TLP **11** (4–5): 503–520, 2011. © Cambridge University Press 2011
doi:10.1017/S1471068411000147

SAT-based termination analysis using monotonicity constraints over the integers

MICHAEL CODISH and IGOR GONOPOLSKIY
Department of Computer Science, Ben-Gurion University, Israel
(*e-mail:* `mcodish@cs.bgu.ac.il`, `gonopols@cs.bgu.ac.il`)

AMIR M. BEN-AMRAM
School of Computer Science, Tel-Aviv Academic College, Israel
(*e-mail:* `amirben@mta.ac.il`)

CARSTEN FUHS and JÜRGEN GIESL
LuFG Informatik 2, RWTH Aachen University, Germany
(*e-mail:* `fuhs@informatik.rwth-aachen.de`, `giesl@informatik.rwth-aachen.de`)

Abstract

We describe an algorithm for proving termination of programs abstracted to systems of monotonicity constraints in the integer domain. Monotonicity constraints are a nontrivial extension of the well-known size-change termination method. While deciding termination for systems of monotonicity constraints is PSPACE complete, we focus on a well-defined and significant subset, which we call MCNP (for "monotonicity constraints in NP"), designed to be amenable to a SAT-based solution. Our technique is based on the search for a special type of ranking function defined in terms of bounded differences between multisets of integer values. We describe the application of our approach as the back end for the termination analysis of Java Bytecode. At the front end, systems of monotonicity constraints are obtained by abstracting information, using two different termination analyzers: AProVE and COSTA. Preliminary results reveal that our approach provides a good trade-off between precision and cost of analysis.

KEYWORDS: termination analysis, monotonicity constraints, SAT encoding

1 Introduction

Proving termination is a fundamental problem in verification. The challenge of termination analysis is to design a program abstraction that captures the properties needed to prove termination as often as possible, while providing a decidable sufficient criterion for termination. Typically, such abstractions represent a program as a finite set of abstract transition rules, which are descriptions of program steps, where the notion of step can be tuned to different needs. The abstraction considered in this paper is based on monotonicity-constraint systems (MCSs).

The MCS abstraction is an extension of the size-change termination (SCT) (Lee *et al.* 2001) abstraction, which has been studied extensively during the last decade (see `http://www2.mta.ac.il/~amirben/sct.html` for a summary and references). In the SCT abstraction, an abstract transition rule is specified by a set of inequalities that show how the sizes of program data in the target state are bounded by those in the source state. Size is measured by a well-founded base order. These inequalities are often represented by a *size-change graph.*

The size-change technique was conceived to deal with well-founded domains, where infinite descent is impossible. Termination is deduced by proving that any (hypothetical) infinite run would decrease some value monotonically and endlessly so that well-foundedness would be contradicted.

Extending this approach, a *monotonicity constraint* (MC) allows for any conjunction of order relations (strict and nonstrict inequalities) involving any pair of variables from the source and target states. So, in contrast to SCT, one may also have relations between two variables in the target state or two variables in the source state. Thus, MCSs are more expressive, and Codish *et al.* (2005) observe that earlier analyzers based on monotonicity constraints (Lindenstrauss and Sagiv 1997; Codish and Taboch 1999; Lindenstrauss *et al.* 2004) apply a termination test, which is sound and complete for SCT, but incomplete for monotonicity constraints, even if one does not change the underlying model, namely, that "data" are from an unspecified well-founded domain. They also point out that monotonicity constraints can imply termination under a different assumption—that the data are integers. Not being well founded, integer data cannot be handled by SCT. As an example, consider the Java program on the right, which computes the average of x and y. The loops in this program can be abstracted to the following monotonicity-constraint transition rules:

```
static int a(int x, int y){
  if (x>y){
    int x1=x-1; int y1=y+1;
    if (x1>=y1)
      return a(x1,y1);
    else return y;
  } else {
    int x1=x+1; int y1=y-1;
    if (x1<=y1)
      return a(x1,y1);
    else return x;
  }
}
```

$$\begin{array}{lll} (1) & a(x,y) \;\; :- & x > y, x > x', y' > y, x' \geqslant y'; \;\; a(x',y'), \\ (2) & a(x,y) \;\; :- & y \geqslant x, x' > x, y > y', y' \geqslant x'; \;\; a(x',y'). \end{array}$$

To prove termination of the Java program, it is sufficient to focus on the corresponding abstraction. Note that termination of this program cannot be proved using SCT, not only because SCT disallows constraints between source variables (such as $x{>}y$), but also because it computes with integers rather than natural numbers.

To see how the transition constraints imply termination, observe that if rule (1) is repeatedly taken, then the value of y grows; constraint $x > y$ (with the fact that x descends) implies that this cannot go on forever. In rule (2), the situation is reversed: y descends and is lower bounded by x. In addition, constraint $y' \geqslant x'$ of rule (2) implies that, once this rule is taken, there can be no more applications of rule (1). Therefore, any (hypothetical) infinite computation would eventually enter a loop of rule (1)s or a loop of rule (2)s; possibilities that we have just ruled out. In this paper, we show how to obtain such termination proofs automatically using SAT solving.

Although MCS and SCT are abstractions where termination is decidable, they have a drawback: the decision problems are PSPACE complete and a certificate for termination under these abstractions can be of prohibitive complexity (not "polynomially computable" (Ben-Amram 2009)). Typical implementations based on the SCT abstraction apply a closure operation on transition rules, which is exponential both in time and space. Ben-Amram and Codish (2008) addressed this problem for SCT, identifying an NP complete subclass of SCT, called SCNP, which yields polynomial-size certificates. Moreover, Ben-Amram and Codish (2008) automated SCNP using a SAT solver. Experiments indicated that, in practice, this method had good performance and power when compared to a complete SCT decision procedure and had the additional merit of producing certificates.

In this paper, we tackle the similar problem to prove termination of monotonicity-constraint systems in the integer domain. As noted above, the integer setting is more complicated than the well-founded setting. Termination is often proved by looking at *differences* of certain program values (which should be decreasing and lower-bounded). One could simulate such reasoning in SCT by creating fresh variables to track the nonnegative differences of pairs of original variables. However, this loses precision and may square the number of variables, which is an exponent in the complexity of most SCT algorithms. Instead, we use an idea from Ben-Amram and Codish (2008), which consists of mapping program states into multisets of argument values. The adaption of this method to integer data is nontrivial. Our new solution uses the following ideas: (1) We associate two sets with each program point and define how to "subtract" them so that the difference can be used for ranking (generalizing the difference of two integers). This avoids the quadratic growth in the exponent of the complexity, since we are only working with the original variables and relations, and is also more expressive. (2) We introduce a concept of "ranking functions," which is less strict than typically used but still suffices for termination. It allows the codomain of the function to be a non-well-founded set that has a well-founded subset. This gives an additional edge over the naïve reduction to SCT, which can only make use of differences which are definitely nonnegative.

After presenting preliminaries in Section 2, Section 3 introduces *ranking structures*, which are termination witnesses. In Section 4, we show that such a witness can be verified in polynomial time; hence, the resulting subclass of terminating MCSs lies in NP. Consequently, we call it MCNP. In Section 5, we devise an algorithm that uses a SAT solver as a back end to solve the resulting search problems. Section 6 describes an empirical evaluation using a prototypical implementation as the back end for termination analysis of Java Bytecode (JBC). Results indicate a good trade-off between precision and cost of analysis. All proofs and further details of the evaluation can be found in the appendices.

Related work. Termination analysis is a vast field and we focus here on the most closely related work. On termination analyzers for JBC, we mention COSTA (Albert *et al.* 2008), Julia (Spoto *et al.* 2010), and AProVE (Brockschmidt *et al.* 2010; Otto *et al.* 2010). Both COSTA and Julia abstract programs into a CLP form, as in this work; but use a richer constraint language that makes termination of the abstract program undecidable. On extending SCT to the integer domain: Avery (2006) uses constraints of the form $x{>}y', x{\geqslant}y', x{<}y', x{\leqslant}y'$ along with polyhedral state invariants (similar constraints as those used by COSTA and Julia) to find lower bounded combinations of the variables. Manolios and Vroon (2006) use SCT constraints on pseudovariables that represent "measures" invented by the system. This allows it to handle integers by taking, for example, the differences of two variables as a measure. Dershowitz *et al.* (2001) and Serebrenik and De Schreye (2004) prove termination of logic programs that depend on numerical constraints by inferring "level mappings" based on constraints selected from the source program; so, a constraint, like $x > y$, can trigger the use of $x - y$ as a level mapping. There are numerous applications of SAT for deciding termination problems for all kinds of programs (e.g., one of the first such papers is Codish *et al.* (2006)).

2 Monotonicity-constraint systems and their termination

Our method is programming-language independent. It works on an abstraction of the program provided by a front end. An abstract program is a transition system with states expressed in terms of a finite number of variables (*argument positions*).

Definition 1 (*constraint transition system*)
A *constraint transition system* is an abstract program, represented by a directed multigraph called a *control-flow graph* (CFG). The vertices are called *program points* and they are associated with fixed numbers (arity) of *argument positions*. We write p/n to specify the arity of vertex p. A *program state* is an association of a value from the *value domain* to each argument position of a program point p, denoted by $p(x_1, \ldots, x_n)$ and abbreviated as $p(\bar{x})$. The set of all states is denoted by St. The arcs of the CFG are associated with *transition rules*, specifying relations on program states, which we write as $p(\bar{x}) \coloneq \pi;\, q(\bar{y})$. The *transition predicate* π is a formula in the *constraint language* of the abstraction.

Note that a state corresponds to a ground atom: argument positions are associated with specific values. In a transition rule, positions are associated with variables that can only be constrained through π. Thus, in the notation $p(\bar{x})$, $\bar{x}$ may represent ground values or variables, according to context. The constraint language in our work is that of *monotonicity constraints.*

Definition 2 (*monotonicity constraint*)
A *monotonicity constraint* (MC) π on $V = \bar{x} \cup \bar{y}$ is a conjunction of constraints $x \rhd y$ where $x, y \in V$, and $\rhd \in \{>, \geqslant\}$. We write $\pi \models x \rhd y$, whenever $x \rhd y$ is a consequence of π (in the theory of total orders). This consequence relation is easily computed, e.g., by a graph algorithm. A transition rule $p(\bar{x}) \coloneq \pi;\, q(\bar{y})$, where π is a MC, is also known as a *monotonicity-constraint transition rule.* An *integer monotonicity-constraint transition system* (MCS)[1] is a constraint transition system where the value domain is $\mathbb{Z}$ and transition predicates are monotonicity constraints.

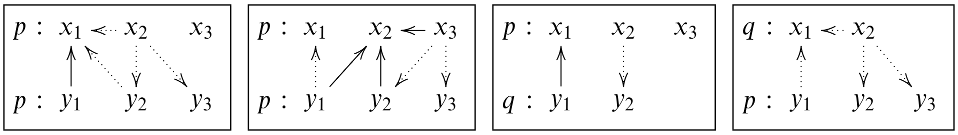
[1] In this work, only the integer domain is of interest; hence, "integer" will be omitted. Moreover, instead of "monotonicity-constraint transition systems" we also speak of "monotonicity-constraint systems".

It is useful to represent a MC as a directed graph (often denoted by the letter g), with vertices $\bar{x} \cup \bar{y}$, and two types of edges (x, y): weak and strict. If $\pi \models x > y$, then there is a strict edge from x to y, and if $\pi \models x \geqslant y$ (but not $x > y$), then the edge is weak. Note that there are two kinds of graphs, those representing transition rules and the CFG. We often identify an abstract program with its set $\mathcal{G}$ of transition rules, the CFG being implicitly specified.

Definition 3 (*run, termination*)
Let $\mathcal{G}$ be a transition system. A *run* of $\mathcal{G}$ is a sequence $p_0(\bar{x}_0) \overset{\pi_0}{\to} p_1(\bar{x}_1) \overset{\pi_1}{\to} p_2(\bar{x}_2) \ldots$ of states labeled by constraints such that each labeled pair of states, $p_i(\bar{x}_i) \overset{\pi_i}{\to} p_{i+1}(\bar{x}_{i+1})$, corresponds to a transition rule $p_i(\bar{x}) \coloneq \pi_i;\ p_{i+1}(\bar{y})$ from $\mathcal{G}$ (identical except that variables $\bar{x}$ and $\bar{y}$ are replaced by values $\bar{x}_i$ and $\bar{x}_{i+1}$) and such that π_i is satisfied. A transition system *terminates* if it has no infinite run.

Example 4
This example presents a MCS in textual form as well as graphical form. This system is terminating, and in the following sections, we shall illustrate how our method proves it. In the graphs, solid arrows stand for strict inequalities and dotted arrows stand for weak inequalities.

$$\begin{array}{lllll}
g_1 = & p(x_1,x_2,x_3) & \coloneq & y_1 > x_1, y_2 \geqslant x_1, x_2 \geqslant y_2, x_2 \geqslant y_3, x_2 \geqslant x_1; & p(y_1,y_2,y_3) \\
g_2 = & p(x_1,x_2,x_3) & \coloneq & y_1 \geqslant x_1, y_1 > x_2, y_2 > x_2, x_3 \geqslant y_2, x_3 \geqslant y_3, x_3 > x_2; & p(y_1,y_2,y_3) \\
g_3 = & p(x_1,x_2,x_3) & \coloneq & y_1 > x_1,\ x_2 \geqslant y_2; & q(y_1,y_2) \\
g_4 = & q(x_1,x_2) & \coloneq & y_1 \geqslant x_1,\ x_2 \geqslant y_2, x_2 \geqslant y_3,\ x_2 \geqslant x_1; & p(y_1,y_2,y_3)
\end{array}$$



3 Ranking structures for monotonicity-constraint systems

This section describes *ranking structures*, a concept that we introduce for proving termination of MCSs. Section 3.1 presents the necessary notions in general form. Then, Section 3.2 specializes them to the form we use for MCNP.

3.1 Ranking structures

Recall that $\gtrsim$ is a *quasi-order* if it is transitive and reflexive; its *strict part* $x \succ y$ is the relation $(x \gtrsim y) \wedge (y \not\gtrsim x)$; the quasi-order is *well founded* if there is no infinite chain with $\succ$. A set is well founded if it has a tacitly understood well-founded order.

A *ranking function* maps program states into a well-founded set such that every transition decreases the function's value. As shown in Ben-Amram (2011), for every terminating MCS, there exists a corresponding ranking function. However, these are of exponential size in the worst case. Since our aim is NP complexity, we cannot use that construction, but instead restrict ourselves to polynomially sized termination witnesses. These witnesses, called *ranking structures*, are more flexible than ranking functions and suffice for most practical termination proofs.

Definition 5 (*anchor, intermittent ranking function*)
Let $\mathscr{G}$ be a MCS with state space St. Let $(\mathscr{D}, \gtrsim)$ be a quasi-order and $\mathscr{D}_+$ a well-founded subset of $\mathscr{D}$. Consider a function $\Phi : St \to \mathscr{D}$. We say that $g \in \mathscr{G}$ is a Φ-*anchor* for $\mathscr{G}$ (or that g is *anchored* by Φ for $\mathscr{G}$) if for every run $p_0(\bar{x}_0) \xrightarrow{\pi_0} p_1(\bar{x}_1) \xrightarrow{\pi_1} \dots \xrightarrow{\pi_{k-1}} p_k(\bar{x}_k) \xrightarrow{\pi_k} p_{k+1}(\bar{x}_{k+1})$, where both $p_0(\bar{x}_0) \xrightarrow{\pi_0} p_1(\bar{x}_1)$ and $p_k(\bar{x}_k) \xrightarrow{\pi_k} p_{k+1}(\bar{x}_{k+1})$ correspond to the transition rule g, we have $\Phi(p_i(\bar{x}_i)) \gtrsim \Phi(p_{i+1}(\bar{x}_{i+1}))$ for all $0 \leqslant i \leqslant k$, where at least one of these inequalities is strict; and $\Phi(p_i(\bar{x}_i)) \in \mathscr{D}_+$ for some $0 \leqslant i \leqslant k$. A function Φ that satisfies the above conditions is called an *intermittent ranking function* (IRF).[2]

Example 6
Consider the transition rules from Example 4. Let $\mathscr{G} = \{g_1, g_2\}$ and let $\Phi_1(p(\bar{x})) = max(x_2, x_3) - x_1$. In any run built with g_1 and g_2, the value of Φ_1 is nonnegative at least in every state followed by a transition by g_1. Moreover, a transition by g_1 decreases the value strictly and a transition by g_2 decreases it weakly. Hence, g_1 is anchored by Φ_1 for $\mathscr{G}$ (in Section 3.2, we come back to this example and show how Φ_1 fits the patterns of termination proofs that our method is designed to discover).

Definition 7 (*ranking structure*)
Consider $\mathscr{G}$ and $\mathscr{D}$ as in Definition 5. Let $\Phi_1, \dots, \Phi_m : St \to \mathscr{D}$. Let $\mathscr{G}_1$ consist of all transition rules $g \in \mathscr{G}$ where Φ_1 anchors g for $\mathscr{G}$. For $2 \leqslant i \leqslant m$, let $\mathscr{G}_i$ consist of all transition rules $g \in \mathscr{G} \setminus (\mathscr{G}_1 \cup \dots \cup \mathscr{G}_{i-1})$ where Φ_i anchors g in $\mathscr{G} \setminus (\mathscr{G}_1 \cup \dots \cup \mathscr{G}_{i-1})$. We say that $\langle \Phi_1, \dots, \Phi_m \rangle$ is a *ranking structure* for $\mathscr{G}$ if $\mathscr{G}_1 \cup \dots \cup \mathscr{G}_m = \mathscr{G}$.

Note that by the above definition, for every $g \in \mathscr{G}$, there is a (unique) $\mathscr{G}_i$ with $g \in \mathscr{G}_i$. We denote this index i as $i(g)$ (i.e., $g \in \mathscr{G}_{i(g)}$ for all $g \in \mathscr{G}$).

Example 8
For the program $\{g_1, g_2\}$ of Example 4, a ranking structure is $\langle \Phi_1, \Phi_2 \rangle$ with Φ_1 as in Example 6 and $\Phi_2(p(\bar{x})) = x_3 - x_2$. Here, we have $i(g_1) = 1$ and $i(g_2) = 2$. Later, in Examples 18 and 27, we will extend the ranking structure to the whole program $\{g_1, g_2, g_3, g_4\}$.

The concept of ranking structures generalizes that of lexicographic global ranking functions used, e.g., in Ben-Amram and Codish (2008) and Alias *et al.* (2010). A lexicographic ranking function is a ranking structure, however, the converse is not always true, since the function Φ does not necessarily decrease on a transition rule, which it anchors, and because Φ may assume values out of $\mathscr{D}_+$ in certain states.

Theorem 9
If there is a ranking structure for $\mathscr{G}$, then $\mathscr{G}$ terminates.

Definition 10
A ranking structure $\langle \Phi_1, \Phi_2, \dots, \Phi_m \rangle$ for $\mathscr{G}$ is *irredundant* if for all $j \leqslant m$, there is a transition $g \in \mathscr{G}$ such that $i(g) = j$.

It follows easily from the definitions that if there is a ranking structure for $\mathscr{G}$, there is an irredundant one, of length at most $|\mathscr{G}|$.

[2] The term "intermittent ranking function" is inspired by Manna and Waldinger (1978).

3.2 Multiset orders and level mappings

The building blocks for our construction are four quasi-orders on multisets of integers, and a notion of *level mappings*, which map program states into pairs of multisets, whose *difference* (not set-theoretic difference; see Definition 15 below) will be used to rank the states.[3] The difference will be itself a multiset, and we now elaborate on the relations that we use to order such multisets.

Definition 11 (*multiset types*)
Let $\wp_n(\mathbb{Z})$ denote the set of multisets of integers of at most n elements, where n is fixed by context.[4] The μ-ordered multiset type, for $\mu \in \{\, max, min, ms, dms \,\}$, is the quasi-ordered set $(\wp_n(\mathbb{Z}), \succsim^{\mu})$ where:

(1) *(max order)* $S \succsim^{max} T$ holds iff $max(S) \geqslant max(T)$, or T is empty; $S \succ^{max} T$ holds iff $max(S) > max(T)$, or T is empty while S is not.
(2) *(min order)* $S \succsim^{min} T$ holds iff $min(S) \geqslant min(T)$, or S is empty; $S \succ^{min} T$ holds iff $min(S) > min(T)$, or S is empty while T is not.
(3) *(multiset order (Dershowitz and Manna 1979))* $S \succ^{ms} T$ holds iff T is obtained by replacing a nonempty $U \subseteq S$ by a (possibly empty) multiset V such that $U \succ^{max} V$; the weak relation $S \succsim^{ms} T$ holds iff $S \succ^{ms} T$ or $S = T$.
(4) *(dual multiset order (Ben-Amram and Lee 2007))* $S \succ^{dms} T$ holds iff T is obtained by replacing a submultiset $U \subseteq S$ by a nonempty multiset V with $U \succ^{min} V$; the weak relation $S \succsim^{dms} T$ holds iff $S \succ^{dms} T$ or $S = T$.

Example 12
For $S = \{10, 8, 5\}$, $T = \{9, 5\}$: $S \succ^{max} T$, $T \succsim^{min} S$, $S \succ^{ms} T$, and $T \succ^{dms} S$.

Definition 13 (*well-founded subset of multiset types*)
For $\mu \in \{\, max, min, ms, dms \,\}$, we define $(\wp_n(\mathbb{Z}), \succsim^{\mu})_+$ as follows: for *min* (respectively *max*) order, the subset consists of the multisets whose minimum (respectively, maximum) is nonnegative. For *ms* and *dms* orders, the subset consists of the multisets all of whose elements are nonnegative.

Lemma 14
For all $\mu \in \{max, min, ms, dms\}$, $(\wp_n(\mathbb{Z}), \succsim^{\mu})$ is a total quasi-order, with $\succ^{\mu}$ its strict part; and $(\wp_n(\mathbb{Z}), \succsim^{\mu})_+$ is well founded.

For MCs over the integers, it is necessary to consider differences: in the simplest case, we have a "low variable" x that is nondescending and a "high variable" y that is nonascending, so $y - x$ is nonascending (and will decrease if x or y changes). If we also have a constraint like $y \geqslant x$, to bound the difference from below, we can use this for ranking a loop (we refer to this situation as "the Π"—due to the diagram on the right). In the more general case, we consider sets of variables. We will search for a similar Π situation involving a "low set" and a "high set." We next define how to form a difference of two sets so that one can follow the same strategy of "diminishing difference."

$$\begin{tikzcd} x & y \arrow[l, dotted] \arrow[d, dotted] \\ x' \arrow[u, dotted] & y' \end{tikzcd}$$

[3] A reader familiar with previous works using this term should note that here, a level mapping is not in itself some kind of ranking function.
[4] For MCSs, n is the maximum arity of program points.

Definition 15 (*multiset difference*)
Let L, H be nonempty multisets with types μ_L, μ_H respectively. Their difference $H - L$ is defined in the following way, depending on the types (there are six cases):

(1) For $\mu_L \in \{max, min\}$, $H - L = \{h - \mu_L(L) \mid h \in H\}$ and has the type of H. (Here, $\mu_L(L)$ signifies $min(L)$ or $max(L)$ depending on the value of μ_L).
(2) For $\mu_L \in \{ms, dms\}$ and $\mu_H \in \{min, max\}$, $H - L = \{\mu_H(H) - \ell \mid \ell \in L\}$ and has type $\overline{\mu}_L$ (where $\overline{ms} = dms$ and $\overline{dms} = ms$).

For L and H such that $H - L$ is defined, we say that the types of L and H are *compatible*. We write $H \mathrel{\underline{\ni}} L$ if the difference belongs to the well-founded subset.

Note that $\mathrel{\underline{\ni}}$ relates multisets of possibly different types and is not an order relation. Termination proofs do not require to define the difference of multisets with types in $\{ms, dms\}$. To see why, observe that in "the Π," only one multiset must change strictly, and the nonstrict relations $\gtrsim^{ms}$, $\gtrsim^{dms}$ are contained in $\gtrsim^{max}$, $\gtrsim^{min}$, respectively. Note also that $H \mathrel{\underline{\ni}} L$ is equivalent, in all relevant cases, to $\mu_1(H) \geqslant \mu_2(L)$ with $\mu_1, \mu_2 \in \{min, max\}$. The intuition into why multiset difference is defined as above is rooted in the following lemma.

Lemma 16
Let L, H be two multisets of compatible types μ_L, μ_H, and let μ_D be the type of $H - L$. Let L', H' be of the same types as L, H respectively. Then,

$$\begin{aligned}
H \gtrsim^{\mu_H} H' \wedge L \lesssim^{\mu_L} L' &\implies H - L \gtrsim^{\mu_D} H' - L';\\
H \succ^{\mu_H} H' \wedge L \lesssim^{\mu_L} L' &\implies H - L \succ^{\mu_D} H' - L';\\
H \gtrsim^{\mu_H} H' \wedge L \prec^{\mu_L} L' &\implies H - L \succ^{\mu_D} H' - L'.
\end{aligned}$$

Level mappings are functions that facilitate the construction of ranking structures. Three types of level mappings are defined in Ben-Amram and Codish (2008): *numeric*, *plain*, and *tagged*. In this paper, we focus on "plain" and "tagged" level mappings and we adapt them for multisets of integers. Numeric level mappings have become redundant in this paper due to the passage from ranking functions to ranking structures. We first introduce the extension for plain level mappings.

Definition 17 (*bimultiset level mapping, or "level mapping" for short*)
Let $\mathcal{G}$ be a MCS. A *bimultiset level mapping*, f_{μ_L,μ_H} maps each program state $p(\bar{x})$ to a pair of (possibly intersecting) multisets $p_f^{low}(\bar{x}) = \{\, u_1, \ldots, u_l \,\} \subseteq \bar{x}$ and $p_f^{high}(\bar{x}) = \{\, v_1, \ldots, v_k \,\} \subseteq \bar{x}$ with types indicated, respectively, by $\mu_L, \mu_H \in \{\, max, min, ms, dms \,\}$. Only compatible pairs μ_L, μ_H are admitted. The selection of argument positions only depends on the program point p.

Example 18
The following are the level mappings used (in Example 27) to prove termination of the program of Example 4. Here, each program point p is mapped to $\langle p_f^{low}(\bar{x}), p_f^{high}(\bar{x})\rangle$.

$f^1_{min,max}(p(\bar{x})) = \langle\{\, x_1 \,\}, \{\, x_2, x_3 \,\}\rangle$ $\quad$ $f^2_{min,max}(p(\bar{x})) = \langle\{\, x_2 \,\}, \{\, x_3 \,\}\rangle$
$f^1_{min,max}(q(\bar{x})) = \langle\{\, x_1 \,\}, \{\, x_2 \,\}\rangle$ $\quad$ $f^2_{min,max}(q(\bar{x})) = \langle\{\ \}, \{\ \}\rangle$

We now turn to tagged level mappings. Assume the context of Definition 17 and let M denote the sum of the arities of all program points. A *tagged* bimultiset level mapping is just like a bimultiset level mapping, except that set elements are pairs of the form (x, t), where x is from $\bar{x}$ and $t < M$ is a natural constant, called a tag. We view such a pair as representing the integer value $Mx + t$ (recall that x is an integer). This transforms tagged multisets into multisets of integers, so Definitions 15 and 17, and the consequent definitions and results can be used without change.

Tags "prioritize" certain argument positions and can usefully turn weak inequalities into strict ones. For example, consider a transition rule $p(\bar{x})$:- $x_1 > y_1, x_1 \geqslant y_2, \dots ; p(\bar{y})$. The tagged set $\{(x_1, 1), (x_2, 0)\}$ is strictly greater (in *ms* order as well as in *max* order) than $\{(y_1, 1), (y_2, 0)\}$ (because $\pi \models (x_1, 1) > (y_2, 0)$). The plain sets $\{x_1, x_2\}$ and $\{y_1, y_2\}$ do not satisfy these relations. Thus, tagging may increase the chance of finding a termination proof. We do not have any fixed rule for tagging; our SAT-based procedure will find a useful tagging if one exists. In the remainder, we write "level mapping" to indicate a, possibly tagged, bimultiset level mapping.

Level mappings are applied in termination proofs to express the diminishing difference of their low and high sets. To be useful, we also need to express a constraint relating the high and low sets, providing, figuratively, the horizontal bar of "the Π." A transition rule that has such a constraint is called *bounded.*

Definition 19 (*bounded*)
Let $\mathcal{G}$ be a MCS, f be a level mapping,[5] and $g \in \mathcal{G}$. A transition rule $g = p(\bar{x})$:- $\pi; q(\bar{y})$ in $\mathcal{G}$ is called *bounded w.r.t. f* if $\pi \models p_f^{high} \trianglerighteq p_f^{low}$.

Definition 20 (*orienting transition rules*)
Let f be a level mapping. (1) f *orients* transition rule $g = p(\bar{x})$:- $\pi; q(\bar{y})$ if $\pi \models p_f^{high}(\bar{x}) \succsim q_f^{high}(\bar{y})$ and $\pi \models p_f^{low}(\bar{x}) \precsim q_f^{low}(\bar{y})$; (2) f orients g *strictly* if, in addition, $\pi \models p_f^{high}(\bar{x}) \succ q_f^{high}(\bar{y})$ or $\pi \models p_f^{low}(\bar{x}) \prec q_f^{low}(\bar{y})$.

Example 21
We refer to Example 4 and the level mapping $f^1_{min,max}$ from Example 18. Function $f^1_{min,max}$ orients all transition rules, where g_1 and g_3 are oriented strictly; g_1 and g_4 are bounded w.r.t. $f^1_{min,max}$ (the reader may be able to verify this by observing the constraints; however, later we explain how our algorithm obtains this information).

Corollary 22 (*of Definition 20 and Lemma 1*)
Let f be a level mapping and define $\Phi_f(p(\bar{x})) = p_f^{high}(\bar{x}) - p_f^{low}(\bar{x})$. If f orients $g = p(\bar{x})$:- $\pi; q(\bar{y})$, then $\pi \models \Phi_f(p(\bar{x})) \gtrsim \Phi_f(q(\bar{y}))$; and if f orients g strictly, then $\pi \models \Phi_f(p(\bar{x})) > \Phi_f(q(\bar{y}))$.

The next theorem combines orientation and bounding to show how a level mapping induces anchors. Note that we refer to cycles in the CFG also as "cycles in $\mathcal{G}$," as the CFG is implicit in $\mathcal{G}$.

[5] We sometimes write f (for short) instead of f_{μ_L,μ_H}.

Theorem 23
Let $\mathcal{G}$ be a MCS and f be a level mapping. Let $g = p(\bar{x}) \coloneq \pi; q(\bar{y})$ be such that every cycle $\mathcal{C}$, including g, satisfies these conditions: (1) all transitions in $\mathcal{C}$ are oriented by f, and at least one of them strictly; (2) at least, one transition in $\mathcal{C}$ is bounded w.r.t. f. Then, g is a Φ_f-anchor for $\mathcal{G}$, where $\Phi_f(p(\bar{x})) = p_f^{high}(\bar{x}) - p_f^{low}(\bar{x})$.

Definition 24 (*MCNP anchors and ranking functions*)
Let $\mathcal{G}$ be a MCS and f be a level mapping. We say that g is a MCNP-anchor for $\mathcal{G}$ w.r.t. f if f and g satisfy the conditions of Theorem 1. The function Φ_f is called a *MCNP (intermittent) ranking function* (MCNP IRF).

Note that if g is not included in any cycle, then the definition is trivially satisfied for any f. Indeed, such transition rules are removed by our algorithm without searching for level mappings at all.

Example 25
The facts in Example 21 imply that g_1, g_3, and g_4 are MCNP-anchors w.r.t. $f^1_{min,max}$.

We remark that numerous termination proving techniques follow the pattern of, repeatedly, identifying and removing anchors. However, typically, the function Φ used for ranking is required to be strictly decreasing, and bounded, on the anchor itself, which (at least implicitly) means that a lexicographic ranking function is being constructed (see, e.g., Colón and Sipma (2002)). The anchor criterion expressed in Theorem 1 (inspired by Giesl *et al.* (2007, Theorem 8)) is more powerful. We note that the difference is only important with non-well-founded domains. When the ranking is only done with orders that are a priori well founded, as, for example, in Giesl *et al.* (2006) and Hirokawa and Middeldorp (2005), considering the strictly oriented transitions as anchors is sufficient. In comparison to Giesl *et al.* (2007), we note that they do not use the concept of anchors and propose an algorithm, which can generate an exponential number of level-mapping-finding subproblems (whereas ours generates, in the worst case, as many problems as there are transition rules).

4 The MCNP problem

In this section, we present necessary and sufficient conditions for orientability and boundedness. On the basis of these, we conclude that proving termination with MCNP IRFs is in NP. This also forms the basis for our SAT-based algorithm in Section 5.

Definition 26 (*MCNP*)
A system of monotonicity constraints is in MCNP if it has a ranking structure which is a tuple of MCNP IRFs.

It follows from Theorem 1, that if a MCS is in MCNP, then it terminates.

Example 27
Consider again Example 4 and the level mappings from Example 18. Then, $\langle \Phi_{f^1}, \Phi_{f^2} \rangle$ is a ranking structure for $\mathcal{G}$. As already observed, g_1, g_3, and g_4 are MCNP-anchors for f^1. Observe now that f^2 is both strict and bounded on g_2.

Ranking structures are constructed through iterative search for suitable level mappings that prescribe pairs of (possibly tagged) multisets of arguments, which must satisfy relations of the form $\succsim^{\mu}$, $\succ^{\mu}$, and $\trianglerighteq$.

Let $g = p(\bar{x}) :- \pi; q(\bar{y})$ and S, T be nonempty sets of (tagged) argument positions of p or of q. We show how to check for each $\mu \in \{\, max, min, ms, dms \,\}$ if $\pi \models S \succsim^{\mu} T$. Viewing g as a graph (as in Example 4), let g^t denote the transpose of g (obtained by inverting the arcs). While tagged level mappings can be represented as "ordinary" bimultiset level mappings (as indicated in Section 3.2), for their SAT encoding, it is advantageous to represent the orders on tagged pairs explicitly:

$$\begin{array}{lcl} \pi \models (x,i) > (y,j) & \iff & (\pi \models x > y) \vee ((\pi \models x \geqslant y) \wedge i > j), \\ \pi \models (x,i) \geqslant (y,j) & \iff & (\pi \models x > y) \vee ((\pi \models x \geqslant y) \wedge i \geqslant j). \end{array} \tag{1}$$

Below, x, y either both represent arguments, or both represent tagged arguments, with relations $x > y$, $x \geqslant y$ interpreted accordingly.

(1) *max order: ($S \succsim^{max} T$)* every $y \in T$ must be "covered" by an $x \in S$ such that $\pi \models x \geqslant y$. Strict descent requires $S \neq \emptyset$ and $x > y$.
(2) *min order: ($S \succsim^{min} T$)* same conditions but on g^t (now T covers S).
(3) *multiset order: ($S \succsim^{ms} T$)* every $y \in T$ must be "covered" by an $x \in S$ such that $\pi \models x \geqslant y$. Furthermore, each $x \in S$ either covers each related y strictly ($x > y$) or covers at most a single y. Descent is strict if there is some x that participates in strict relations.
(4) *dual multiset order: ($S \succsim^{dms} T$)* same conditions but on g^t (now T covers S).

We also show how to decide if the relation $H \trianglerighteq L$ holds: for $\mu_L, \mu_H \in \{max, min\}$ and $\mu_L = \mu_H$, $H \trianglerighteq L$ holds iff $\mu_H(H) \geqslant \mu_L(L)$.[6] For $\mu_L = min$ and $\mu_H \in \{ms, dms\}$, $H \trianglerighteq L$ holds iff $H \succsim^{min} L$. For $\mu_L \in \{ms, dms\}$ and $\mu_H = max$, $H \trianglerighteq L$ holds iff $H \succsim^{max} L$. For $\mu_L = max$ and $\mu_H \in \{ms, dms\}$, $H \trianglerighteq L$ holds if $min(H) \geqslant max(L)$. For $\mu_L \in \{ms, dms\}$ and $\mu_H = min$, $H \trianglerighteq L$ holds if $min(H) \geqslant max(L)$.

Since the above conditions allow for verification of a proposed MCNP ranking structure in polynomial time, we obtain the following theorem.

Theorem 28
MCNP is in NP.

5 A SAT-based MCNP algorithm

Given that MCNP is in NP, we provide a reduction (an encoding) to SAT, which enables us to find termination proofs using an off-the-shelf SAT solver. We invoke a SAT solver iteratively to generate level mappings and construct a ranking structure $\langle \Phi_1, \Phi_2, \ldots, \Phi_m \rangle$. Our main algorithm is presented in Section 5.1. Section 5.2 discusses how to find appropriate level mappings and Section 5.3 introduces the SAT encoding.

[6] Note that checking this amounts to checking for $\succsim^{\mu}$ in the case $\mu_L = \mu_H = \mu$; for the other cases, $max(H) \geqslant min(L)$ holds if there is at least one arc from an H vertex to an L vertex; $min(H) \geqslant max(L)$ holds if there is an arc from every H vertex to every L vertex.

5.1 Main algorithm

Given a MCS $\mathscr{G}$, the idea is to iterate as follows: while $\mathscr{G}$ is not empty, find a level mapping f inducing one or more anchors for $\mathscr{G}$. Remove the anchors, and repeat. The instruction "find a level mapping" is performed using a SAT encoding (for each of the compatible pairs of multiset orders). To improve performance, the algorithm follows the strongly connected components (SCCs) decomposition of (the CFG of) $\mathscr{G}$. This leads to smaller subproblems for the SAT solver and is justified by the observation that intercomponent transitions are trivially anchors (not included in any cycle). In the following, let $scc(\mathscr{G})$ denote the set of nonvacant SCCs of $\mathscr{G}$ (that is, SCCs, which are not a vertex without any arcs).

Main Algorithm.

```
input: $\mathscr{G}$ (a MCS)
output: $\rho = \langle f^1, f^2, \ldots \rangle$ (tuple of level mappings such that $\langle \Phi_{f^1}, \Phi_{f^2}, \ldots \rangle$
        is a ranking structure for $\mathscr{G}$). The algorithm aborts if $\mathscr{G}$ is not in MCNP.
(1) $\rho = \langle\ \rangle$ (empty queue);   $\mathscr{S} = scc(\mathscr{G})$ (stack with nonvacant SCCs of $\mathscr{G}$);
(2) while ($\mathscr{S} \neq \emptyset$)
      • pop $\mathscr{C}$ from $\mathscr{S}$ (a MCS) and find (using SAT) a level mapping
        $f$ to anchor some transition rules in $\mathscr{C}$   (if none, abort: $\mathscr{C} \notin$ MCNP),
      • extend $f$ to program points $p$ not in $\mathscr{C}$ by $f(p(\bar{x})) = \langle \emptyset, \emptyset \rangle$,
      • append $f$ to $\rho$ and remove from $\mathscr{C}$ the $\Phi_f$-anchors that were found,
      • push elements of $scc(\mathscr{C})$ to $\mathscr{S}$;
(3) return $\rho$
```

Theorem 29
The main algorithm succeeds if and only if $\mathscr{G}$ is in MCNP.

5.2 Finding a level mapping

The main step in the algorithm is to find a level mapping, which anchors some transition rules of a strongly connected MCS. Let $\mathscr{G}$ be strongly connected and f be a level mapping that orients all transition rules in $\mathscr{G}$, strictly orients the transition rules from a nonempty set $S \subseteq \mathscr{G}$, and where $B \subseteq \mathscr{G}$ (nonempty) are bounded. Following Theorem 1, a transition rule g is an anchor if every cycle in $\mathscr{G}$ containing g has an element from S and an element from B. We need to check all cycles in $\mathscr{G}$ (possibly exponentially many). We describe a way of doing so by numbering nodes, which lends itself well to a SAT-based solution.

Definition 30 (*node numbering*)
A *node numbering* is a function *num* from n program points to $\{1, \ldots, n\}$. For $g = p(\bar{x}) \coloneq \pi; q(\bar{y})$, we denote $\Delta num(g) = num(q) - num(p)$. For a set $\mathscr{H} \subseteq \mathscr{G}$, we say that *num agrees with* $\mathscr{H}$ if for all $g \in \mathscr{G}$: $\Delta num(g) > 0 \Rightarrow g \in \mathscr{H}$.

Now, for $g \in \mathscr{G}$, checking that every cycle of $\mathscr{G}$ containing g also contains an element of S, is reduced to finding a node numbering num_S with $\Delta num_S(g) \neq 0$ that agrees with S. Then, any cycle containing g must contain also an edge g' with $\Delta num_S(g') > 0$. But this implies that $g' \in S$ because num_S agrees with S.

Lemma 31

Let $\mathcal{G}$, f, S, and B be as above. Then, $g \in \mathcal{G}$ is a MCNP-anchor for $\mathcal{G}$ w.r.t f if and only if: (1) $g \in S \cap B$; or (2) there are node numberings num_S and num_B agreeing with S and B, respectively, such that $\Delta num_S(g) \neq 0$ and $\Delta num_B(g) \neq 0$.

Example 32

We now describe the application of the Main Algorithm to Example 4. Initially, there is a single SCC, $\mathcal{C} = \mathcal{G}$. Using SAT solving (as described in Section 5.3), we find that level mapping f^1 of Example 18 orients all transitions, strictly orients $S = \{g_1, g_3\}$ and is bounded on $B = \{g_1, g_4\}$. Hence, by choosing the numbering $num_B(p) = 2$, $num_B(q) = 1$, $num_S(p) = 1$, $num_S(q) = 2$, we obtain that g_1, g_3, and g_4 are anchors. Note that the problem encoded to SAT represents the choice of the level mapping and node numbering at once. Now, ρ is set to $\langle f^1 \rangle$, and the anchors are removed from $\mathcal{C}$, leaving a SCC consisting of point p and transition rule g_2. In a second iteration, level mapping f^2 of Example 18 is found and appended to ρ. No SCC remains, and the algorithm terminates.

Note that our algorithm is nondeterministic (due to leaving some decisions to the SAT solver). In this example, the first iteration could come up with the numbering $num_B(p) = num_B(q) = 1$, which would cause only g_1 to be recognized as an anchor. Thus, another iteration would be necessary, which would find a numbering according to which g_3 and g_4 are anchors, since this time there is no other option.

5.3 A SAT encoding

Let $\mathcal{G}$ be a strongly connected MCS (assume the context of the Main Algorithm of Section 5.1). For a compatible pair μ_L, μ_H, we construct a propositional formula $\Phi^{\mathcal{G}}_{\mu_L,\mu_H}$, which is satisfiable iff there exists a level mapping f_{μ_L,μ_H} that anchors some transition rules in $\mathcal{G}$. We focus on tagged level mappings (omitting tags is the same as assigning them all the same value).

Each program point p and argument position i is associated with an integer variable tag_p^i. Integer variables are encoded through their bit representation. In the following, we write, for example, $||n > m||$ to indicate that the relation $n > m$ on integer variables is encoded to a propositional formula in CNF. Let $g = p(\bar{x}) \coloneq \pi; q(\bar{y})$ and consider each $a, b \in \bar{x} \cup \bar{y}$. At the core of the encoding, we use a formula φ^g_{rel}, which introduces a propositional variable $e^g_{a>b}$ to specify a corresponding "tagged edge", $e^g_{a>b} \leftrightarrow \pi \models (a, tag_1) > (b, tag_2)$, as prescribed in equation (1). Here, tag_1 and tag_2 are the integer tags associated with the program points and argument positions of a and b (in g). We proceed likewise for the propositional variable $e^g_{a \geqslant b}$.

Example 33

Consider $g_3 = p(x_1, x_2, x_3) \coloneq y_1 > x_1, x_2 \geqslant y_2; q(y_1, y_2)$ from Example 4. The formula $\varphi^{g_3}_{rel}$ contains (among others) the following conjuncts. From $(y_1 > x_1)$, $(e^{g_3}_{y_1>x_1} \leftrightarrow$ `true`) and $(e^{g_3}_{y_1 \geqslant x_1} \leftrightarrow$ `true`); from $(x_2 \geqslant y_2)$, $(e^{g_3}_{x_2>y_2} \leftrightarrow ||tag_p^2 > tag_q^2||)$ and $(e^{g_3}_{x_2 \geqslant y_2} \leftrightarrow ||tag_p^2 \geqslant tag_q^2||)$. Observe also, $e^{g_3}_{x_1>y_2} \leftrightarrow$ `false` and $e^{g_3}_{x_1 \geqslant y_2} \leftrightarrow$ `false`.

We introduce the following additional propositional variables:

- $weak^g \Leftrightarrow g$ oriented weakly by f_{μ_L,μ_H}
- $strict^g \Leftrightarrow g$ oriented strictly by f_{μ_L,μ_H}
- $bound^g \Leftrightarrow p_f^{high}(\bar{x}) \mathrel{\underline{\overset{\pm}{\supset}}} p_f^{low}(\bar{x})$
- $anchor^g \Leftrightarrow g$ is an anchor w.r.t. f in $\mathscr{G}$
- $weak^g_{low} \Leftrightarrow q_f^{low}(\bar{y}) \gtrsim^{\mu_L} p_f^{low}(\bar{x})$
- $strict^g_{low} \Leftrightarrow q_f^{low}(\bar{y}) \succ^{\mu_L} p_f^{low}(\bar{x})$
- $weak^g_{high} \Leftrightarrow p_f^{high}(\bar{x}) \gtrsim^{\mu_H} q_f^{high}(\bar{y})$
- $strict^g_{high} \Leftrightarrow p_f^{high}(\bar{x}) \succ^{\mu_H} q_f^{high}(\bar{y})$

and, for every program point r, two integer variables num^r_S and num^r_B to represent the node numberings from Definition 30.

Our encoding takes the following form:

$$\Phi^{\mathscr{G}}_{\mu_L,\mu_H} = \left(\bigwedge_{g\in\mathscr{G}} weak^g\right) \wedge \left(\bigvee_{g\in\mathscr{G}} anchor^g\right) \wedge \left(\begin{array}{c} \varphi^{\mathscr{G}}_{rel} \wedge \psi^{\mathscr{G}} \wedge \psi^{\mathscr{G}}_{pos} \wedge \psi^{\mathscr{G}}_{low} \wedge \\ \wedge\ \psi^{\mathscr{G}}_{high} \wedge \psi^{\mathscr{G}}_{bound} \wedge \psi^{\mathscr{G}}_{ne} \end{array}\right).$$

The first two conjuncts specify that f_{μ_L,μ_H} is a level mapping, which orients $\mathscr{G}$, the third is specified as $\varphi^{\mathscr{G}}_{rel} = \bigwedge_{g\in\mathscr{G}} \varphi^g_{rel}$, and the rest are explained below:

Proposition $\psi^{\mathscr{G}}$ imposes the intended meanings on $weak^g$, $strict^g$, and $anchor^g$ (see Definition 20 and Lemma 1).

$$\psi^{\mathscr{G}} = \bigwedge_{g=\,p(\bar{x})\,:-\,\pi;\,q(\bar{y})} \left(\begin{array}{l} weak^g \leftrightarrow (weak^g_{low} \wedge weak^g_{high}) \quad \wedge \\ strict^g \leftrightarrow (weak^g \wedge (strict^g_{low} \vee strict^g_{high})) \quad \wedge \\ anchor^g \leftrightarrow ((p \neq q) \wedge (||num^p_S \neq num^q_S|| \wedge ||num^p_B \neq num^q_B||)) \vee \\ \qquad\qquad\quad ((p = q) \wedge strict^g \wedge bound^g) \end{array}\right).$$

Proposition $\psi^{\mathscr{G}}_{pos}$ enforces that the node numberings num_S and num_B agree with sets S and B, cf. Lemma 1:

$$\psi^{\mathscr{G}}_{pos} = \bigwedge_{g=\,p(\bar{x})\,:-\,\pi;\,q(\bar{y})} \left(\begin{array}{l} (||num^p_S < num^q_S|| \rightarrow strict^g) \wedge \\ (||num^p_B < num^q_B|| \rightarrow bound^g) \end{array}\right).$$

Proposition $\psi^{\mathscr{G}}_{high}$ imposes that $weak^g_{high}$ and $strict^g_{high}$ are `true` exactly when $p_f^{high}(\bar{x}) \gtrsim^{\mu_H} q_f^{high}(\bar{y})$ and $p_f^{high}(\bar{x}) \succ^{\mu_H} q_f^{high}(\bar{y})$, respectively. We focus on the case when $\mu_H = max$, the other cases are similar and omitted for lack of space. The encoding of proposition $\psi^{\mathscr{G}}_{low}$ is similar (and also omitted for lack of space).

$$\psi^{\mathscr{G}}_{high} = \bigwedge_{g=\,p(\bar{x})\,:-\,\pi;\,q(\bar{y})} \left(\begin{array}{l} weak^g_{high} \leftrightarrow \displaystyle\bigwedge_{1\leqslant j\leqslant m} \left(q_j^{high} \rightarrow \bigvee_{1\leqslant i\leqslant n} (p_i^{high} \wedge e^g_{x_i \geqslant y_j})\right) \wedge \\ strict^g_{high} \leftrightarrow \displaystyle\bigwedge_{1\leqslant j\leqslant m} \left(q_j^{high} \rightarrow \bigvee_{1\leqslant i\leqslant n} (p_i^{high} \wedge e^g_{x_i > y_j})\right) \wedge \bigvee_{1\leqslant i\leqslant n} p_i^{high} \end{array}\right).$$

The propositional variables p_i^{low}, p_i^{high}, q_j^{low}, and q_j^{high} $(1 \leqslant i \leqslant n, 1 \leqslant j \leqslant m)$ indicate the argument positions of p/n and q/m selected by the level mapping f_{μ_L,μ_H} for the low and high sets, respectively. The first subformula specifies that a transition rule is weakly oriented by the *max* order if for each j where q_j^{high} is selected (i.e., the jth argument of q is in q^{high}), at least one of the selected positions p_i^{high} has to "cover" q_j^{high} with a weak constraint $x_i \geqslant y_j$. The second subformula is similar for the case of strict orientation with the additional requirement that at least one p_i^{high} should be selected.

Proposition $\psi^{\mathcal{G}}_{bound}$ constrains $bound^g$ to be `true` iff $p^{high}_f \unrhd p^{low}_f$ is satisfied by g. As observed in Section 4, this test boils down to four cases. We illustrate the encoding for the case $min(p^{high}_f(\bar{x})) \geqslant max(p^{low}_f(\bar{x}))$:

$$\psi^{\mathcal{G}}_{bound} = \bigwedge_{g = p(\bar{x}) :- \pi;\, q(\bar{y})} \left(bound^g \leftrightarrow \bigwedge_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n} \left((p^{high}_i \wedge p^{low}_j) \rightarrow e^g_{x_i \geqslant x_j} \right) \right).$$

Proposition $\psi^{\mathcal{G}}_{ne}$ constrains the level mapping so that for each program point p, the sets p^{low} and p^{high} are not empty. Let $\mathcal{P}$ denote the set of program points in $\mathcal{G}$.

$$\psi^{\mathcal{G}}_{ne} = \bigwedge_{p \in \mathcal{P}} \left(\left(\bigvee_{1 \leqslant i \leqslant n} p^{low}_i \right) \wedge \left(\bigvee_{1 \leqslant i \leqslant n} p^{high}_i \right) \right).$$

6 Implementation and experiments

We implemented a termination analyzer based on our SAT encoding for MCNP and tested it on three benchmark suites. Experiments were conducted running the SAT4J (Le Berre and Parrain 2010) solver on an Intel Core i3 at 2.93 GHz with 2 GB RAM. For further details on our experiments see Appendix B and `http://aprove.informatik.rwth-aachen.de/eval/MCNP`.

Suite 1 consists of 81 MCSs obtained from various research papers on termination and from abstracting textbook style C programs.[7] MCNP proves 66 of them terminating with an average runtime of 0.55 s (maximal runtime is 5.15 s). This suite contains the 32 examples from the evaluation of Fuhs *et al.* (2009). That paper introduced *integer term rewrite systems* (ITRSs), where standard operations on integers are predefined, and showed how to use a rewriting-based termination prover like AProVE for algorithms on integers. MCNP shows termination of 27 of these. AProVE[8] proves termination of these 27 and one more example. On the 32 examples from Fuhs *et al.* (2009), the average runtime of MCNP is 0.22 s, whereas the average runtime of AProVE is 5.3 s for the examples with no timeout (AProVE times out after 60 s on four examples). This shows that MCNP is sufficiently powerful for representative programs on integers and demonstrates the efficiency of our SAT-based implementation. The comparison with AProVE on the examples from Fuhs *et al.* (2009) indicates that MCNP has about the same precision and is significantly faster.

Suite 2 originates from the JBC programs in the *JBC* and *JBC Recursive* categories of the *International Termination Competition* 2010.[9] One hundred sixty-five MCS instances were obtained by first applying the preprocessor of the termination analyzer COSTA (Albert *et al.* 2008) resulting in (binary clause) constraint logic programs

[7] Using a translator developed by A. Ben-Shabtai and Z. Mann at Tel-Aviv Academic College.

[8] Using an Intel Core 2 Quad CPU Q9450 at 2.66 GHz with 8 GB RAM.

[9] In this competition, AProVE, COSTA, and Julia competed against each other. See `http://www.termination-portal.org/wiki/Termination_Competition` for details.

with linear constraints. After minor processing, these are abstracted to MCSs (applying SWI Prolog with its library for CLPQ (constraint logic programming with linear constraints). MCNP provides a termination proof for 92 of these with an average runtime of 0.66 s (maximal runtime is 16.31 s). In contrast, COSTA[10] shows termination of 102 programs. However, it encounters a (120 s) timeout on five instances. COSTA's average runtime for the examples with no timeout is 0.076 s. From these experiments, we see that although MCNP is based on very simple ranking functions, it is able to provide many of the proofs and does not encounter timeouts. Moreover, there are five programs where MCNP provides a proof and COSTA does not (four due to timeouts).

Suite 3. Here, the Competition 2010 version of the termination analyzer AProVE abstracts JBC programs from the (nonrecursive) *JBC* category of the Termination Competition 2010 to ITRSs. (This abstraction from Brockschmidt *et al.* (2010) and Otto *et al.* (2010) only works for programs without recursion.) To further transform ITRSs into MCSs, we apply an abstraction that maps terms to their size and replaces nonlinear arithmetic subexpressions by fresh variables. This results in a CLPQ representation, which is further abstracted to MCSs as for Suite 2. For the resulting 127 instances, MCNP provides 63 termination proofs, eight timeouts after 60 s, and an average runtime of 5.76 s (we count timeouts as 60 s). To compare, we apply AProVE directly[11] but fix the abstraction to be the same as in the preprocessor for MCNP. This results in 73 termination proofs and eight timeouts with an average time of 14.16 s. There are five instances where MCNP provides a proof not found by AProVE. Applying AProVE without fixing the abstraction gives 95 termination proofs, 19 timeouts, and an average time of 17.12 s (there are still three instances where MCNP provides a proof not found by AProVE). This shows that the additional proving power in AProVE comes primarily from the search for the right abstraction. Once fixing the abstraction, MCNP is of similar precision and much faster. Thus, it could be fruitful to use a combination of tools where the MCNP analyzer is tried first and the rewrite-based analyzer is only applied for the remaining "hard" examples.

7 Conclusion

We introduced a new approach to prove termination of MC transition systems. The idea is to construct a ranking structure, of a novel kind, extending previous work in this area. To verify whether a MCS has such a ranking structure, we use an algorithm based on SAT solving. We implemented our algorithm and evaluated it in extensive experiments. The results demonstrate the power of our approach and show that its integration into termination analyzers for JBC advances the state of the art of automated termination analysis.

[10] Experiments for COSTA were performed on an Intel Core i5 at 3.2 GHz with 3 GB RAM.
[11] Using an Intel Xeon 5140 at 2.33 GHz with 16 GB RAM and imposing a time limit of 60 s.

Acknowledgement

We thank Samir Genaim for help with the benchmarking. This paper is supported by the G.I.F. grant 966-116.6. Part of Amir M. Ben-Amram's work was carried out while visiting DIKU, the University of Copenhagen.

References

Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G. and Zanardini, D. 2008. Termination analysis of Java Bytecode. In *Proc. of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '08).* Lecture Notes in Computer Science, vol. 5051. Springer-Verlag, Berlin, 2–18.

Alias, C., Darte, A., Feautrier, P. and Gonnord, L. 2010. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proc. of the International Symposium on Static Analysis (SAS '10).* Lecture Notes in Computer Science, vol. 6337. Springer-Verlag, Berlin, 117–133.

Avery, J. 2006. Size-change termination and bound analysis. In *Proc. of the International Symposium on Functional and Logic Programming (FLOPS '06).* Lecture Notes in Computer Science, vol. 3945. Springer-Verlag, Berlin, 192–207.

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

Ben-Amram, A. M. Monotonicity constraints for termination in the integer domain. Accepted for publication in *Logical Methods of Computer Science.* URL: http://arxiv.org/abs/1105.6317.

Ben-Amram, A. M. and Codish, M. 2008. A SAT-based approach to size-change termination with global ranking functions. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08).* Lecture Notes in Computer Science, vol. 4963. Springer-Verlag, Berlin, 218–232.

Ben-Amram, A. M. and Lee, C. S. 2007. Size-change analysis in polynomial time. *ACM Transactions on Programming Languages and Systems 29*(1), 5:1–5:37.

Brockschmidt, M., Otto, C., von Essen, C. and Giesl, J. 2010. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis.* Lecture Notes in Artificial Intelligence, vol. 6463. Springer-Verlag, Berlin, 17–37.

Codish, M., Lagoon, V. and Stuckey, P. J. 2005. Testing for termination with monotonicity constraints. In *Proc. of the International Conference on Logic Programming (ICLP '05).* Lecture Notes in Computer Science, vol. 3668. Springer-Verlag, Berlin, 326–340.

Codish, M., Lagoon, V. and Stuckey, P. J. 2006. Solving partial order constraints for LPO termination. In *Proc. of the International Conference on Rewriting Techniques and Applications (RTA '06).* Lecture Notes in Computer Science, vol. 4098. Springer-Verlag, Berlin, 4–18.

Codish, M. and Taboch, C. 1999. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming 41*(1), 103–123.

Colón, M. and Sipma, H. 2002. Practical methods for proving program termination. In *Proc. of the International Conference on Computer Aided Verification (CAV '02*). Lecture Notes in Computer Science, vol. 2404. Springer-Verlag, Berlin, 442–454.

Dershowitz, N., Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. 2001. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing 12*(1–2), 117–156.

Dershowitz, N. and Manna, Z. 1979. Proving termination with multiset orderings. *Communications of the ACM 22*(8), 465–476.

Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P. and Falke, S. 2009. Proving termination of integer term rewriting. In *Proc. of the International Conference on Rewriting Techniques and Applications (RTA '09)*. Lecture Notes in Computer Science, vol. 5595. Springer-Verlag, Berlin, 32–47.

Giesl, J., Thiemann, R., Schneider-Kamp, P. and Falke, S. 2006. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning 37*(3), 155–203.

Giesl, J., Thiemann, R., Swiderski, S. and Schneider-Kamp, P. 2007. Proving termination by bounded increase. In *Proc. of the International Conference on Automated Deduction (CADE '07)*. Lecture Notes in Artificial Intelligence, vol. 4603. Springer-Verlag, Berlin, 443–459.

Hirokawa, N. and Middeldorp, A. 2005. Automating the dependency pair method. *Information and Computation 199*(1–2), 172–199.

Le Berre, D. and Parrain, A. 2010. The SAT4J library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation 7*, 59–64.

Lee, C. S., Jones, N. D. and Ben-Amram, A. M. 2001. The size-change principle for program termination. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM Press, 81–92.

Lindenstrauss, N. and Sagiv, Y. 1997. Automatic termination analysis of Prolog programs. In *Proc. of the International Conference on Logic Programming (ICLP '97)*. MIT Press, 64–77.

Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. 2004. Proving termination for logic programs by the query-mapping pairs approach. In *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*. Lecture Notes in Computer Science, vol. 3049. Springer-Verlag, Berlin, 453–498.

Manna, Z. and Waldinger, R. 1978. Is "sometime" sometimes better than "always"? *Communications of the ACM 21*, 159–172.

Manolios, P. and Vroon, D. 2006. Termination analysis with calling context graphs. In *Proc. of the International Conference on Computer-Aided Verification (CAV '06)*. Lecture Notes in Computer Science, vol. 4144. Springer-Verlag, Berlin, 401–414.

Otto, C., Brockschmidt, M., von Essen, C. and Giesl, J. 2010. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. of the International Conference on Rewriting Techniques and Applications (RTA '10)*. Leibniz International Proceedings in Informatics, vol. 6. Dagstuhl, Germany, 259–276.

Serebrenik, A. and De Schreye, D. 2004. Inference of termination conditions for numerical loops in Prolog. *Theory and Practice of Logic Programming 4*(5–6), 719–751.

Spoto, F., Mesnard, F. and Payet, E. 2010. A termination analyser for Java Bytecode based on path-length. *ACM Transactions on Programming Languages and Systems 32*(3), 8:1–8:7.