Automata and Program Analysis

Thomas Colcombet   Laure Daviaud   Florian Zuleger

July 12, 2017

Abstract

We show how recent results concerning quantitative forms of automata help providing refined understanding of the properties of a system (for instance, a program). In particular, combining the size-change abstraction together with results concerning the asymptotic behavior of tropical automata yields extremely fine complexity analysis of some pieces of code.

This abstract gives an informal, yet precise, explanation of why termination and complexity analysis are related to automata theory.

Program analysis and termination

Program analysis is concerned with the automatic inference of properties of a chunk of code (or full program). Such analysis may serve many purposes, such as guaranteeing that some division by zero cannot occur in the execution of a program, or that types are properly used (when the language is not statically typed), or that there is no memory leakage, etc... Here, to start with, we are concerned with termination analysis, i.e. proving that all the executions of the program (or pieces of a program) eventually halt.

Consider the following *code* $\mathcal{C}$:

```
void main() {
  uint x,y;
  x = read_input();
  y = read_input();
  while (x >= 0) {
    if (y > 0) { // branch a
       y--;
    }
    else { // branch b
      x--;
      y = read_input();
    }
  }
}
```

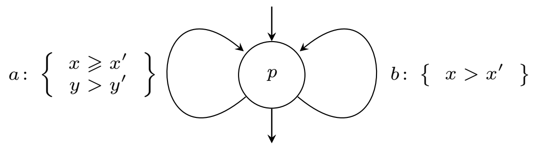
It is clear for a human being that this piece of code eventually *terminates* whatever are the input values read during its execution. The question is how this can be automatically inferred? Of course, such a problem is in general undecidable unless restrictions are assumed (using variations around the theorem of Rice). The path we follow here consists in approximating the behavior of this code using size-change abstraction. We shall see that this abstraction transforms the code into a formalism, the size-change abstraction, for which termination is decidable.

Size-change abstraction

The *size-change abstraction* amounts to abstract a piece of code in the following manner:

- We identify some *size-change variables* that are considered of interest, that range over non-negative integers (in our example x and y). In general, size-change variables can represent any norm on the program state (a function which maps the state to the non-negative integers), such as the length of a list, the height of a tree, the sum of two non-negative variables, etc.
- We construct the *control-flow graph* (possibly simplified) of the code: vertices are positions in the code, and *edges* are steps of computations. We also identify some *entry* and *exit vertices* of the graph as one can expect.
- We abstract tests, i.e., we replace all tests by non-determinism. This means that we consider possible executions independently of whether the `if`-conditions or tests in `while` loops are true or not.
- Finally, each edge of the control-flow graph is labeled by *guards* expressing how the values of the size-change variables may evolve while taking the edge. The language for these relations is very restricted: it consists of a conjunction of properties of the form $x \geqslant y'$ or $x > y'$ where $x, y, \dots$ represent the value of the variables before the edge is taken, while $x', y', \dots$ represent the value of the variables after the edge is taken. We add guards *conservatively* in order to ensure the correctness of the abstraction, i.e., we only add a $x \geqslant y'$ to some edge guard if we can guarantee that the value of y is not greater than the value of x before edge is taken, and similarly for $x < y'$.

  For the code $\mathcal{C}$, we obtain the following size-change abstraction $\mathcal{S}$:


$a\colon \left\{ \begin{array}{c} x \geqslant x' \\ y > y' \end{array} \right\}$
p
$b\colon \{\ x > x'\ \}$

We comment on the size-change abstraction $\mathcal{S}$ of code $\mathcal{C}$. Here, we only consider one position in the code, which is the beginning of the `while` loop. This needs not be the case in general. The two edges a and b correspond respectively to executing the `if` branch and the `else` branch of code $\mathcal{C}$. In the first case (edge a), the value of y strictly decreases while the value of x does not change. In the second case (edge b), the value of x strictly decreases while we have no information about the value that y might take after the transition (because of the '`y = read_input()`' in the code). We see that the two transitions a and b of $\mathcal{S}$ are an *abstraction* of the branches a and b of $\mathcal{C}$: If the code $\mathcal{C}$ executes branch a resp. b and the variable values (x, y) change to some (x', y'), then transition a resp. b of $\mathcal{S}$ also allow the variable values (x, y) to change to (x', y'). This has the following important consequence for termination analysis: Every execution of $\mathcal{C}$ is also an execution of $\mathcal{S}$. Thus, if we can show that $\mathcal{S}$ terminates, then we can deduce the termination of $\mathcal{C}$.

In the following we explain how to reason about the termination of the size-change abstraction $\mathcal{S}$.

In order to be precise, we have to define the semantics of the model. Call an *execution path* of the size-change abstraction a sequence of edges that 1) starts in an entry vertex, and 2) is formed of compatible edges, meaning that for any two consecutive edges in the sequence, the target of the first one should coincide with the source of the second one. Such an execution path is *halting* if it is finite and ends in an exit vertex. Such a definition of an execution path does not yet capture the semantics of variables. For this, we shall consider the traces that realize an execution. Formally, a *trace* of the size-change abstraction is a sequence of *configurations* consisting of a vertex and a valuation of the variables by non-negative integers, that respect the transitions of the size-change abstraction. This is best seen in an example. Consider the execution path *aabaabaa*. One possible trace that realizes this execution is the following one (where the second component represents the value of the variable x and the third one the value of the variable y):

$$\begin{tikzcd}
(p,2,2) \arrow[r, bend right, "a"'] & (p,2,1) \arrow[r, bend right, "a"'] & (p,2,0) \arrow[r, bend right, "b"'] & (p,1,2) \arrow[r, bend right, "a"'] & (p,1,1) \arrow[r, bend right, "a"'] & (p,1,0) \arrow[r, bend right, "b"'] & (p,0,2) \arrow[r, bend right, "a"'] & (p,0,1) \arrow[r, bend right, "a"'] & (p,0,0)
\end{tikzcd}$$

We say that the *size-change abstraction terminates* if every execution path that is realized by a trace is finite. As argued above, this implies that the program also terminates since all the executions of the program are captured by the abstraction. Of course, the converse does not hold, in particular because we threw away a good part of the original semantics, and thus many reasons for the program to terminate are not recovered.

Max-plus Automata

The above definition of size-change abstraction does not make the termination property obviously decidable, yet. What we show now is how this question can be reduced to a problem of universality in automata theory. The key concept behind this last reduction is that the pattern that prevents an execution path to be realizable would be an infinite sequence of variables that are related by the guards via (non-necessarily stricts) inequalities, and infinitely many times by strict inequalities (finite sequences of arbitrarily large number of strict inequalities would also be a witness in some cases). Indeed, such a sequence would mean the existence in a trace of an infinite decreasing sequence of non-negative integers; a contradiction. For instance, the following infinite execution path of the size-change abstraction:

$$\overbrace{a \quad \ldots \quad a}^{n_0 \text{ times}}\ b\ \overbrace{a \quad \ldots \quad a}^{n_1 \text{ times}}\ b\ \overbrace{a \quad \ldots \quad a}^{n_2 \text{ times}}\ b\ \ldots$$

is impossible since the a edge contains the guard $x \leqslant x'$, and the b relation $x < x'$. Hence the impossibility infinite sequence relation:

$$x_0 \overbrace{\leqslant x_1 \leqslant \cdots \leqslant}^{n_0 \text{ inequalities}} x_{n_0-1} < x_{n_0} \overbrace{\leqslant x_{n_0+1} \leqslant \cdots \leqslant}^{n_1 \text{ inequalities}} x_{n_0+n_1} < x_{n_0+n_1+1} \leqslant \quad \cdots\ ,$$

in which x_i accounts for the value assumed by variable x at time i. No valuations of the x_i's by *non-negative integers* could fulfill these constraints. For similar reasons, an infinite execution path that would eventually consists only of the edge b would be impossible, this time because of the variable y.

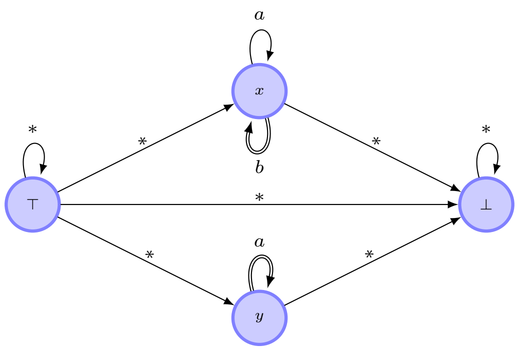
We shall define now an automaton model that can measure these 'bad sequences' of dependences: max-plus automata[1].

We will define a *max-plus automaton* that is able to 'count' the maximal length of such sequences of strict inequalities. The recipe is the following:

[1]Though we take the principle, we do not use the standard notation of max-plus automata, which are traditionally defined as automata weighted over the max-plus semiring.

- The *input alphabet* is the set of edges of the size-change abstraction.
- The *states* of the automaton are the size-change variables, plus two extra states, called $\top$ and $\bot$.
- The *transitions* of the automaton are labeled by the edges of the size-change abstraction, and there is a transition from state x to state y labeled by δ if the guard of edge δ contains either '$x \geqslant y'$' or '$x > y'$'; Furthermore the state $\top$ is the origin of all possible transitions to every state (including itself), and $\bot$ is the target all possible transitions originating from any states (including itself).
- Some transitions are marked *special* (or *costly*): the ones that arose from the case '$x > y'$'.[2]
- All states are marked both *initial* and *final* (the feature of initial and final states, which is important in the theory of tropical automata, happens to be irrelevant for this application).

In our case, this yields the automaton below, in which the $*$ symbol means 'all possible labels', double arrows identify costly transitions, and we omitted to represent initial and final states:



In this automaton, sequence of inequalities over size-change variables can be witnessed by a path (formally, a run). Costly transitions correspond to strict inequalities in the guards.

The semantics of this max-plus automaton is to count the number of costly transitions and to maximize this cost among all runs. Formally, an *input* of the automaton is a sequence of edges of the size-change abstraction, such as the word *aaabaabaa* (it is not necessarily a execution path so far). A priori, we consider both finite and infinite sequences of edges. A *run* of the automaton over the input u is a sequence of transitions which forms a path in the graph of the automaton, and it is accepting if either it is infinite and starts in an initial state or it is finite, starts in an initial state and ends in a final one. The max-plus automaton $\mathcal{A}$ can be used to compute a quantity given some input

[2]In the standard terminology, non-costly transitions would be given *weight* 0, while costly ones would be attributed weight 1.

word u:

$$[\![\mathcal{A}]\!](u) = \sup\{\mathtt{cost}(\rho) \mid \rho \text{ accepting run over the input } u\} \in \mathbb{N} \cup \{\infty\} \ ,$$
$$\text{where} \quad \mathtt{cost}(\rho) = \text{number of costly transitions in } \rho.$$

For instance, over the input $u = aaabaabaa$, $[\![\mathcal{A}]\!](u) = 3$. It corresponds to a run (there are in fact several of them) that assumes state y during the first three letters of the word. This is the maximal one, since in our example the automaton computes the maximum of the number of b-edges with the maximum of the longest block of consecutive a-edges.

The following lemma formalizes the correction of this reduction to automata:

Lemma 1 *The following properties are equivalent:*

1. *The size-change abstraction terminates.*
2. *All infinite execution paths u satisfy $[\![\mathcal{A}]\!](u) = \infty$.*

It happens that Item 2 of Lemma 1 is decidable, from which we get:

Corollary 2 ([10, 9]) *The termination of size-change abstractions is decidable.*

Let us establish the decidability of Item 2. This requires some knowledge about Büchi automata theory. The reader may as well proceed to the next section.

In this proof, rephrase the second item of Lemma 1 as an inclusion of Büchi automata. The first Büchi automaton $\mathcal{E}$ accepts a sequence of edges if it forms a valid infinite execution path of the size-change abstraction. The second Büchi automaton, $\mathcal{B}$, is syntactically the max-plus automaton seen as a Büchi automaton, in which special transitions have to be visited infinitely often (the Büchi condition): it accepts an infinite execution path u if there exists a run containing infinitely many costly transitions. In particular, if u is accepted by $\mathcal{B}$, then $[\![\mathcal{A}]\!](u) = \infty$ (property $\star$). In general the converse is not true, but one can check that if u is ultimately periodic (i.e. of the form $uvvv\dots$), then $[\![\mathcal{A}]\!](u) = \infty$ implies $u \in L(\mathcal{B})$ (property $\star\star$).

Let us now show that Item 2 of Lemma 1 is equivalent to:

$$L(\mathcal{E}) \subseteq L(\mathcal{B}) \ .$$

Indeed, if $L(\mathcal{E}) \subseteq L(\mathcal{B})$, it means that all infinite execution paths u are accepted by $\mathcal{B}$, and thus $[\![\mathcal{A}]\!](u) = \infty$ by $\star$. For the converse direction, assume that $L(\mathcal{E}) \subseteq L(\mathcal{B})$ does not hold, i.e., there is an input that is accepted by $\mathcal{E}$ but not by $\mathcal{B}$. It is known from Büchi that in this case there exists such an input u which is ultimately periodic. By $\star\star$, this means that $[\![\mathcal{A}]\!](u)$ is finite, contradicting Item 2 of Lemma 1.

Complexity analysis

Counting the number of costly transitions also gives an idea of the worst-case complexity of the program. Indeed, a possible execution of the program corresponds to an execution path of the size-change abstraction that is realized by a trace, and the time complexity of the execution is nothing but the length of the execution path.

To put this idea in action, let us consider the following slightly modified code:

```
void main(uint n) {
  uint x,y;
  x = read_input(n);
  y = read_input(n);
  while (x >= 0) {
    if (y > 0) { // branch a
       y--;
    }
    else { // branch b
      x--;
      y = read_input(n);
    }
  }
}
```

This new code takes a non-negative integer as input, and the `read_input(n)` calls now guarantee that the value produced is in the interval $\{0, \dots, n\}$. A more careful look at this code reveals that it terminates in $O(n^2)$. We would like the analysis to reach this level of precision.

In fact, everything is similar to the termination case we have explained up to now. The only change is that the values of the variables are implicitly ranging over the interval $\{0, \dots, n\}$. Under this assumption, the size-change abstraction also terminates within a quadratic bound.

The reduction to a max-plus automaton also remains valid, as shown by this variation around the ideas of Lemma 1:

Lemma 3 *The following properties are equivalent for all n and k:*

- *The size-change abstraction terminates within time bound k, assuming the variable values range in $\{0, \dots, n\}$.*
- *All execution paths u such that $[\![\mathcal{A}]\!](u) \leqslant n$ have length at most k.*

However, we need now a much more delicate result of automata theory than the inclusion of Büchi automata. Here follows what we can do:

Theorem 4 ([7]) *One can effectively compute, given as input a max-plus automaton $\mathcal{A}$, the value*

$$\liminf_{|u|\to\infty} \frac{\log([\![\mathcal{A}]\!](u)+1)}{\log|u|}$$

which happens to be a rational in $[0, 1]$ or ∞.

Now, as a corollary, we get:

Theorem 5 ([7]) *The length of the longest execution path realized by a trace in a size-change abstraction is of order $\Theta(n^\alpha)$ if the variables are restricted to take values in $[0, n]$, where $\alpha \geqslant 1$ is a rational number. Moreover, there is an algorithm that given a terminating size-change abstraction computes such an α.*

It can also be proved that all the rationals $\alpha \geq 1$ can be achieved by a given size-change abstraction.

Related Work

The goal of this paper was to illustrate the size-change abstraction (SCA), which is a popular technique for automated termination analysis. The last decade has seen considerable interest in automated techniques for proving the termination of programs. In this short paper we limit ourselves to describing the related work on SCA. SCA has been introduced by Ben-Amram, Lee and Jones in [10]. SCA is employed for the termination analysis of functional [10, 11], logical [12] and imperative [1, 6] programs, term rewriting systems [5], and is implemented in the industrial-strength systems ACL2 [11] and Isabelle [8]. Recently, SCA has also been used for resource bound and complexity analysis of imperative programs [14], which motivated the results on complexity analysis presented in this paper. SCA is an attractive domain for an automated analysis because of several strong theoretical results on termination analysis [10], complexity analysis [7, 13] and the extraction of ranking functions [3, 13]. Further research has investigated the generalization of size-change constraints to richter classes of constraints, including difference constraints [2], gap-order constraints [4] and monotonicity constraints [3].

References

[1] Hugh Anderson and Siau-Cheng Khoo. Affine-based size-change termination. In *APLAS*, pages 122–140, 2003.

[2] Amir M. Ben-Amram. Size-change termination with difference constraints. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[3] Amir M. Ben-Amram. Monotonicity constraints for termination in the integer domain. *Logical Methods in Computer Science*, 7(3), 2011.

[4] Laura Bozzelli and Sophie Pinchinat. Verification of gap-order constraint abstractions of counter systems. In *VMCAI*, pages 88–103, 2012.

[5] Michael Codish, Carsten Fuhs, Jürgen Giesl, and Peter Schneider-Kamp. Lazy abstraction for size-change termination. In *LPAR (Yogyakarta)*, pages 217–232, 2010.

[6] Michael Codish, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs, and Jürgen Giesl. Sat-based termination analysis using monotonicity constraints over the integers. *TPLP*, 11(4-5):503–520, 2011.

[7] Thomas Colcombet, Laure Daviaud, and Florian Zuleger. Size-change abstraction and max-plus automata. In *MFCS*, pages 208–219, 2014.

[8] Alexander Krauss. Certified size-change termination. In *CADE*, pages 460–475, 2007.

[9] Chin Soon Lee. Ranking functions for size-change termination. *ACM Trans. Program. Lang. Syst.*, 31(3):10:1–10:42, 2009.

[10] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

[11] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *CAV*, pages 401–414, 2006.

[12] Germán Vidal. Quasi-terminating logic programs for ensuring the termination of partial evaluation. In *PEPM*, pages 51–60, 2007.

[13] Florian Zuleger. Asymptotically precise ranking functions for deterministic size-change systems. In *CSR*, pages 426–442, 2015.

[14] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, pages 280–297, 2011.