCCC of order N^2 instead of the correct $O(N \log_2 N)$, see Section 4.4 for an explanation. However, we are for the moment unable to prove the termination of `mergesort` due to scalability reasons. We also point out that our algorithm for the elimination of useless variables is still rudimentary. We therefore manipulate polyhedra of dimensions higher than needed. In addition, since the termination of concurrent programs sometimes depends on a fairness hypothesis, we were unable to solve some of the examples of [29]. We found the precision of our algorithm to be strongly dependent on the quality of the invariants, and also the quality of the affine approximation of some (non affine) affectations in the C programs.

6   Related work

Using ranking functions to prove correctness was first proposed in [20]. Early approaches were semi-automatic: one had to guess ranking functions, and then prove their correctness using some form of Hoare logic. Attempts to automate this process followed [12, 13, 8]. It was then realized that one-dimensional rankings were not powerful enough, and propositions to build multi-dimensional [7] or polynomial rankings [15] followed. We believe that our method (which was suggested by Feautrier's work on scheduling [18]), is a satisfactory solution to this problem. As mentioned in Section 1, we extend references [13, 29, 8] in several directions.

First, unlike [29, 8], we are not limited to one loop, i.e., our automaton can have an arbitrary number of vertices (as in reference [13]). This is mandatory to be able to analyze complex loops, either nested loops, or multi-path simple loops which have been transformed into an automaton with several vertices by path-sensitive analysis. Abstracting a loop with multi-paths into a single loop with multiple simple transitions may be possible, as we did with path compression (Section 4.2) but the feasibility and cost of such a transformation in general is unclear. Also, even if the semantics is the same, the fact that there will be a unique (in the formalism of [8]) ranking function for all transitions is a strong limitation for proving termination. Consider the following simple example:

```
while (x>=0 && 0<=y<=n) {
    if (b==0) {
      y++;                     /* transition t1 */
      if (random()) b=1;       /* transition t3 */
    }
    else {
      y--;                     /* transition t2 */
      if (random()) {x--; b=0;} /* transition t4 */
    }
}
```

$$b = 0, 0 \leqslant x, y \leqslant n$$
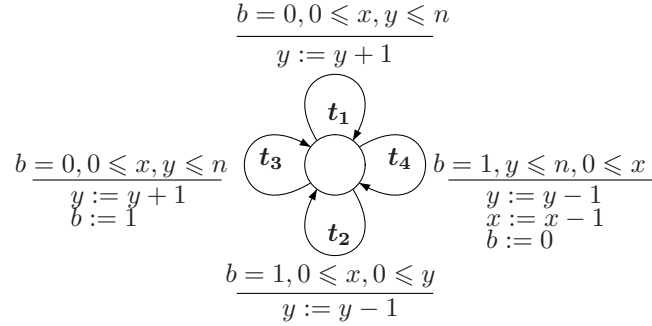$$\overline{\quad y := y + 1 \quad}$$

Figure 5: A one control point automaton

If this loop is encoded with a single control point with multiple parallel transitions, as in Figure 5, then, looking for a single ranking function, even multi-dimensional, is not enough to prove termination. Indeed, transitions $t1$ and $t2$ are contradictory and it is not possible to find a function decreasing for one and non-increasing for the other one. To analyze such a loop, we propose to analyze the Boolean value "b" and abstract it in

Figure 6: A two control points automaton

the control to define an automaton with more than one state (this is standard technique). The resulting automaton (see Figure 6) has two control points and can be proved to terminate with the following ranking functions:

$$c0 \; : \; (2x + 1, n - y), \;\; c2 \; : \; (2x, y).$$

The situation is the same for the following nested loops example:

```
while(x>=0){
    while(y>=0 && random()) y--;
    x--
    while(y<=n && random()) y++;
}
```

These two situations, nested loops and simple loops with multiple states obtained after analysis of Booleans (or other variables with finite domains), illustrate why it is mandatory to be able to consider automata with more than one vertex, thus ranking functions with a different function for each vertex.

Second, looking for a unique lexicographic affine function as in [8] is much easier than the general case we address. Indeed, one has just to decide for which dimension of the ranking function the transition decreases (it must be non-increasing in the previous dimensions), i.e., in our terminology, at which depth each transition is "satisfied". The algorithm `has_llrf` in Figure 2 of reference [8] is a recursive exploration (a kind of

branch-and-bound technique, with a linear program solving at each node), thus potentially exponential, while ours execute only one call to the LP solver per dimension of the ranking function. Since the algorithm is potentially exhaustive, there is no need to prove completeness. In our case, since our algorithm is not exhaustive but greedy, a completeness proof is needed to show that we do not lose any solution. Furthermore, our completeness proof is an order of magnitude more general since we are able to deal with the much larger space of multi-dimensional affine ranking functions, not just one single lexicographic function.

Third, unlike previous papers, we are able to prove that we get the smallest number of dimensions for each ranking function. In [7], the authors do notice that they may have as many dimensions as the number of transitions. As explained in Section 4.4, this dimension reduction is important for the computation of the WCCC.

In a different context, a large body of research followed the introduction of the size change termination (SCT) principle in [27]. The difference in the two approaches are mainly in semantics: the automaton represents a call graph instead of a control graph, and the variables may be summary information about data structures, like the length of a list or the size of a tree. More importantly, the relations between input and output variables of a transition are restricted to one of the two forms $x' < y$ and $x' \leq y$. An attempt to lift this restriction can be found in [2]. In [3], it is shown that termination of an SCT system can be proved using an exponential number of very simple local ranking functions, or with a global ranking function involving an exponential number of subterms. This is in contrast to a very simple consequence of Theorem 1 that the dimension of our ranking function is no larger than the number of transitions (and even no larger than the number of variables with some stronger hypotheses [17]). The explanation is probably that the two sets of programs for which our algorithm succeeds and for which the SCT formalism succeeds are almost disjoint.

Another trend of research has been started in [30] and pursued in [10]. Here, one uses several (local) ranking relations, all of them well founded, the intuition being that each relation proves termination of a part of the program. A consistency condition is necessary: the transitive closure of the transition relation of the program must be included in the union of all local ranking relations. The problem is how to find the local rankings, and how to prove the consistency condition. It may be that we can help at least for the first problem: apply our algorithm to cleverly chosen subsets of the automaton states, as for example strongly connected components or loops.

As for [23, 24], their approach is very different from ours. They either introduce counters, and hope to find bounds for them through the computation of invariants, or use pattern matching for extracting loop bounds. In our case, we find the counters a posteriori, through the different dimensions of our ranking functions.

7   Conclusion

7.1   Contributions

The first main contribution of this paper is the design of an algorithm for the construction of multi-dimensional affine ranking functions, which, in contrast to the combinatorial algorithm of [7], is greedy but nevertheless complete (with respect to the invariants found and the class of ranking functions considered) and optimal in the dimension of the ranking function. The algorithm makes no assumption whatever on the shape of the source program, and can handle, with proper preprocessing (i.e., after the program

is approximated to fit into the affine interpreted automaton model), multiple loops of arbitrary nesting patterns, premature termination and goto's, nondeterministic choices and values, exceptions, and Boolean guards of arbitrary structure. We also point out that, in case of failure, our algorithm can also exhibit an execution trace which may not terminate, but all the details are not worked up yet.

The computation of the worst-case computational complexity (WCCC) is delegated to a very comprehensive stand-alone algorithm. This means that no arbitrary restrictions about the shape of loops and tests are necessary. We can directly rely on existing methods and tools for counting integer points within $\mathbb{Z}$-polyhedra and images of $\mathbb{Z}$-polyhedra by affine functions. More generally, our work establishes a strong link with computation models, theoretical results, and tools, developed by the automatic parallelization and high-performance computing community, and which seem to be not so used (or partly re-discovered) in the context of program termination. We believe that this connection can lead to further fruitful developments to face other problems faced by both communities.

7.2 Future work

There is nevertheless room for many improvements. The preprocessor we use for converting a program into an interpreted automaton is somewhat brute force: any construct that is not affine in integer variables is replaced by the bottom value, which is absorbing ($\bot \oplus x = \bot$ for most operators), and which prints as `true` in a guard and as a question mark in an action. This can be improved by noticing that some operations, like modulo and integer division, can be linearized by the introduction of fresh variables, or that a bottom value may be constrained: for instance, a square is always non-negative. Also, variables with a finite domain, like Booleans and enums, can be used to refine the states. This may result in a large increase in the size of the automaton but has the direct benefit of extending the class of ranking functions considered, as these do not need to be affine anymore for such "unrolled" variables. Making sure that domains of integer variables are "fat" (to use the terminology of [16]) increases the chance that an affine ranking exists and improves the quality of the WCCC produced.

There is always room for improving an invariant constructor like ASPIC. One may for instance improve the acceleration algorithms and loops treatment, or use additional abstract interpretation frameworks, like the congruences and lattices of [22]. It may also be interesting to construct the invariants *on demand*, both to improve the accuracy and reduce the overhead.

Last but not least, the power of the ranking algorithm can be increased in many ways. For instance, as pointed out in Section 4.2, imposing that ranking functions are nonnegative everywhere (Inequality (2)) is too strong a constraint in many cases, it is enough to impose it at a set of cut points. If the automaton graph becomes acyclic when these cut points are removed, then termination is still guaranteed, notwithstanding the relaxed nonnegativity constraint. In a way, eliminating all states but cutpoints before computing a ranking (by path compression) is equivalent to relaxing the positivity constraint, but it is obtained at the cost of a potential increase in the number of transitions: if the eliminated state has n ingoing and m outgoing transitions, its elimination will generate $n \times m$ transitions. We still need to explore this trade-off and analyze its consequences on the WCCC computations.

Research on the SCT paradigm has shown that ranking functions of a more complex shape, like piecewise affine functions, are necessary in some cases. In our framework,

this means splitting the invariant of some state(s) by an affine constraint. How to choose the states to split and the splitting predicate is left for future research.

There remains the question of interprocedural termination. If there is no recursive call, one may resort to inlining, but this will raise again the question of scalability. One may want to combine the SCT approach with the present one: if there is no infinite path in the call graph, and no infinite path in each function control graph, then termination is guaranteed. However, in many cases, the caller and the callee may interact in complex ways, especially in the presence of side effects, and this will greatly complicate a termination test.

A point we have not investigated is the termination of distributed programs. Our algorithm fails when termination depends on a fairness hypothesis.

References

[1] Christophe Alias, Alain Darte, Paul Feautrier, Laure Gonnord, and Clément Quinson. Program termination and worst time complexity with multi-dimensional affine ranking functions. Technical Report 7037, INRIA, November 2009.

[2] Hugh Anderson and Siau-Cheng Khoo. Affine-based size-change termination. In Atsushi Ohori, editor, *1st Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 122–140, Beijing, 2003. Springer Verlag.

[3] Amir M. Ben-Amram. A complexity tradeoff in ranking-function termination proofs. *Acta Informatica*, 46(1):57–72, 2009.

[4] Amir M. Ben-Amram and Chin Soon Lee. Program termination analysis in polynomial time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1):5, 2007.

[5] Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. *ACM Transactions on Computational Logic (TOCL)*, 2(1):1–11, 2001.

[6] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[7] Aaron A. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1349–1361. Springer Verlag, July 2005.

[8] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In Kousha Etessami and Sriram K. Rajamani, editors, *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *Lecture Notes in Computer Science*, pages 491–504. Springer Verlag, July 2005.

[9] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Termination analysis of integer linear loops. In *16th International Conference on Concurrency Theory (CONCUR)*, volume 3653 of *Lecture Notes in Computer Science*, pages 488–502. Springer Verlag, August 2005.

[10] Aziem Chawdhary, Byron Cook, Sumit Gulwani, Mooly Sagiv, and Hongseok Yang. Ranking abstractions. In *17th European Symposium on Programming (ESOP'08)*, volume 4960 of *Lecture Notes in Computer Science*, pages 81–92, Budapest, April 2008. Springer Verlag.

[11] Philippe Clauss. Handling memory cache policy with integer points counting. In *Parallel Processing, 3rd International Euro-Par Conference*, volume 1300 of *Lecture Notes in Computer Science*, pages 285–293, Passau, August 1997. Springer Verlag.

[12] Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer Verlag, 2001.

[13] Michael A. Colón and Henny B. Sipma. Practical methods for proving program termination. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer Verlag, January 2002.

[14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96, Tucson, January 1978.

[15] Patrick Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation, and semidefinite programming. In *6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24, Paris, January 2005. Springer Verlag.

[16] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[17] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000. ISBN 0-8176-4149-1.

[18] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[19] Paul Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34(5):459–487, May 2006.

[20] Robert W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Symposium on Applied Mathematics*, volume 19, pages 19–32. A.M.S., 1967.

[21] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *13th International Static Analysis Symposium (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160, Seoul, August 2006. Springer Verlag.

[22] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91)*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192, Brighton, 1991. Springer Verlag.

[23] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 375–385, Dublin, 2009. ACM.

[24] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *36th ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 127–139, Savannah, January 2009.

[25] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[26] C. S. Lee. Ranking functions for size-change termination. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(3):1–42, 2009.

[27] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001.

[28] Zohar Manna. *Mathematical Theory of Computing*. MacGraw-Hill, 1974.

[29] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI'03)*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer Verlag, 2004.

[30] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In Harald Ganzinger, editor, *IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41. IEEE Computer Society, July 2004.

[31] A. Schrijver. *Theory of linear and integer programming*. Wiley, NewYork, 1986.

[32] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.

[33] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The determination of worst-case execution times—overview of the methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

A   Source code of kernels

This section provides the source code of the kernels used for the experimental results.
We also recall the execution times given in table 1. The timings were obtained on a
2GHz Pentium with 1GByte of memory running on a Debian 2.6.

easy1 (0.2s)
```
int x,y,z;
x = 0;
y = 100;
while (x < 40)
{
  if (z==0)
    x = x + 1;
  else
    x = x + 2;
}
```

easy2 (0.07s)
```
int x,y,z;
x = 12;
y = 0;
while (z > 0)
{
    x = x + 1;
    y = y - 1;
    z = z - 1;
}
```

ackermann (0.07s)
```
//Automaton entered by hand
//Assuming m>=0 && n>=0
A:
  if (m<=0) //A(0,n) = n+1
    return;
  if (n<=0) { //A(m,0) = A(m-1,1)
    m = m - 1; n = 1; goto A;
  }
  //A(m,n) = A(m-1,A(m,n-1))
  m = m - 1;
  n = random(); //n = A(m,n-1);
  goto A;
```

terminate (0.08s)
```
int i, j, k, ell;
while (i <= 100 && j <= k)
{
  ell = i;
  i = j;
  j = ell + 1;
  k--;
}
```

gcd (0.2s)
```
int x,y;
if(x<=0 || y<=0) return;
while (x != y)
{
  if (x<y) y = y - x;
  else x = x - y;
}
```

rsd (0.2s)
```
int r,da,db,temp;
if (r>=0){
  da = 2*r;
  db = 2*r;
  while (da >= r) {
    if (brandom()){
      da --;
    }
    else{
      temp = da;
      da = db - 1;
      db = temp;
    }
  }
}
```

nd-loop (0.05s)
```
int x,y;
x=0;
do {
  y=x;
  x=random();
  if ((x-y>2) || (x-y<1))
    break;
}
while (x<10);
```

wcet2 (0.15s)
```
int i,j;
while (i<5) {
  j = 0;
  while (i>2 && j<=9) j++;
  i++;
}
```

relation1 (0.14s)
```
int x,y;
do {
  y=x;
  x=random();
  if ((x-y>2) || (x-y<1)) break;
}
while (x<10);
```

ndecr (0.08s)
```
int i,n;
i = n-1;
while (i>1)
{
  i--;
}
```

perfect (0.35s)
```
int y1, y2, y3;
if (x <= 1) return 0;
y1 = y2 = y3 = x;
for(y1=x-1; y1>0; y1=y1-1)
{
  while (y2 >= y1)
    y2 = y2 - y1;
  if (y2 == 0)
    y3 = y3 - y1;
  y2 = x;
}
//return (y3 == 0);
```

cousot16 (0.05s)
```
int i,j;
i = 2;
j = 0;
while (random())
{
  if(j >= 0 && i >= 2*j+2)
    if (brandom()) i = i + 4;
    else {
      i = i + 2;
      j = j + 1;
    }
}
```

random2d (1.1s)
```
int x,y,i,r,N;
x = 0;
y = 0;
i = 0;
while (i<N) {
  i=i+1;
  r=random();
  if (r>=0 && r<=3) {
    if (r==0) x=x+1;
    else if (r==1) x=x-1;
    else if (r==2) y=y+1;
    else if (r==3) y=y-1;
  }
}
```

random1d (0.1s)
```
int a,x,max;
if (max > 0) {
  a = 0;
  x = 1;
  while (x<=max) {
    if (brandom())
      a = a + 1;
    else
      a = a - 1;
    x = x + 1;
  }
}
```

wise (0.11s)
```
int x, y;
if (x<0 || y<0) return;
while (x-y>2 || y-x>2)
{
  if (x < y)
    ++x;
  else
    ++y;
}
```

wcet1 (0.25s)
```
int i,j,n;
j = 0;
i = n;
if (n>=1)
{
  do {
    if (brandom()){
      j++;
      if (j>=n) j = 0;
    }
    else {
      j--;
      if (j<=0) j = 0;
    }
    i--;
  }
  while (i>0);
}
```

complex (0.28s)
```
int a,b;
while(a < 30) {
  while(b < a) {
    if(b > 5)
      b = b + 7;
    else
      b = b + 2;
    if(b >= 10 && b <= 12)
      a = a + 10;
    else
      a = a + 1;
  }
  a = a + 2;
  b = b - 10;
}
```

nestedLoop (1.3s)
```
int i, j, k;
if (0<=n && 0<=m && 0<=N)
{
  i = 0;
  while (i<n){
    j = 0;
    while (j<m){
      j += 1;
      k = i;
      while (k<N)
        k += 1;
      i = k;
    }
    ++i;
  }
}
```

exmini (0.1s)
```
int i,j,k,tmp;
while (i<=100 && j<=k){
  tmp = i;
  i = j;
  j = tmp + 1;
  k = k - 1;
}
```

aaron2 (0.2s)
```
int tx, x, y;
if (tx >= 0) {
  while (x >= y) {
    if (tx < 0) return 0;
    if (brandom())
      x = x - 1 - tx;
    else
      y = y + 1 + tx;
  }
}
```

while2 (0.1s)
```
int i,j,N;
i = N;
if (i > 0) {
  j = N;
  while (j > 0) j--;
}
```

cousot9 (0.2s)
```
int i,j,N;
i = N;
while (i>0) {
  if (j>0) j--;
  else
    {
      j = N;
      i--;
    }
}
```

ax (0.16s)
```
int i,j,n;
if (n>=1)
{
  i = 0;
  do {
    j = 0;
    while (j<n-1) j++;
    i++;
  }
  while (j>=n-1 && i<n-1);
}
```

loops (0.15s)
```
int n; /* n > 0 */
int x, y;
x = n;
if (x >= 0)
{
  while (x >= 0){
    y = 1;
    if (y < x)
      while (y < x)
        y = 2*y;
    x = x - 1;
  }
}
```

counterex1 (1.1s)
```
int n,x,y;
while(x>=0) {
  while(y>=0 &&
        brandom())
    y--;
  x--;
  while(y<=n &&
        brandom())
    y++;
}
```

determinant (0.15s)
```
//Automaton entered by hand
//from [33], p. 199
int i,j,k,n,y,z;
int X[10][10];
y = random();
//y = X[1][1];
k = 1;
B1:
if(k == n)
{
  z = y;
  return;
}
else
{
  i = k+1;
 B2:
  if(i == n+1)
    {
      k = k + 1;
      y = random();
      //y = y*X[k][k];
      goto B1;
    }
  else
    {
      j = n;
      while(j != k)
      {
//X[i][j] = X[i][j] -
//  X[k][j]*X[i][k]/X[k][k];
        j = j - 1;
      }
      i = i + 1;
      goto B2;
    }
}
```

maccarthy91 (1.2s)
```
int  y1,y2,z;
y1 = x;
y2 = 1;
if (y1>100) z = y1 - 10;
else
{
  while (y1 <= 100)
  {
    y1 = y1 + 11;
    y2 = y2 + 1;
  }
  while (y2 > 1)
  {
    y1 = y1 - 10;
    y2 = y2 - 1;
    if (y1 > 100 && y2 == 1)
      z = y1 - 10;
    else
    {
      if (y1 > 100)
      {
        y1 = y1 - 10;
        y2 = y2 - 1;
      }
      y1 = y1 + 11;
      y2 = y2 + 1;
    }
  }
}
```

realselect (0.4s)
```
int tab[];
int size,i,j,min,temp;
for (i=0; i<size-1; i++)
{
  min = i;
  for (j=i+1; j<size; j++)
  {
    if (tab[j] < tab[min])
      min = j;
  }
  temp = tab[i];
  tab[i] = tab[min];
  tab[min] = temp;
}
```

insertsort (0.22s)
```
int a[];
int len,i,j,value;
for (i=1; i<len; i++)
{
  value = a[i];
  for (j=i-1;
       j>=0 && a[j]>value;
       j--)
  {
    a[j + 1] = a[j];
  }
  a[j+1] = value;
}
```

sipmabubble (0.33s)
```
int A[];
int n,tmp,i,j;
i = n;
while(i >= 0) {
  j = 0;
  while(j <= i-1) {
    if(A[j] > A[j+1]) {
      tmp = A[j];
      A[j] = A[j+1];
      A[j+1] = tmp;
    }
    j++;
  }
  i--;
}
```

realbubble (0.4s)

```
int tab[];
int size,i,j,temp,test;
for(i=size-1; i>0; i--)
{
  test=0;
  for(j=0; j<i; j++)
  {
    if(tab[j] > tab[j+1])
    {
      temp = tab[j];
      tab[j] = tab[j+1];
      tab[j+1] = temp;
      test=1;
    }
  }
  if(test==0) break;
}
```

realshellsort (1.1s)

```
int tab[];
int i,j,inc,tmp;
inc = size / 2;
while (inc > 0)
{
  for (i=0; i<size; i++)
  {
    j = i;
    tmp = tab[i];
    while ((j >= inc) &&
           (tab[j-inc]>tmp))
    {
      tab[j] = tab[j-inc];
      j = j - inc;
    }
    tab[j] = tmp;
  }
  inc = inc/2;
}
```

realheapsort (2.8s)

```
int t[]
int temp,N,k,j,m;
if(N > 2){
  //construction
  for(k=1; k<=N-1; k++)
  {
    j=k;
    while((j>0) &&
          (t[(j+1)/2-1]>t[j]))
    {
      temp = t[j];
      t[j] = t[(j+1)/2-1];
      t[(j+1)/2-1] = temp;
      j = (j+1)/2-1;
    }
  }

  //destruction
  for(k=0; k<=N-2; k++)
  {
    j = 0; m = 0;
    temp = t[N-k-1];
    t[N-k-1] = t[0];
    t[0] = temp;
    while(2*j+1<=N-2-k)
    {
      if((2*j+1==N-2-k) ||
         (t[2*j+1]<t[2*j+2]))
        m = 2*j+1;
      else
        m = 2*j+2;
      if(t[j]>t[m])
      {
        temp = t[m];
        t[m] = t[j];
        t[j] = temp;
        j = m;
      }
      else j = N;
    }
  }
}
```

Contents