Resource Control Graphs

Jean-Yves Moyen

Resource Control Graphs

JEAN-YVES MOYEN
University of Paris 13

Resource Control Graphs can be seen as an abstract representation of programs. Each state of the program is abstracted as its size, and each instruction is abstracted as the effects it has on the size whenever it is executed. The Control Flow Graph of the programs gives indications on how the instructions might be combined during an execution.

Termination proofs usually work by finding a decrease in some well-founded order. Here, the sizes of states are ordered and such kind of decrease is also found. This allows to build termination proofs similar to the ones in Size Change Termination.

But the size of states can also be used to represent the space used by the program at each point. This leads to an alternate characterisation of the Non Size Increasing programs, that is the ones that can compute without allocating new memory.

This new tool is able to encompass several existing analysis and similarities with other studies hint that even more analysis might be expressable in this framework thus giving hopes for a generic tool for studying programs.

Categories and Subject Descriptors: D.2.4 [**Software engineering**]: Software/Program Verification; F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and reasoning about Programs; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Abstraction, implicit complexity, non-size increasing, program analysis, size change termination, termination

1. INTRODUCTION

1.1 Motivations

The goal of this study is an attempt to predict and control computational resources like space or time, which are used during the execution of a program. For this, we introduce a new tool called *Resource Control Graphs* and focus here on explaining how it can be used for termination proofs and space complexity management.

We present a data flow analysis of the low-level language sketched by means of Resource Control Graph, and we think that this is a generic concept from which several programs properties could be checked.

The first problem we consider is the one of detecting programs able to compute within a constant amount of space, that is without performing dynamic memory allocation. These were dubbed *Non Size Increasing* by Hofmann [2000].

There are several approaches which are trying to solve this problem. The first protection

Author's address: J.-Y. Moyen, LIPN, Institut Galilée, 99 avenue J.B. Clément, 93430 Villetaneuse, France.

mechanism is by monitoring computations. However, if the monitor is compiled with the program, it could crash unpredictably by memory leak. The second is the testing-based approach, which is complementary to static analysis. Indeed, testing provides a lower bound on the memory while static analysis gives an upper bound. The gap between both bounds is of some value in practical. Lastly, the third approach is type checking done by a bytecode verifier. In an untrusted environment (like embedded systems), the type protection policy (Java or .Net) does not allow dynamic allocation. Actually, the former approach relies on a high-level language, which captures and deals with memory allocation features [Aspinall and Compagnoni 2003]. Our approach guarantees, and even provides, a proof certificate of upper bound on space computation on a low-level language without disallowing dynamic memory allocations.

The second problem that we study is termination of programs. This is done by closely adapting ideas of Lee et al. [2001], Ben-Amram [2006] and Abel and Altenkirch [2002]. The intuition being that a program terminates whenever there is no more resources to consume.

There are long term theoretical motivations. Indeed a lot of works have been done in the last twenty years to provide syntactic characterisations of complexity classes, *e.g.* by Bellantoni and Cook [1992] or Leivant and Marion [1993]. Those characterisations are the bare bone of recent research on delineating broad classes of programs that run in some amount of time or space, like Hofmann, but also Niggl and Wunderlich [2006], Amadio et al. [2004], and Bonfante et al. [2004].

We believe that our Resource Control Graphs will be able to encompass several or even all of these analysis and express them in a similar way. In this sense, Resource Control Graphs are an attempt to build a generic tool for program analysis.

1.2 Coping with undecidability

All these theoretical frameworks share the common particularity of dealing with behaviours of programs (like time and space complexity) and not only with the inputs/outputs relation which only depends on the computed function.

Indeed, a given function can be computed by several programs with different behaviours (in terms of complexity or other). Classical complexity theory deals with functions and computes *extensional* complexity. Here, we want to compute *intensional* or *implicit* complexity, that is try to understand why a given algorithm is more efficient than another to compute the same function.

The study of extensional complexity quickly reaches the boundary of Rice's theorem. Any extensional property of programs is either trivial or undecidable. Intuition and empirical results point out that intensional properties are even harder to decide. Section 3 will formalise this impression.

However, several very successful works do exist for studying both extensional properties (like termination) or intensional ones (like time or space complexity). As these works provide decidable criteria, they must be either incomplete (reject a valid program) or unsound (accept an invalid program). Of course, the choice is usually to ensure soundness: if the program is accepted by the criterion then the property (termination, polynomial bound,...) is guaranteed. This allows the criterion to be seen as a certificate in a proof carrying code paradigm.

When studying intensional properties, two different kinds of approaches exist. The first one consist of restricting the syntax of programs so that any program written necessarily

has the wanted property. This is in the line of the works on primitive recursive functions where the recurrence schemata is restricted to only primitive recursion. This approach gives many satisfactory results, such as the characterisations of PTIME by Cobham [1962] or Bellantoni and Cook [1992], the works of Leivant and Marion on tiering and predicative analysis [1993] or the works of Jones on CONS-free programs [2000]. On the logical side, this leads to explicit management of resources in Linear Logic [Girard 1987].

All these characterisations usually have the very nice property of *extensional completeness* in the sense that, *e.g.*, every polynomial time computable function can be computed by a bounded recursive function (Cobham). Unfortunately, they're usually very poor on the *intensional completeness*, meaning that very few programs fit in the characterisation [Colson 1998] and programmers have to rewrite their programs in a non-natural way.

So, the motto of this first family of methods can be described as leaving the proof burden to the programmer rather than to the analyser. If you can write a program with the given syntax (which, in some cases, can be a real challenge), then certain properties are guaranteed. The other family of methods will go in the other way. Let the programmer write whatever he wants but the analysis is not guaranteed to work.

Since syntax is not hampered in these methods, decidability is generally achieved by loosening the semantics during analysis. That is, one will consider *more* that all the executions a program can have. A trivial example of this idea would be "a program without loop uniformly terminates". The reason we consider loops as bad is because we assume it is always possible to go through the loop infinitely many time. That is, the control of the loop is completely forgotten by this "analysis".

A more serious example of this kind of characterisation is the Size Change Termination [Lee et al. 2001]. The set $FLOW^\omega$ that is build during the analysis contains all "well-formed call sequences". Every execution of the program can be mapped to a well-formed call sequence but several (most) of the call sequences do not correspond to any execution of the program. Then, properties (termination) of call sequences in $FLOW^\omega$ are necessarily shared by all execution of the program.

However, the methods sometimes fails – which is normal since it's a decidable method for partly solving an undecidable problem – because $FLOW^\omega$ does contain well formed call sequences which correspond to no execution of the program but nonetheless do not have the wanted property (*i.e.* are infinite).

This second kind of methods can thus be described as not meddling with the programmer and let the whole proof burden lay on the analysis. Of course, the analysis being incomplete, one usually finds out that certain kinds of programs wont be analysed correctly and have to be rewritten. But this restriction is done *a prosteriori* and not *a priori* and it can be tricky to find what exactly causes the analysis to fail.

This work was greatly inspired by the Size Change Principle (see Section 9 for more on this issue) and is so strongly intended to live within the second kind of analysis.

Section 3 deals with global decidability issues of properties of programs, establishing the fact that the set of poly-time programs is Σ_2-complete and Section 4 will describe the core idea of Resource Control Graphs that can be summed up as finding a decidable (recursive) superset of all the executions that still ensure a given property (such as termination or a complexity bound). Then, Section 5 presents Vectors Addition Systems with States which are generalised into Resource Systems with States in Section 6. They form the backbone of the Resource Control graphs. Section 8 present the tool itself and explain how to build

a Resource Control Graph for a program and how it can be used to study the program. Sections 7 and 9 shows applications of RCG in detecting Non Size Increasing programs or building termination proofs similar to the Size Change Termination principle.

2. STACKS MACHINES

2.1 Syntax

A stacks machine consist of a finite number of *registers*, each able to store a letter of an alphabet, and a finite number of *stack*s, that can be seen as lists of letters. Stacks can only be modified by usual `push` and `pop` operations, while registers can be modified by a given set of operators each of them assumed to be computed in a single unit of time.

Definition 2.1 *(Stacks machine).* Stacks machines are defined by the following grammar:

(Alphabet)	Σ		finite set of symbols
(Programs)		$p ::=$	$\texttt{lbl}_1 : i_1; \ldots \texttt{lbl}_n : i_n;$
(Instructions)	$\mathcal{I} \ni$	$i ::=$	$\texttt{if}$ (test) $\texttt{then goto lbl}_0 \texttt{ else goto lbl}_1 \mid$
			$\mathbf{r} := \texttt{pop}(\texttt{stk}) \mid \texttt{push}(\mathbf{r}, \texttt{stk}) \mid \mathbf{r} := \texttt{op}(\mathbf{r}_1, \cdots, \mathbf{r}_k) \mid \texttt{end}$
(Labels)	$\mathcal{L} \ni$	$\texttt{lbl}$	finite set of labels
(Registers)	$\mathcal{R} \ni$	$\mathbf{r}$	finite set of registers
(Stacks)	$\mathcal{S} \ni$	$\texttt{stk}$	finite set of stacks
(Operators)	$\mathcal{O} \ni$	$\texttt{op}$	finite set of operators

Each operator has a fixed arity k and n is an integer constant. The syntax of a program induces a function $\texttt{next} : \mathcal{L} \to \mathcal{L}$ such that $\texttt{next}(\texttt{lbl}_i) = \texttt{lbl}_{i+1}$ and a mapping $\iota : \mathcal{L} \to \mathcal{I}$ such that $\iota(\texttt{lbl}_k) = i_k$. The `pop` operation removes the top symbol of a stack and put it in a register. The `push` operation copy the symbol in the register onto the top of the stack. The if instruction giving control to either $\texttt{lbl}_0$ or $\texttt{lbl}_1$ depending on the outcome of the test. Each operator is interpreted with respect to a given semantics function $[\![\texttt{op}]\!]$.

The precise sets of labels, registers and stacks can be infered from the program. Hence if the alphabet is fixed, the machine can be identified with the program itself.

The syntax $\texttt{if}$ $(test)$ $\texttt{then goto lbl}_0$ can be used as a shorthand if the second label is the next one. Similarly, $\texttt{goto lbl}$ is a macro for $\texttt{if}$ true $\texttt{then goto lbl}$, that is an unconditional jump to a given label.

If the alphabet contains a single letter, then the registers are useless and the stacks can be seen as unary numbers. The machine then becomes an usual counters machine [Shepherdson and Sturgis 1963].

Example 2.2. The following program reverses a list in stack $\mathbf{l}$ and put the result in stack $\mathbf{l}'$. It uses register $\mathbf{a}$ to store intermediate letters. The empty stack is denoted $[]$.

$$
\begin{array}{l|l}
0 : \texttt{if } \mathbf{l} = [] \texttt{ then goto end}; & 3 : \texttt{goto } 0; \\
1 : \mathbf{a} := \texttt{pop}(\mathbf{l}); & \texttt{end} : \texttt{end}; \\
2 : \texttt{push}(\mathbf{a}, \mathbf{l}'); &
\end{array}
$$

2.2 Semantics

Definition 2.3 *(Stores).* A *store* is a function σ assigning to each register of a program a symbol (letter of the alphabet) and to each stack a finite string in Σ^*. Store update is

$$\frac{i = \iota(\texttt{IP}) = \mathbf{r} := \texttt{op}(\mathbf{r}_1, \cdots, \mathbf{r}_k) \quad \sigma' = \sigma\{\mathbf{r} \leftarrow [\![\texttt{op}]\!](\sigma(\mathbf{r}_1), \ldots, \sigma(\mathbf{r}_k))\}}{p \vdash \langle \texttt{IP}, \sigma\rangle \xrightarrow{i} \langle \texttt{next}(\texttt{IP}), \sigma'\rangle}$$

$$\frac{\iota(\texttt{IP}) = \texttt{if (test) then goto lbl}_1 \texttt{ else goto lbl}_2 \quad \text{(test) is true}}{p \vdash \langle \texttt{IP}, \sigma\rangle \xrightarrow{\text{(test)}_{\text{true}}} \langle \texttt{lbl}_1, \sigma\rangle}$$

$$\frac{\iota(\texttt{IP}) = \texttt{if (test) then goto lbl}_1 \texttt{ else goto lbl}_2 \quad \text{(test) is false}}{p \vdash \langle \texttt{IP}, \sigma\rangle \xrightarrow{\text{(test)}_{\text{false}}} \langle \texttt{lbl}_2, \sigma\rangle}$$

$$\frac{i = \iota(\texttt{IP}) = \mathbf{r} := \texttt{pop}(\texttt{stk}) \quad \sigma(\texttt{stk}) = \lambda.w \quad \sigma' = \sigma\{\mathbf{r} \leftarrow \lambda, \texttt{stk} \leftarrow w\}}{p \vdash \langle \texttt{IP}, \sigma\rangle \xrightarrow{i} \langle \texttt{next}(\texttt{IP}), \sigma'\rangle}$$

$$\frac{i = \iota(\texttt{IP}) = \mathbf{r} := \texttt{pop}(\texttt{stk}) \quad \sigma(\texttt{stk}) = \epsilon}{p \vdash \langle \texttt{IP}, \sigma\rangle \xrightarrow{i} \bot}$$

$$\frac{i = \iota(\texttt{IP}) = \texttt{push}(\mathbf{r}, \texttt{stk}) \quad \sigma' = \sigma\{\texttt{stk} \leftarrow \sigma(\mathbf{r}).\sigma(\texttt{stk})\}}{p \vdash \langle \texttt{IP}, \sigma\rangle \xrightarrow{i} \langle \texttt{next}(\texttt{IP}), \sigma'\rangle}$$

Fig. 1. Small steps semantics

denoted $\sigma\{x \leftarrow v\}$.

Definition 2.4 *(States).* Let p be a stack program. A *state* of p is a couple $\theta = \langle \texttt{IP}, \sigma\rangle$ where the *Instruction Pointer* IP is a label and σ is a store. Let Θ be set of all states, Θ^* (Θ^ω) be the set of finite (infinite) sequences of states and $\Theta^{*\omega}$ be the union of both.

Definition 2.5 *(Executions).* The operational semantics of Figure 1 defines a relation $p \vdash \theta \xrightarrow{i} \theta'$.

An *execution* of a program p is a sequence (finite or not) $p \vdash \theta_0 \xrightarrow{i_1} \theta_1 \xrightarrow{i_2} \ldots \xrightarrow{i_n} \theta_n \ldots$

An infinite execution is said to be *non-terminating*. A finite execution with n states in it is *terminating*. If the program admits no infinite execution, then it is *uniformly terminating*.

We use $\bot$ to denote runtime error. We may also allow operators to return $\bot$ if we want to allow operators to generate errors. It is important to notice that $\bot$ is not a state and hence won't be considered when quantifying over all states.

If the instruction is not specified, we will write simply $p \vdash \theta \to \theta'$ and use $\xrightarrow{+}$, $\xrightarrow{*}$ for the transitive and reflexive-transitive closures.

Definition 2.6 *(Traces).* Let $p \vdash \theta_0 \xrightarrow{i_1} \theta_1 \xrightarrow{i_2} \ldots \xrightarrow{i_n} \theta_n \ldots$ be an execution. Its *trace* is the sequence of all instructions $i_1 \ldots i_n \ldots$

Definition 2.7 *(Length).* Let $\theta = \langle \texttt{IP}, \sigma\rangle$ be a state. Its *length* $|\theta|$ is the sum of the

number of elements in each stack[1]. That is:

$$|\theta| = \sum_{\mathtt{stk} \in \mathcal{S}} |\mathtt{stk}|$$

Length is the usual notion of space. Since there is a fixed number of registers and each can only store a bounded number of different values, the space need to actually store all registers is always bounded. So, we do not take registers into account while computing space usage.

The notion of length allows to define usual time and space complexity classes.

Definition 2.8 *(Running time, running space).* The *time usage* of a finite execution is the number of states in it. The *running time* of a program is an increasing function f such that the time usage of each execution starting at θ is bounded by $f(|\theta|)$.

The *space usage* of a finite execution is the maximum length of a state in it. The *running space* of a program is an increasing function f such that the space usage of each execution starting at θ is bounded by $f(|\theta|)$.

Definition 2.9 *(Complexity).* Let $f : \mathbb{N} \to \mathbb{N}$ be an increasing function. The class $T(f)$ is the set of functions which can be computed by a program whose running time is bounded by f. The class $S(f)$ is the set of function which can be computed by a program whose running space is bounded by f.

As usual, PTIME denotes the set of all functions computable in polynomial time, that is the union of $T(P)$ for all polynomials P and so on.

If we want to define classes such as LOGSPACE, then we must, as usual, use some read only stacks which can only be `pop`ed but not `push`ed and who play no role when computing the length of a state.

2.3 Turing Machines

Stacks machines are Turing complete. We quickly describe here the straightforward way to simulate a Turing machine by a stack machine.

Simulating a TM with a single tape and alphabet Σ can be done with a stack machine with the alphabet $\Sigma \bigcup Q$ (where Q is the set of states of the TM), two stacks and two registers. The two stacks and the first register will encode the tape in an usual way (one stack, reversed, for the left-hand side, the register for the scanned symbol and the other stack for the right-hand side). Another register will contains the current state of the automaton.

At each step, the program will go through a sequence of tests on the state in order to find the right set of instructions to perform and after that jump back to the beginning of the program. There will be at most q such tests where q is the number of states of the TM (a more clever binary search can reduce this to $\log(q)$). Then, simulation of a step is quite easily done by modifying the "scanned symbol" register and then simulating movement.

Simulating movement first has to check that the correct stack is not empty, `push` a "blank symbol" on it if necessary and then `push` the scanned symbol on one stack and `pop` the other stack onto it.

[1] Hence, it should more formally be $||\langle \mathtt{IP}, \sigma \rangle|| = (|\sigma(\mathtt{stk}_1)|, \ldots, |\sigma(\mathtt{stk}_s)|)_{\mathtt{stk}_i \in \mathcal{S}}$. Since explicitly mentioning the store everywhere would be quite unreadable, we use $\mathtt{stk}_i$ instead of $\sigma(\mathtt{stk}_i)$ and, similarly, $\mathbf{r}$ instead of $\sigma(\mathbf{r})$, when the context is clear.

Each step of the TM is simulated in a constant number of steps of the stack machine (depending only on the TM). So that the time complexity of the stack machine will be the same as the time complexity of the TM (up to a multiplicative constant). Similarly, at any step of the simulation, the length of the configuration of the stack machine will be the number of non-blank or scanned symbols on the tape (minus one because one symbol is stored into a register). So the space complexity will be the same.

Notice that the simulation of TM by stack machine preserve space complexity very tightly.

3. SOME DECIDABILITY ISSUES

In this section, we hint that intensional properties are more undecidable than extensional ones by proving this result for polynomial time.

Rice's theorem [Rice 1953] states that any *extensional* property of Turing Machines (or programs) is either trivial or undecidable. An extensional property is one that depends only of the inputs and the output of the machine, *i.e.* that depends only of the function computed by the machine.

However, the polytime property that is often studied is stronger. When studying function, the class PTIME corresponds to those functions who *can be* computed in polynomial time (in the given model). However, there also exist some algorithms (or programs) that do compute the correct function but take much more than polynomial time in order to do so. This is because a single function is computed by several algorithms and some of them can be inefficient.

Typically, in order to compute the Fibonacci's numbers, one can either use the straightforward recursive algorithm that run in exponential time or use some kind of dynamic programming in order to get a polynomial time algorithm. The function computed by both these algorithms is the same and belongs to PTIME, but there still exists algorithms computing it in exponential (or more) time.

However, we want here to study algorithms rather than functions. The polynomial bound that we're looking for should be established on a program by program basis. That is, we're studying here intensional properties of programs, depending on the algorithm used, and not extensional properties depending only on the computed function (that is on the inputs/outputs relation).

Empirically, intensional properties seem even harder to decide that extensional ones. This can be formalised a bit by the following theorem.

THEOREM 3.1 (MARION 00, TERUI 06). *Let p be a program. The question "does p computes in polynomial time" is undecidable* even if we know that p uniformly terminates.

PROOF (MARION AND MOYEN [2006]). Let q be a program and consider the program p that works as follows:

—p answers 0 if its input is 0.

—On input $x \neq 0$, p simulates $q(0)$ for x steps.

  —If $q(0)$ halts within x steps, then p answers 0.

  —Else, p loops for 2^x (or any other large value depending on x) steps and then answers 0.

Obviously, p uniformly terminates. Is it polynomial-time?

Let $T_q(n)$ be the time (number of steps) needed to compute $q(n)$ (may be infinite if $q(n)$ does not terminate).

If $q(0)$ terminates, then there exists $N \in \mathbb{N}$ such that $T_q(0) = N$. The time $T_p(x)$ needed to compute $p(x)$ is bounded by $N + 2^N$, that is a constant value (N only depends on q). Indeed, if $x < N$, then p will run for x steps simulating q and then for an additional 2^x steps looping. This leads to a total of $x + 2^x < N + 2^N$. On the other hand, if $x \geq N$ then p will run for N steps simulating q and then immediately return 0. So, if $q(0)$ terminates, then $p(x)$ runs in constant time $N + 2^N$ for all x.

Notice that there might be some simulation overhead (that is simulating q for N steps might require more than N steps of p) but this overhead is depending only on N and not on x, so that the reasoning is still true.

If $q(0)$ does not terminates, then $p(x)$ runs for $x + 2^x$ steps (x for the simulation (plus an eventual overhead) and 2^x for the final loop), that is exponential time with respect to x.

So, p runs in polynomial time if and only if $q(0)$ terminates. Since the halting problem is not decidable, so is polytime computability of programs. □

Uniform termination of programs, in itself, is a non semi-recursive property. So even with an oracle powerful enough to solve (some) non semi-recursive problems, the intensional property of running in polynomial time is still undecidable! Intensional properties are, indeed, much harder than extensional ones.

Notice that this proof can be easily adapted to show the undecidability of any complexity class (of programs). It is sufficient to change the function computed by p if $q(0)$ does not terminates. Notice also that p does computes the constant function 0 which, as a function, certainly belongs to PTIME. It is really important to separate the extensional property (the function can be computed by some program in polynomial time) from the intensional one (the program we're considering is polytime).

Remark 3.2. A similar proof, for the undecidability of running in polynomial time, based on Hilbert's tenth problem [Matiyasevich 1993] ($p(x+1) = 0$ if some polynomial P has a root x, $2 \times p(x)$ otherwise) was presented as the Geocal ICC workshop in Feb. 2006 by K. Terui. This work has been done independently from a similar result presented by J.-Y. Marion in March 2000 at a seminar in ENS Lyon. This is kind of a folklore result but is nonetheless worth mentioning because lot of confusion is done on the subject.

The above result can be improved. Indeed, the set of programs that run in polynomial time is Σ_2-complete (in the arithmetical hierarchy). Recall, that a typical Σ_2-complete set is the set of partial computable functions.

In order to establish the fact that polytime programs is a Σ_2-complete set, we take a class C of computable functions which contains all constant functions. Assume also that there is a computable set of function codes $\tilde{C}$ which enumerates all functions in C. The set of linear functions $\{x + b \mid \forall b \in \mathbb{N}\}$ satisfies the above hypothesis. Another example is the set of polynomials or the set of affine functions ($a \cdot x + b$).

Next, let $[\![p]\!]$ be the function computed by the program p with respect to an acceptable enumeration of programs. We refer to Rogers' textbook [1967] for background. Say that $T_p(x)$ is the number of steps to execute the program p on input x with respect to some universal (Turing) machine.

Now, define the set of programs whose runtime is uniformly bounded by functions in C:

$$A_T = \{p \mid \exists e \in \tilde{C}\ \forall x,\ T_p(x) < [\![e]\!](|x|)\}$$

THEOREM 3.3 (MARION 00). *The set A_T is Σ_2-complete.*

PROOF (MARION AND MOYEN [2006]). It is clear that the statement which defines A_T is a Σ_2 statement.

Let B be any Σ_2 set defined as follows:

$$B = \{q \mid \exists y \forall x,\ R(q, y, x)\} \qquad R \text{ is a computable predicate}$$

We prove that B is reducible to A_T.

For this, we construct a binary predicate Q as follows. $Q(q, t)$ tests during t steps whether there is a y with respect to some canonical ordering such that $\forall x, R(q, y, x)$ holds. If it holds, $Q(q, t)$ also holds. Here, the predicate Q is computable with a complete Π_1 set as oracle.

Using the s-m-n theorem, there is a program q' such that $[\![q']\!](t) = Q(q, t)$.

(1) Suppose that $q \in B$. We know that there is an y such that $\forall x,\ R(q, y, x)$. It follows that the witness y will be found by Q after t steps. Since the constant function $\lambda z.t$ is in C, we conclude that q' is in A_T.

(2) Conversly, suppose that q' is in A_T. This means that $Q(q', t)$ holds for some t, which yields an y verifying $\forall x,\ R(q, y, x)$

□

Notice that we may change time by space in the above proof, which leads to the following consequence:

COROLLARY 3.4. *Let*

$$A_S = \{p \mid \exists e \in \tilde{C}\ \forall x,\ S_p(x) < [\![e]\!](|x|)\}$$

where $S_p(x)$ is the space use by p on x. The set A_S is Σ_2-complete.

Depending on the choice of the set of functions C, this proves the Σ_2-completeness of the following sets:

—The set of programs running in polynomial time (for polynomials functions and A_T).

—The set of programs running in exponential time.

—The set of programs running in logarithmic space or in polynomial space.

—Almost any set of program defined by a space or time bound.

Since the arithmetical hierarchy is separated, this means that all these problems are strictly harder than the halting problem.

4. A TASTE OF RCG

This section describes the idea behind Resource Control Graph in order to get a better grip on the more formal definitions later on.

4.1 Control and memory

The undecidability results means that given a program it is impossible to say if the set of executions, Υ, and Θ^ω, the set of infinite sequences of states, are disjoint. So, the idea here is to find a set $\mathcal{A}$ of *admissible* sequences, which is a superset of the set of all executions, and whose intersection with Θ^ω can be computed. If this intersection is empty, then *a*

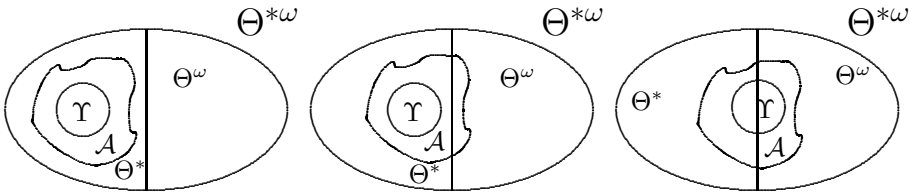

Fig. 2. Sequences of states, executions and admissible sequences

fortiori, there is no infinite executions but if the intersection is not empty, then we cannot say if this is due to some non-termination of the program or if some of the sequences added for the analysis caused trouble. This means that depending on the machine considered and the way $\mathcal{A}$ is build, we can be in three different situations as described in Figure 2. We build $\mathcal{A} \supset \Upsilon$ such that $\mathcal{A} \bigcap \Theta^\omega$ is recursive. If its empty, then the program uniformly terminates. else, we cannot say anything. Of course, the undecidability theorem means that if we require $\mathcal{A}$ (or at least $\mathcal{A} \bigcap \Theta^\omega$) to be recursive, then there will necessarily be some programs for which the situation will be the one in the middle (in Figure2), that is we falsely suppose that the program does not uniformly terminate.

Clearly, states can be split up into a control part, namely the label, and a memory part, namely the store. This two parts interact in both directions: the control can change the memory via assignments of a new value to a register or stack and the memory part changes the behaviour of the control when tests are performed.

By analogy to Turing machines, the interaction of the control over the memory corresponds to *writing* while the interaction of the memory over the control corresponds to *reading*. Figure 3 describes this situation. On Turing machines, the control part will be the automaton while the memory part is the tapes. On C or any other imperative programming language, the control part is the program itself and the memory is the content of the heap and stack. For functional programming, we can see the current function as the control part and the values of its parameters as memory.

The control part is completely finite because there are only finitely many labels. On the other hand, the memory part is infinite because there are infinitely many different strings. So, the first idea will be to really split states in two in order to have a finite representation of the control part that can then, in a more or less dynamical way, be completed by the memory.

4.2 The folding trick

The first try at building such an admissible set of executions will be to completely forget the memory part and only keep the control part. So we study elements of $\mathcal{L}^{*\omega}$ which are build over a finite alphabet ($\mathcal{L}$) while executions are build over an infinite one (since there are infinitely many stores).

Given a word over labels, we can "fold" it by identifying all identical labels in it into a single vertex of a graph and then adding edges between two vertices if and only if they appear in sequence somewhere in the word. Because of the syntax of the program, not all edges will appear in such a graph and, even more, this folding trick can be applied to all possible executions (only keeping the labels) yielding to a single graph called the Control Flow Graph of the program.

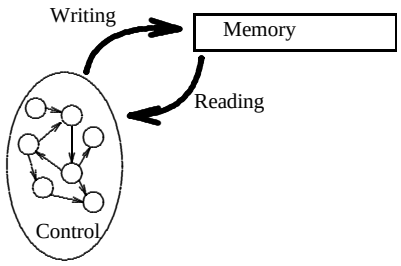

Fig. 3. Control and memory interactions.

Definition 4.1 *(Control Flow Graph)*. Let p be a program. Its *Control Flow Graph* (CFG) is a directed graph $G = (S, A)$ where:

—$S = \mathcal{L}$. There is one vertex for each label.

—If $\iota(\texttt{lbl}) = \texttt{if}$ (test) $\texttt{then goto lbl}_1\ \texttt{else goto lbl}_2$ then there is one edge from $\texttt{lbl}$ to $\texttt{lbl}_1$ labelled $(\text{test})_{\text{true}}$ and one from $\texttt{lbl}$ to $\texttt{lbl}_2$ labelled $(\text{test})_{\text{false}}$.

—If $\iota(\texttt{lbl}) = \texttt{end}$ then there is no edge going out of $\texttt{lbl}$.

—Otherwise, there is one edge from $\texttt{lbl}$ to $\texttt{next}(\texttt{lbl})$ labelled $\iota(\texttt{lbl})$.

Both vertices and edges are named after the label or instruction they represent. No distinction are made between the vertex and the label or the edge and the instruction as long as the context is clear.

Example 4.2. The CFG of the reverse program is displayed on Figure 4.

Now, to each execution corresponds a path (finite or not) in the CFG. The converse, however, is not true. There are paths in the CFG that correspond to no executions.

Let $\mathcal{P}$ be the set of paths in the CFG. $\mathcal{P}$ is a regular language over the alphabet of the edges (see Lemma 6.19), hence $\mathcal{P}$ is recursive. Since we can associate a path to each execution, we can say that $\mathcal{P}$ is a superset of Υ[2]. So, we can choose $\mathcal{A} = \mathcal{P}$ and have a first try at an admissible set of sequences of executions.

However, as soon as the graph contains loops, $\mathcal{P}$ will contain infinite sequences. So this is quite a poor try at building an admissible set of sequences, corresponding exactly to the trivial analysis "*A program without loop uniformly terminates*".

In order to do better, we need to plug back the memory into the CFG.

4.3 Walks

So, in order to take memory into account but still keep the CFG, we will not consider vertices any more but states again. Clearly, each state is associated to a vertex of the CFG. Moreover to each instruction i, we can associate a function $[\![i]\!]$ such that for all states θ, θ' such that $p \vdash \theta = \langle \texttt{IP}, \sigma \rangle \xrightarrow{i} \langle \texttt{IP}', \sigma' \rangle = \theta'$, we have $\sigma' = [\![i]\!](\sigma)$.

So, instead of considering paths in the graph, we can now consider walks. Walks are sequences of states following a path where each new store is computed according to the semantics function $[\![i]\!]$ of the edge just followed.

[2]in some loose sense of "superset" that can be tolerated in this informal description.

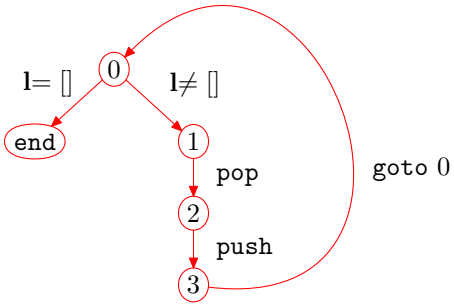

Fig. 4. CFG of the reverse program.

But if we do this exactly that way, then there will be a bijection between the executions and the walks and nothing can be studied.

Paths can be seen as removing both interaction of control and memory. Indeed, a path only looks at the control part of a program and is completely oblivious of its memory. When trying to plug back memory, if we keep both interaction (reading and writing) then we only manage to describe again the executions (in terms of walks). So we need to find something that lay between the two.

The idea at this point is to keep both branches of a test as possible whenever a test is encountered. With the CFG, when there is a vertex with out-degree ≥ 2, both paths are considered. So we will now relax the constraints on walks in order to do the same. That is, replace $[\![i]\!]$ with the identity for tests. This can be seen as removing the reading part of the program and keeping only the writing part.

This yield to a bigger set of walks. However, in certain cases (depending on the shape of the semantics functions for instructions, hence on the set of operators of the program), this set will be decidable. Often, in order to achieve decidability, it is necessarily to consider not stores (and states) but only approximations of such (*e.g.*, only the total size of each store).

5. VECTOR ADDITION SYSTEM WITH STATES

This section describes Vectors Addition Systems with States (VASS). Resources Control Graphs are a generalisation of VASS. VASS are known to be equivalent to Petri Nets [Reutenauer 1989].

In a directed graph[3] $G = (S, A)$, will write $s \xrightarrow{a} r$ to say that a is an edge between s and r. Similarly, we will write $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ to say that $a_1 \dots a_n$ is a path going through vertices $s_0, \cdots, s_n$. Or simply $s_0 \xrightarrow{w} s_n$ if $w = a_1 \dots a_n$. $s \to s'$ means that there exists an edge a such that $s \xrightarrow{a} s'$ and $\xrightarrow{+}$, $\xrightarrow{*}$ are the transitive and reflexive-transitive closures of $\to$.

Definition 5.1 *(VASS, configurations, walks).* A *Vector Addition System with States* is a directed graph $G = (S, A)$ together with a *weighting function* $\omega : A \to \mathbb{Z}^k$ where k is a fixed integer.

[3]We will use $s \in S$ to designate vertices and $a \in A$ to designates edges. The choice of using french initials ("Sommet" and "Arête") rather than the usual (V, E) is done to avoid confusion between vertices and valuations.

A *configuration* is a couple $\theta = (s, v)$ where $s \in S$ is a vertex and $v \in \mathbb{Z}^k$ is the *valuation*. A configuration is *admissible* if and only if $v \in \mathbb{N}^k$.

A *walk* is a sequence (finite or not) of configurations $(s_0, v_0) \xrightarrow{a_1} \dots \xrightarrow{a_n} (s_n, v_n)$ such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$ and for all $i \in \mathbb{N}^*$, $v_i = v_{i-1} + \omega(a_i)$. A walk is *admissible* if all configurations in it are admissible.

We say that path $a_1 \dots a_n$ is the *underlying path* of the walk and the walk *follows* this path. Similarly, G is the *underlying graph* for the VASS.

As for graphs and paths, we will write $\theta \to \theta'$ if there exists an edge a such that $\theta \xrightarrow{a} \theta'$ and $\xrightarrow{+}$, $\xrightarrow{*}$ for the closures.

Definition 5.2 *(Weight of a path).* Let V be a VASS and $a_1 \dots a_n$ be a path in it. The weight of edges is extended to paths canonically: $\omega(a_1 \dots a_n) = \sum \omega(a_i)$.

LEMMA 5.3. *Let V be a VASS and $a_1 \dots a_n$ be a finite path in it. There exists a valuation v_0 such that for $0 \leq i \leq n$, $v_0 + \omega(a_1 \dots a_i) \in \mathbb{N}^k$.*

This means that every finite path is the underlying path of an admissible walk.

PROOF. Because the path is finite, the jth component of $\omega(a_1 \dots a_i)$ is bounded from below by α_j (of course, this bound is not necessarily reached with the same i for all components, but nonetheless such a bound exists for each component separately). By putting $\beta_j = \max(0, -\alpha_j)$ (that is 0 if α_j is positive), then $v_0 = (\beta_1, \cdots, \beta_k)$ verifies the property. □

LEMMA 5.4. *Let $(s_0, v_0) \to \dots \to (s_n, v_n)$ be an admissible walk in a VASS. Then, for all $v'_0 \geq v_0$ (component-wise comparison), $(s_0, v'_0) \to \dots \to (s_n, v'_n)$ is an admissible walk (following the same path).*

PROOF. By monotonicity of the addition. □

Definition 5.5 *(Uniform termination).* A VASS is said to be *uniformly terminating* if it admits no infinite admissible walk. That is, every walk is either finite or reaches a non-admissible configuration.

THEOREM 5.6. *A VASS is* not *uniformly terminating if and only if there exists a cycle whose weight is in $\mathbb{N}^k$ (that is, is positive with respect to each component).*

PROOF. If such a cycle exists, starting and ending at vertex s, then by Lemma 5.3 there exists v_0 such that the walk starting at (s, v_0) and following it is admissible. After following the cycle once, the configuration (s, v_1) is reached. Since the weight of the cycle is positive, $v_1 \geq v_0$. Then, by Lemma 5.4 the walk can follow the cycle one more time, reaching (s, v_2), and still be admissible. By iterating this process, it is possible to build an infinite admissible walk.

Conversely, let $(s_0, v_0) \to \dots \to (s_n, v_n) \dots$ be an infinite admissible walk. Since there are only finitely many vertices, there exists at least one vertex s' appearing infinitely many times in it. Let (s'_k, v'_k) be the occurrences of the corresponding configurations in the walk. Since the component-wise order over vectors is a well partial order, there exists i, j such that $v'_i \leq v'_j$. The cycle followed between s'_i and s'_j has a positive weight. □

Definition 5.7 *(Characteristic matrix, Parikh vector).* Let V be a VASS and consider a given enumeration $a_1, \cdots, a_n$ of the edges. Its *characteristic matrix* Γ is the $k \times n$ matrix whose ith column is $\omega(a_i)$.

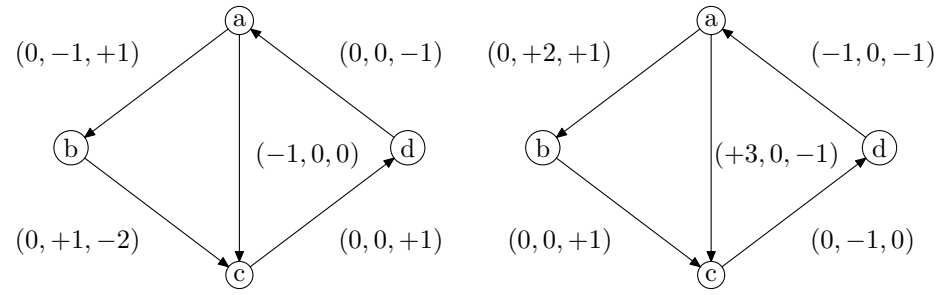

Fig. 5. Two VASS

Given an enumeration $s_1, \cdots, s_m$ of the vertices of V, its *connectivity matrix* Γ' is the $m \times n$ matrix where:

—$\Gamma'_{i,j} = -1$ if $s_i \xrightarrow{a_j} s'$ for some vertex s'

—$\Gamma'_{i,j} = 1$ if $s' \xrightarrow{a_j} s_i$ for some vertex s'.

—$\Gamma'_{i,j} = 0$ otherwise.

Let $p = a_1 \dots a_q$ be a path. Its *Parikh vector* t_p is a vector in $\mathbb{N}^n$ whose ith component is the number of occurrences of a_i in p.

LEMMA 5.8. *The weight of a path p is $\Gamma \times t_p$.*

This is due to the commutativity of the addition.

THEOREM 5.9. *A VASS with characteristic matrix Γ and connectivity matrix Γ' is* not *uniformly terminating if and only if the system:*

$$\begin{aligned} X &> O \\ \Gamma' \cdot X &= 0 \\ \Gamma \cdot X &\geq 0 \end{aligned}$$

admits a solution. This can be checked in polynomial time.

PROOF. X is the Parikh vector of a cycle whose weight is positive, hence the VASS is not uniformly terminating due to Theorem 5.6.

The first inequation ensure that X is the Parikh vector of a non-empty walk. The equation ensure that it is the Parikh vector of a cycle and the second inequation ensure that the weight of this cycle is positive.

Since the set of solutions is a cone (that is if X is a solution then so is $\alpha \times X$ for all α), the existence of a real solution ensure the existence of an integer one. Since we're only concerned with the existence of a solution, this can be solved in polynomial time by usual Linear Programming techniques. □

Since VASS and Petri nets are equivalent, this also shows that uniform termination of Petri nets is decidable. Without going through the equivalence, a direct and very similar proof can be made for Petri nets (characteristic matrix and Parikh vectors also exist, and there is no need for connectivity matrix). Such a proof an be found in [Moyen 2003], (theorem 60, page 83).

Example 5.10. Figure 5 display two VASS. More formally, the first one should be described as a graph $G = (S, A)$ with:

—$S = \{a, b, c, d\}$
—$A = \{a \xrightarrow{a_1} b, a \xrightarrow{a_2} c, b \xrightarrow{a_3} c, c \xrightarrow{a_4} d, d \xrightarrow{a_5} a\}$
—$\omega(a_1) = (0, -1, +1)$, $\omega(a_2) = (-1, 0, 0)$, $\omega(a_3) = (0, +1, -2)$, $\omega(a_4) = (0, 0, +1)$, $\omega(a_5) = (0, 0, -1)$.

Its characteristic matrix Γ_1 and connectivity matrix Γ'_1 are:

$$\Gamma_1 = \begin{vmatrix} 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & +1 & 0 & 0 \\ +1 & 0 & -2 & +1 & -1 \end{vmatrix} \quad \Gamma'_1 = \begin{vmatrix} -1 & -1 & 0 & 0 & +1 \\ +1 & 0 & -1 & 0 & 0 \\ 0 & +1 & +1 & -1 & 0 \\ 0 & 0 & 0 & +1 & -1 \end{vmatrix}$$

If we put $X = (u, v, x, y, z)$, the system of Theorem 5.9 becomes:

$$u \geq 0, v \geq 0, x \geq 0, y \geq 0, z \geq 0$$
$$\begin{array}{rcl|rcl} u+v+x+y+z & \geq & 1 & u+v & = & z \\ -v & \geq & 0 & u & = & x \\ x & \geq & u & v+x & = & y \\ u+y & \geq & z+2x & y & = & z \end{array}$$

Since this system admits no solution, the VASS is uniformly terminating.

For the second VASS, which is build on the same underlying graph (hence, the connectivity matrix is the same), the characteristic matrix Γ_2 is:

$$\Gamma_2 = \begin{vmatrix} 0 & +3 & 0 & 0 & -1 \\ +2 & 0 & 0 & -1 & 0 \\ -1 & -1 & +1 & 0 & -1 \end{vmatrix}$$

And so the system is:

$$u \geq 0, v \geq 0, x \geq 0, y \geq 0, z \geq 0$$
$$\begin{array}{rcl|rcl} u+v+x+y+z & \geq & 1 & u+v & = & z \\ 3v & \geq & z & u & = & x \\ 2u & \geq & y & v+x & = & y \\ x & \geq & u+v+z & y & = & z \end{array}$$

This system admits solutions with: $u = x = 2v$ and $y = z = 3v$, among other, there is the integer solution $(2, 1, 2, 3, 3)$ which is the Parikh vector of a cycle with positive weight and so, for example, the walk:$(a, (5, 6, 2)) \to (b, (5, 8, 3)) \to (c, (5, 8, 4)) \to (d, (5, 7, 4)) \to (a, (4, 7, 3)) \to (c, (7, 7, 2)) \xrightarrow{+} (a, (6, 6, 1)) \to (b, (6, 8, 2)) \xrightarrow{+} (a, (5, 7, 2))$. Since $(5, 7, 2) \geq (5, 6, 2)$, this walk can then be repeated infinitely and leads to an infinite admissible walk.

It is worth noticing that in the second case, the cycle detected is *not* a simple cycle. So the problem is different from the one of detecting simple cycles in graphs and require a specific solution.

6. RESOURCE SYSTEMS WITH STATES

Resource Systems with States (RSS) are a generalisation of VASS seen in the previous section. In VASS, the only information kept if a vector of integers and only additions

of vectors can be performed on them. When modelling programs, this is not sufficient. Indeed, if one want to closely represent the memory of a stack machine, a vector is not sufficient. Moreover, vector addition is not powerful enough to represent usual operations such as copy of a variable ($x := y$).

Hence, we will now relax the constraints on valuations and weights and basically allow any set to be the set of valuations and any kind of functions (between valuations) to be a weight. Notice that for VASS, the addition of a vector v could be represented as the function $\lambda x.x + v$.

In order to be a bit more general, we will even allow the sets of valuations to be different for each vertex. This may seems strange, but a typical use of that is to have vectors with different numbers of components as valuations (that is the set of valuations for vertex s_i would be $\mathbb{Z}^{k_i}$) and matrix multiplications as weights (where the matrices have the correct number of rows and columns). Of course, it is always possible to take the (disjoint) union of these sets, but it usually clutters needlessly the notations. In section 9, we'll use RSS with separate sets of valuations and discuss this a bit further.

6.1 Graphs and States

Definition 6.1 *(RSS).* A *Resource System with States* (RSS) is a tuple (G, V, V^+, W, ω) where

—$G = (S, A)$ is a directed graph, $S = \{s_1, \cdots, s_n\}$ is the set of vertices and $A = \{a_1, \cdots, a_m\}$ is the set of edges.

—$V_1, \cdots, V_n$ are the sets of *valuations*. V is the union of all of them.

—$V_i^+ \subset V_i$ are the sets of *admissible valuations*. V^+ is the union of them.

—$W_{i,j} : V_i \to V_j$ are the sets of *weights*. W is the union of them.

—$\omega : A \to W$ is the *weighting function* such that $\omega(a) : A \to W_{i,j}$ if $s_i \xrightarrow{a} s_j$.

When both the valuations and weights sets are clear, we will name the RSS after the underlying graph G.

The idea behind having both valuations and admissible valuations is that this allows V to have some nice algebraic properties not shared by V^+. Moreover, this also allows the set of valuations to be the closure of the admissible valuations under the weighting functions, thus removing the deadlock problem of reaching something that would not be a valuation (and replacing it by the more semantical problem of detecting non admissible valuations). Typically with VASS, V is $\mathbb{Z}^k$, thus being a ring, and V^+ is $\mathbb{N}^k$. Since weights can add any vector, with positive or negative components, to a valuation, V is the closure of V^+ by this operation. Moreover, VASS do not suffer from the deadlock problems that appear in Petri nets (but this is done by introducing the problem of deciding if a walk is admissible).

Notice that either unions (for V, V^+ or W) can be considered to be a disjoint union without loss of generality.

Definition 6.2 *(Walk).* Let (G, V, V^+, W, ω) be a RSS. A *configuration* in G is a pair $\theta = (s, v)$ where $s = s_i \in S$ is a vertex of the graph and $v \in V_i$ is a valuation. A configuration is *admissible* is $v \in V_i^+$ is admissible.

A *walk* is a sequence of states $(s_0, v_0) \xrightarrow{a_1} \dots \xrightarrow{a_n} (s_n, v_n)$ such that $p = a_1 \dots a_n$ is a path in G and $v_i = \omega(a_i)(v_{i-1})$. A walk is *admissible* if every configuration in it is admissible.

The walk *follows* path p which is called either *underlying path* or *trace* of the walk.

As earlier, we write $\theta \to \theta'$ if the relation holds for an unspecified edge and $\xrightarrow{+}$, $\xrightarrow{*}$ for the transitive and reflexive-transitive closures.

Definition 6.3 *(Weight).* Let G be a RSS. The weighting function can be canonically extended over all paths in G by choosing $\omega(ab) = \omega(b) \circ \omega(a)$.

$(W, \circ)$ is a magma. It is not a monoid because the identity is not unique. There is a finite set of neutral elements, the identities over each V_i.

Of course, If we consider V_i as dots and $\omega \in W$ as arrows, we have a category. Indeed, identity exists for each V_i and composition of two arrows is properly defined.

Notice that we do not actually need the whole W. Only the part generated by the individual weights of edges is necessary to handle a RSS. We will often overload the notation and call it W as well.

In practise, it is often more convenient to describe W as a set together with some right-action on V. That is, there is an operation $\circledast : V \times W \to V$ such that $v \circledast \omega(a) = \omega(a)(v)$. In this case, the function composition becomes an internal law of W, $⨟ : W \times W \to W$ such that $\omega(a) ⨟ \omega(b) = \omega(b) \circ \omega(a)$.

This notation is a much more convenient when composing weights along a path. Indeed, we have $\omega(ab) = \omega(a) ⨟ \omega(b)$, that is the weights are composed in the same order as the edges along the path while using functional composition we had $\omega(ab) = \omega(b) \circ \omega(a)$, needing to reverse the order of edges along the path. Moreover, since weight usually have some common shape, $(W, ⨟)$ is often a well known algebraic structure.

Example 6.4. For the VASS of previous section, we have $V_i = \mathbb{Z}^k$ and $V_i^+ = \mathbb{N}^k$ for all i, and $\omega(a) = \lambda x.x + \alpha$ for some vector $\alpha \in \mathbb{Z}^k$. Or we could describe VASS by saying that $V = W = \mathbb{Z}^k$, $V^+ = \mathbb{N}^k$ and $\circledast = ⨟ = +$.

The notation with $\circledast$ and $⨟$ is much more convenient, especially to handle easily weights of paths such as done in the lemmas and theorems of the previous section.

Moreover, the fact that weights (as functions) all have the same shape (namely, $\lambda x.x+\alpha$) allows to identify each weight with the vector α, thus giving a more convenient definition.

6.2 Properties of RSS

6.2.1 *Order*

Definition 6.5 *(Ordered RSS).* An *ordered RSS* is an RSS $G = (G, V, V^+, W, \omega)$ together with a partial ordering $\prec$ over valuations such that the restriction of $\prec$ over V^+ is a well partial order.

Remember that a partial order $\prec$ is a well partial order if there are no infinite anti-chain, that is for every infinite sequence $x_1, \cdots, x_n, \ldots$ there are indexes $i < j$ such that $x_i \preceq x_j$. This mean that the order is well-founded (no infinite decreasing sequence) but also that there is no infinite sequence of pairwise incomparable elements. The order induced by the divisibility relation on $\mathbb{N}$, for example, is not a well partial order since the sequence of all prime numbers is an infinite sequence of pairwise incomparable elements.

For VASS, the component-wise order on vectors of the same length is the well partial order (over $V^+ = \mathbb{N}^k$) that was used in the previous section.

In the following, we will not always explicitly mention if a RSS is ordered.

Definition 6.6 *(Monotonicity, positivity)*. Let (G, V, V^+, W, ω) be an ordered RSS. We say that it is *monotonic* if all weighting function $\omega(a_i)$ are increasing with respect to $\prec$. Since the composition of increasing functions is still increasing, the weighting function of any path will be increasing.

We say that (G, V, V^+, W, ω) is *positive* if $\prec$ is such that for each $v \in V^+$ and $v' \in V$, $v \prec v'$ implies $v' \in V^+$.

VASS are both monotonic and positive. Monotonicity is the key of Lemma 5.4 while positivity is implicitly used in the proof of Theorem 5.6 to say that the valuation reached after one cycle is still admissible.

Definition 6.7 *(Resource awareness)*. Let G be an ordered RSS and $f : V \to V$ be a function. G is *f-resource aware* if for all walk $(s_0, v_0) \xrightarrow{*} (s_n, v_n)$ we have $v_n \preceq f(v_0)$

6.2.2 *Uniform termination*

Definition 6.8 *(Uniform termination)*. Let G be a RSS. G is *uniformly terminating* if there is no infinite admissible walk in G.

Notice that if a RSS is not uniformly terminating, then there exists an infinite admissble walk that stay entirely within one strongly connected component of the underlying graph. In the following, when dealing with infinite walks we may suppose without loss of generality that the RSS is strongly connected.

For VASS, uniform termination is decidable as stated in Theorem 5.9. It is worth noticing that it does only depends on the underlying graph and the weights and not on the valuations. Indeed, since uniform termination is a global property that must hold for all valuations, this will often be the case.

THEOREM 6.9. *If G does* not *uniformly terminates then there is an admissible cycle $(s, v) \xrightarrow{+} (s, u)$ with $v \preceq u$. If G is monotonic and positive, then this is an equivalence.*

PROOF. If an infinite admissible walk exists, then we can extract from it an infinite sequence of admissible states (s', v_k) since there is only a finite number of vertices. Since the order is a well partial order on V^+, there exists a $i < j$ with $v_i \preceq v_j$, thus leading to the cycle.

If the cycle exists, then it is sufficient to follow it infinitely many time to have an infinite admissible walk. Monotonicity is needed to ensure that every time one follows the cycle, the valuation does indeed increase. Positivity is needed to ensure than when going through always increasing valuations one will never leave V^+. □

Of course, this is simply the generalisation of Theorem 5.6.

PROPOSITION 6.10.

(*1*) *If V is finite, then W is finite.*

(*2*) *If V is finite, then uniform termination of G is decidable.*

(*3*) *If both V and W are enumerable, then it is semi-decidable to know if a RSS is not uniformly terminating.*

PROOF.

(1) Because the set of functions $\mathcal{F}(V, V)$ is finite and contains W.

(2) Since both V and W are finite, it is possible to computes all the values $v \circledast \omega(a)$ and check whether one is both increasing (with respect to $\prec$) and corresponds to a cycle.
(3) By enumerating $V \times W$.

□

6.2.3 *Grounded RSS*

Definition 6.11 *(Grounded RSS).* An ordered RSS $G = (G, V, V^+, W, \omega)$ is *grounded* if there is a set $\widehat{V} \subset V^+$ of *grounded valuations* such that:

—Each finite subset of ground valuations has a ground valuation as a least upper bound (that is, $(\widehat{V}, \preceq)$ is a join-semi lattice).
—V^+ is build from $\widehat{V}$ by adding a least upper bound for each infinite subset (if it does not already exist).

Non-grounded valuations are used to represent "unknown" values. For example, we could allow $+\infty$ to appear in weights as well as valuations in VASS. In this case, $+\infty$ in a valuation would mean that nothing is known about this component of the vector but we keep it admissible by default. However, we keep grounded valuations as a reminder that at some point something should be known before we are allowed to say anything.

In the following, non-admissible valuations are considered to be ground valuations, but $\widehat{V}$ only designates the ground and admissible valuations.

If G is positive, then every non-ground valuation is admissible. Even more, non-ground valuations are the maximal elements of V^+:

LEMMA 6.12 (POSITIVITY). *If $v \in V^+ \setminus \widehat{V}$ is a non ground valuation and $v \prec u$, then $u \notin \widehat{V}$.*

PROOF. Because v has been added a the join of an infinite subset of $\widehat{V}$ and if $u \in \widehat{V}$, this would not have been necessary because u would already have been a join for this subset. □

Definition 6.13 *(Grounding walks).* Let G be a grounded RSS, Consider a walk (finite or not) $w = (s_0, v_0) \xrightarrow{i_1} \dots \xrightarrow{i_n} (s_n, v_n) \dots$ and let $(\widehat{v})_{\mathbb{N}}$ be a sequence of ground admissible valuations. Let $\theta_j = (s_j, v_j)$.

We say that $(\widehat{v})_{\mathbb{N}}$ is an *oracle* for w if the following properties hold:

—If $v_0 \in \widehat{V}$ then $\widehat{v}_0 = v_0$ else $\widehat{v}_0 \preceq v_0$.
—For each $j > 0$, we have $(s_{j-1}, \widehat{v}_{j-1}) \xrightarrow{i} (s_j, v'_j)$. If v'_j is a ground valuation, then $\widehat{v}_j = v'_j$ else $\widehat{v}_j \preceq v'_j$.

The *grounded walk* $\widehat{w}$ is $\theta_0 \rightsquigarrow \widehat{\theta_0} \to \theta_1 \rightsquigarrow \widehat{\theta_1} \to \dots \to \theta_n \rightsquigarrow \widehat{\theta_n}$ where $\widehat{\theta}_j = (s_j, \widehat{v}_j)$. $\rightsquigarrow$ denotes the replacement of a non-ground valuation by the corresponding one from the oracle. We may simply write the grounded walk as $\widehat{\theta_0} \to \dots \to \widehat{\theta_n}$.

A walk is *ground-admissible* if there exists an oracle for it.

The idea is that as soon as a non-ground valuation is reached, it is replaced by a ground one (which is provided by the oracle) and the walk resumes following the same path. Using grounded RSS and grounding walks can be seen as introducing some kind of non

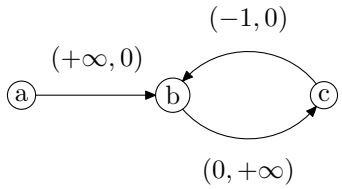

Fig. 6. A grounded VASS

determinism into RSS. Sometimes, when the valuation cannot be completely specified (that is, is non-ground), a random valuation can be chosen.

Notice that whenever the valuation reached by using the regular rules of RSS is ground, then the oracle is not allowed to change it. That is, we don't need to ask anything to the oracle as long as everything can be computed. However, as soon as something "unknown" is reached, then we ask the oracle what this should be and go on along the walk.

Definition 6.14 *(Uniform termination)*. A grounded RSS is *uniformly terminating* if it admits no infinite ground-admissible walk.

This goes with the idea of "unknown" as stated above. Indeed, unknown spreads faster than knowledge and this definition of uniform termination for grounded RSS prevents things such as a prefix of the walk setting the valuation to "unknown" and then there is a cycle looping infinitely with unknown valuations, but the same cycle could no be followed infinitely with known valuations.

If we consider again VASS with $+\infty$ allowed, then we could have a prefix of the walk setting the valuation to $(+\infty, \dots, +\infty)$ and then a loop with a negative weight that could be taken infinitely many times just because $+\infty$ stays $+\infty$ (no knowledge is gained). This definition of uniform termination could be stated as allowing the RSS to forget (part of) the valuation but still remind what it is suppose to represent. The replacement of a valuation in the infinite walk means that something can be unknown but we must remember that this actually represent a ground valuation and should be able to set it so any time.

With grounded RSS, Theorem 6.9 becomes:

THEOREM 6.15. *Let G be a positive monotonic grounded RSS. It is* not *uniformly terminating if and only if there is a ground-admissible cycle $(s, v_0) \xrightarrow{+} (s, v'_n)$ with oracle $(\widehat{v})_{0..n}$ such that $\widehat{v}_0 \preceq \widehat{v}_n$.*

PROOF. If a ground-admissible infinite walk exists, then we can extract from it a ground-admissible cycle in exactly the same way as in Theorem 6.9.

Conversely, if a ground-admissible cycle exists, it can be followed infinitely many time (the oracle being build from the repetition of the oracle for the cycle). □

Example 6.16. Figure 6 displays a grounded VASS, that is a VASS where $+\infty$ is allowed in weights as well as valuation. The ground valuations are those where no component is $+\infty$. Here, the non-ground valuations can be seen as a lack of information: if a component is $+\infty$, it means that nothing is known on the real value which should be here. Typically, this might arise if the operation one want to represent cannot be described as the addition of a vector (*e.g.*, it would need a multiplication).

The notion of oracle for walks is important. Indeed, without this notion of grounded walks, we could consider the walk $(a,(0,0)) \to (b,(+\infty,0)) \to (c,(+\infty,+\infty))$ which can then loop forever with the valuation $(+\infty,+\infty)$. This walk can be seen as first forgetting everything (including the fact that we should be working over integers) and then looping forever in blissful ignorance. However, the loop itself has weight -1 on the first component and so should not really be taken infinitely many times.

So, we need an oracle who knows everything and is able to give the lost knowledge back. Since no oracle leads to an infinite ground-admissible walk, we say that this grounded VASS is uniformly terminating.

Notice also the importance of the condition $\widehat{v}_i \preceq v'_i$. Indeed, without it the oracle could be use to reset the first component of the valuation whenever it is interrogated, thus leading to things like: $(c,(1,0)) \to (b,(0,0)) \to (c,(0,+\infty)) \rightsquigarrow (c,(1,0))$. So this condition $\widehat{v}_i \preceq v'_i$ can here be seen as tightly keeping all that is still known. Allowing to take a smaller valuation is harmless for two reasons. Firstly, when considering only uniform termination, smaller things mean faster termination and secondly since results are usually quantified over all ground-admissible walks, the ones with bigger things also get considered.

6.3 Semi-linearity

Definition 6.17 *(Linear parts, semi-linear parts).* Let $(M,+)$ be a commutative monoid. A *linear part* of M is a subset of the form $u + U^*$ where $u \in M$ and U is a finite part of M. That is, if $U = \{u_1, \cdots, u_p\}$, a linear part can be expressed as:

$$\{u + \sum_{i=1}^{i=p} n_i v_i | n_i \in \mathbb{N}\}$$

A *semi-linear part* of M is a finite union of linear parts.

LEMMA 6.18. *In a commutative monoid, rational parts are exactly the semi-linear parts.*

Recall that rational parts are build from $+$, union and Kleene's star *. When dealing with words (that is the free monoid generated by a finite alphabet), the $+$ is word concatenation (not commutative) and so rational parts are exactly the regular languages.

PROOF. Semi-linear parts are clearly expressed as rational parts.

Conversely, it is sufficient to show that the set of semi-linear parts contains all finite parts and is closed by union, sum and *. The hard point being the closure under * which is a consequence of commutativity. See [Reutenauer 1989] (Proposition 3.5) for details. □

LEMMA 6.19. *The set of cycles in a graph is a rational part (of the free monoid generated by the edges).*

PROOF. Consider the graph as an automaton with each edge labelled by a separate label. The set of paths between two given vertices is a regular language (accepted by the automaton with the proper input and accepting nodes). So is the set of cycles as finite union of regular languages. □

COROLLARY 6.20. *The set of weights of cycles in a RSS is a rational part of W.*

PROOF. Because the weighting function is a morphism between the free monoid generated by the edges and $(W, \fatsemi)$. □

THEOREM 6.21. *If $(W, ⨟)$ is commutative, then the set of weights of cycles is a semi-linear part of it.*

This allows to easily find candidates for Theorem 6.9. Indeed, we need to find a valuation v and a cycle of weight w such that $v \circledast w \succeq v$. Since the construction of the semi-linear part is effective, it is possible to have a concise representation of weights of cycles.

This hints that commutativity of $⨟$ plays an important role in decidability of uniform termination. It is however not a necessary condition (as shown by δSCT [Ben-Amram 2006]) and maybe not a sufficient one.

7. MONITORING SPACE USAGE

In this section, we introduce the notion of Resource Control Graph for the very special case of monitoring space usage. In Section 8, this notion will be fully generalised to define Resource Control Graphs.

7.1 Space Resource Control Graphs

Definition 7.1 *(Weight).* For each instruction i, we define a *weight* k_i as follows:

—The weight of any instruction that is neither `push` nor `pop` is 0.

—The weight of a `push` instruction is $+1$.

—The weight of a `pop` instruction is -1.

PROPOSITION 7.2. *For all stores σ such that $p \vdash \theta = \langle \mathtt{IP}, \sigma\rangle \xrightarrow{i} \theta'$, we have $|\theta'| = |\theta| + k_i$.*

It is important here that both θ and θ' are states. Indeed, this means that when an error occurs ($\bot$), we remove all constraints.

Definition 7.3 *(Space Resource Control Graph).* Let p be a program. Its Space Resource Control Graph (Space-RCG) is a RSS $G = (G, V, V^+, W, \omega)$ where:

—G is the Control Flow Graph of p.

—$V = \mathbb{Z}$, $V^+ = \mathbb{N}$.

—For each edge i, The weighting function $\omega(i)$ is $\lambda x.x + k_i$.

Of course, since all weights have the same shape, they can simply be identified by the constant k_i. In this case, we'll have $\omega(i) = k_i$, $\circledast = ⨟ = +$. That is, the Space Resource Control Graph can be seen as an usual weighted graph.

PROPOSITION 7.4. *Let p be a program, G be its Space-RCG and $p \vdash \theta_1 = \langle \mathtt{IP}_1, \sigma_1\rangle \to \ldots \to \theta_n = \langle \mathtt{IP}_n, \sigma_n\rangle$ be an execution with trace t, then there is an admissible walk $(\mathtt{IP}_1, |\theta_1|) \to \ldots \to (\mathtt{IP}_n, |\theta_n|)$ with the same trace t.*

PROOF. By construction of the RCG and induction on the length of the execution. □

7.2 Characterisation of Space usage

THEOREM 7.5. *Let f be a total function $\mathbb{N} \to \mathbb{N}$. Let p be a program and G be its Space-RCG.*

$p \in S(f)$ if and only if for each state $\theta_0 = \langle \mathtt{IP}_0, \sigma_0\rangle$ and each execution $p \vdash \theta_0 \xrightarrow{*} \theta_n$, the trace of the execution is also the trace of an admissible walk $(\mathtt{IP}_0, |\theta_0|) \to (\mathtt{IP}_1, i_1) \to \ldots \to (\mathtt{IP}_n, i_n)$ and for each k, $i_k \leq f(|\theta_0|)$.

PROOF. Proposition 7.4 tells us that $i_k = |\theta_k|$. Then, both implications hold by definition of space usage. □

COROLLARY 7.6. *Let $f : \mathbb{N} \to \mathbb{N}$ be a total function, p be a program and G be its Space-RCG.*
If G is f-resource aware, then $p \in S(f)$.

Here, the converse is not true because the Space-RCG can have admissible walks with uncontrolled valuations but who do not correspond to any real execution.

7.3 Non Size Increasingness

The study of Non Size Increasing (NSI) functions was introduce by Hofmann [1999]. Formers syntactical restrictions for PTIME, such as the safe recurrence of Bellantoni and Cook [1992], forbid to iterate a function because this can yield to super-polynomial growth. However, this prevents from using perfectly regular algorithms such as the insertion sort where the insertion function is iterated. The idea is then that iterating functions *who do not increase the size of data* is harmless.

In order to detect these functions, Hofmann uses a typed functional programming language. A special type, $\diamond$, is added. This type has no closed terms (that is, no constructors), and can only be used in variables. It can be seen as pointers to free memory. Constructors of other types now require one of more $\diamond$. Typically, the usual `cons` for lists require a $\diamond$ in addition to the data and the list and will then be typed $\texttt{cons} : \diamond \times \alpha \times L(\alpha) \to L(\alpha)$.

Whenever a list is destroyed, the $\diamond$ in the `cons` is freed (in a variable) and can thus be later reused to build another list. By ensuring a linear type discipline, one can be sure that no $\diamond$ is ever duplicated. Then, any program that can be typed with this type system can be computed in a NSI way, *e.g.* be compiled into `C` without any `malloc` instruction.

With Space RCG, valuations in a walk play exactly the same role as Hofmann's diamonds ($\diamond$). The higher the value, the more diamonds are needed in the current configuration. `push` has positive weight, meaning that it uses diamonds but `pop` has negative weight, meaning that it releases diamonds for later use.

Definition 7.7 *(Non Size Increasing).* A program is *Non Size Increasing* (NSI) if its space usage is bounded by $\lambda x.x + \alpha$ for some constant α.

NSI is the class of functions which can be computed by Non Size Increasing programs. That is $\bigcup_\alpha S(\lambda x.x + \alpha)$.

PROPOSITION 7.8. *Let p be a program and G be its Space-RCG. If G is $\lambda x.x + \alpha$-resource aware for some constant α, then p is NSI.*

PROOF. This is a direct consequence of Theorem 7.5. □

THEOREM 7.9. *Let p be a program and G be its Space-RCG. G is $\lambda x.x + \alpha$-resource aware (for some α) if and only if it contains no cycle of strictly positive weight.*

PROOF. If there is no cycle of strictly positive weight, then let α be the maximum weight of any path in G. Since there is no cycle of strictly positive weight, it is well-defined. Consider a walk $(s_0, v_0) \xrightarrow{*} (s_n, v_n)$ in G. Since α is the maximum weight of a path, we have $v_n \leq v_0 + \alpha$. Hence, G is $\lambda x.x + \alpha$-resource aware.

Conversely, if there is a cycle of strictly positive weight, the it can be followed infinitely many time and provides an admissible walk with unbounded valuations. □

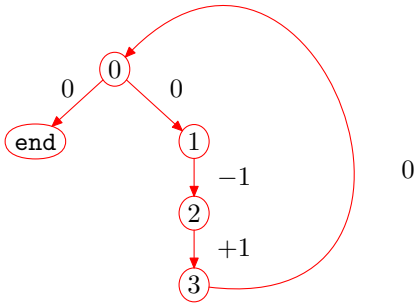

Fig. 7. Space-RCG of the reverse program.

Building the Space-RCG can be done in linear time in the size of the program. Finding the maximum weight of a path can be done in polynomial time in the size of the graph (so in the size of the program) with Bellman-Ford's algorithm ([Cormen et al. 1990] chapter 25.5). So we can detect NSI programs and find the constant α in polynomial time in the size of the program.

Example 7.10. The Space-RCG of the reverse program (from Example 2.2) is displayed on Figure 7. Since it contains no cycle of strictly positive weight, the program is Non Size Increasing. Moreover, since the maximum weight of any path is 0, it can be computed in space $\lambda x.x$, that is the constant α is 0 for this program.

This result, however, lacks an intentionality statement (how much of all NSI programs are caught?) or even an extensional completeness one (does there exist functions in NSI that are not captured by such a program?) Of course, the class NSI is undecidable and the class of all *programs* which are NSI is Σ_2-complete according to Theorem 3.3. This means that intentionality statements are hard to achieve. However, we can reach an extensional completeness one.

Without loss of generality, we consider here that in the initial configuration of a TM, the tape consists in only blank symbols except for a *consecutive* sequence of symbols which are *all* non-blank. That is, we do not allow input tape to have the shape $x \sqcup y$ where x and y are non-blank symbols and $\sqcup$ is the blank symbol. This allows to detect the end of input as the first non-blank symbol. The head is assumed to scan the first non-blank symbol at the beginning of computation.

PROPOSITION 7.11 (NORMALISING TMS). *Let M be a NSI Turing Machine running in space $\lambda x.x + \alpha$. There exists a TM $\widetilde{M}$, computing the same function, running in space $\lambda x.x + \alpha + 2$, proceeding in 2 phases:*

(*1*) *Firstly, $\widetilde{M}$ writes 2 $\#$ and α B on blank squares of its tape, where both $\#$ and B are new symbols.*

(*2*) *Secondly, $\widetilde{M}$ never scan any blank symbol again.*

PROOF. $\widetilde{M}$ starts by going one square left and writing $\#$ there. Then it goes to the end of the input, write α B after it (since α is fixed for M and does not depend on the input, this is doable) and lastly another $\#$. Then, it goes back to the beginning of the input (the symbol immediately after the $\#$) and goes into the second phase.

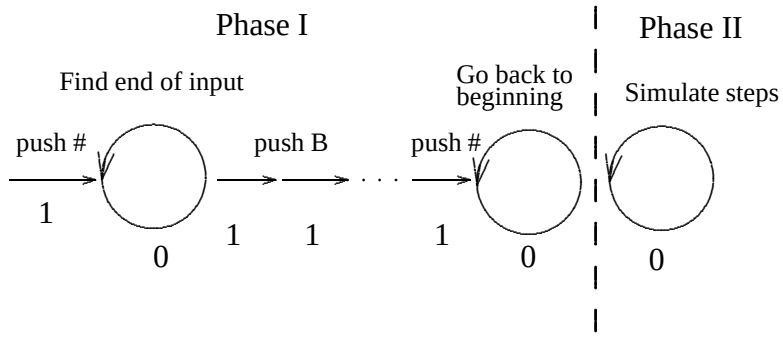

Fig. 8. Space-RCG of the simulation of a normalised NSI Turing machine.

In the second phase, $\widetilde{M}$ simulates M. However, $\#$ are never overwritten. Whenever M request to write over a $\#$, the whole content of the tape is shifted one square left (or right) and simulation of M resumes where it stopped. Since M is NSI, the simulation can be done entirely between the two $\#$. □

Of course, this simulation is rather costly from a time point of view, but since we're only controlling space here, that does not matter. Notice also that such a normalisation could be made for a TM running in space $f(x)$ for any computable function f. However, in that case the simulation would require an additional tape to compute $f(x)$ from the input and then allocate sufficiently many new squares. This would be quite tricky to do and require control over the space used to compute $f(x)\ldots$

THEOREM 7.12 (EXTENSIONAL COMPLETENESS). *Let M be a NSI TM, $\widetilde{M}$ be the corresponding normalised machine and $\widetilde{p}$ be the program simulating $\widetilde{M}$ according to the simulation of Section 2.3. Let $\widetilde{G}$ be the Space-RCG of $\widetilde{p}$.*
$\widetilde{G}$ contains no cycle of strictly positive weight.

PROOF. During the second phase of the simulation of M, $\widetilde{M}$ never scan a blank symbol. Hence, there is no need to push new (blank) symbol on any of the stacks. While moving the head, each push on one stack is immediately followed by a pop and the other, thus yielding only paths of weight 0.

During the first phase of the simulation, p starts by adding a symbol on a stack (a blank symbol immediately erased by $\#$, or alternatively directly a $\#$ with a slightly smarter simulation). Then it loops to find the end of the input. During this loops, each push is also followed by a pop, thus creating only cycles of weight 0. Then it adds $\alpha + 1$ new symbols (B and $\#$), but since α does not depend on the input, this can be done by $\alpha + 1$ separate push, thus creating no cycle. And lastly it goes back to the start of the input, again each push is followed immediately by a pop. Figure 8 shows how the Space-RCG of $\widetilde{p}$ looks like. □

This result means that our characterisation of NSI is extensionally complete. Each function in NSI can be computed by a program which fit into the characterisation (that is, whose Space-RCG is $\lambda x.x + \alpha$-resource aware). Of course, intentional completeness (capturing all NSI programs) is far from reached (but is unreachable with a decidable algorithm).

7.4 Linear Space

LINSPACE seems to be closely related to NSI. Indeed, LINSPACE functions can be computed in space $\lambda x.\beta x + \alpha$ and so NSI is a special case of LINSPACE with $\beta = 1$. So we want to adapt our result to detect linear space usage.

The idea is quite easy: since we're allowed to used β time more space than what is initially allocated, it is sufficient to consider that every time some of the initial data is freed, β "tokens" ($\diamond$) are released and can later be used to control β different allocations.

In order to do so, the most convenient way is to design certain stacks of the machine (or certain tapes of a TM) as *input stacks* and the others must be initially empty. Then, a $\mathtt{pop}$ operation over an input stack would have weight $-\beta$ instead of simply -1 to account for this linear factor. However, doing so we must be careful that newly allocated memory (that is, further $\mathtt{push}$) will only be counted as 1 when freed again (to avoid a cycle of freeing one slot, allocating β, freeing these β slots and reallocating β^2 and so on). In order to do so, we simply require that the input stacks are read-only in the sense that it is not possible to perform a $\mathtt{push}$ operation on them.

Notice that any program can be turned into such kind of program by having twice more stacks (one input and one work for each) and starting by copying all the input stacks into the corresponding working stacks and then only deal with the working stacks.

With these programs, the invariant will not be the length of states, but something slightly more complicated, namely β times the length of input stacks plus the length of work stacks. We will call this measure *size*. Globally, we'll use size to denote some kind of measure on states that is used by the RCG for analysis. The terminology is close from the one of the Size Change Termination [Lee et al. 2001] where values are assumed to have some (well-founded) "size ordering" which is not specified and not necessarily related to the actual space usage of the data. Typically, termination of a program working over positive integers can be proved using usual ordering of $\mathbb{N}$ as size ordering, even if the integers are all 32 bits integers, thus taking exactly the same space in memory.

Definition 7.13. The set of stacks is now partitioned in two. $\mathcal{S}_i$ is the set of *input stacks* and $\mathcal{S}_w$ is the set of *working stacks*. There are two instructions $\mathtt{pop}_i$ and $\mathtt{pop}_w$ depending on whether an input or working stack is considered but only one $\mathtt{push} = \mathtt{push}_w$ instruction, that is it is impossible to $\mathtt{push}$ anything on an input stack.

The *β-size* of a state is β times the length of input stacks plus the length of working stacks, that is:

$$||\langle \mathtt{IP}, \sigma\rangle||_\beta = \beta \sum_{\mathtt{stk}_i \in \mathcal{S}_i} |\mathtt{stk}_i| + \sum_{\mathtt{stk}_w \in \mathcal{S}_w} |\mathtt{stk}_w|$$

The *weight* of $\mathtt{pop}_i$ is $-\beta$, the weight of $\mathtt{pop}_w$ is -1, the weight of $\mathtt{push}$ is $+1$. the weight of other instructions is 0.

The β-Space RCG is build as the Space-RCG: the underlying graph is the control flow graph and the weighting function of each edge is $\lambda x.x + k_i$ where k_i is the weight of the corresponding instruction. Alternatively, we can identify weighting functions with the constant k_i.

Proposition 7.4 becomes:

PROPOSITION 7.14. *Let p be a program, G_β be its β-Space RCG and $p \vdash \theta_1 = \langle \mathtt{IP}_1, \sigma_1\rangle \to \ldots \to \theta_n = \langle \mathtt{IP}_n, \sigma_n\rangle$ be an execution with trace t, then there is an admissible walk $(\mathtt{IP}_1, ||\theta_1||_\beta) \to \ldots \to (\mathtt{IP}_n, ||\theta_n||_\beta)$ with the same trace t.*

Then, adapting Theorem 7.5 and Theorem 7.9, we have:

PROPOSITION 7.15. *Let p be a program and G_β be its β-Space RCG. If G_β is $\lambda x.x + \alpha$-resource aware for some constant α, then $S(p) \leq \lambda x.\beta x + \alpha$.*

THEOREM 7.16. *Let p be a program and G_β be its β-Space RCG. G_β is $\lambda x.x + \alpha$-resource aware (for some α) if and only if it contains no cycle of strictly positive weight.*

COROLLARY 7.17. *Let p be a program. If there exists β such that its β-Space RCG contains no cycle of strictly positive weight, then p is in* LINSPACE.

This can be checked in NPTIME since β is clearly polynomially bounded in the size of the program.

For LINSPACE also, the normalisation process of Turing Machine can quite easily be performed, typically by using an input (read-only) tape and a working tape were space usage is counted. The first phase of the normalised TM consist of repeatedly copy one symbol from the input tape to the right of the working tape and add $\beta - 1$ B at the left of the working tape, then putting the two $\#$ on the working tape. This means that here also the characterisation is extensionally complete: for each LINSPACE function, there exists one program computing it that fits into the characterisation.

8. RESOURCES CONTROL GRAPHS

Instead of the simple weighted graphs used for Space-RCG, we will now use any RSS to modelise programs. A set of admissible valuations will be given to each state and weighting functions simulate the corresponding instruction.

Since we can now have any approximation of the memory (the stores) for valuations, we cannot simply use the length of a state. Instead, we consider given a *size function* that associate to each state (or to each store) some size. The size function is unspecified in general. Of course, when using RCG to modelise programs, the first thing to do is usually to determine a suitable size function (according to the studied property). Notice that depending on the size function, weights of instructions can or cannot be defined properly (that is, some sizes are either too restrictive or too loose and no function can accurately reproduce on the size the effect of a given instruction on actual data). In this case, the RCG cannot be defined and another size function has to be considered.

8.1 Resources Control Graphs

Definition 8.1 *(RCG).* Let p be a program and G be its control flow graph. Let V^+ be a set of admissible valuations (and $\prec$ be a well partial order on it). Let $|| \bullet || : \Theta \to V^+$ be a size function from states to valuations and $V^+_{\mathtt{lbl}}$ be the image by $|| \bullet ||$ of all states $\langle \mathtt{lbl}, \sigma \rangle$ for all stores σ.

For each i edge of G, let $\omega(i)$ be a function such that for all stores σ verifying $p \vdash \langle \mathtt{IP}, \sigma \rangle = \theta \xrightarrow{i} \theta'$, $\omega(i)(||\theta||) = ||\theta'||$. Let V be the closure of V^+ by all the weighting functions $\omega(i)$.

The *Resource Control Graph* (RCG) of p is the RSS build on G with weights $\omega(i)$ for each edge i, valuations V and admissible valuation V^+ (ordered by $\prec$). $V^+_{\mathtt{lbl}}$ being the admissible valuations for vertex $\mathtt{lbl}$.

Example 8.2.

(1) Of course, a Space-RCG as defined earlier is a special case of general RCG. In this case, $||\theta|| = |\theta|$, this leads to $V^+_{\mathtt{lbl}} = V^+ = \mathbb{N}$ for each label $\mathtt{lbl}$. Similarly, $\omega(i) = \lambda x.x + k_i$ with k_i as in definition 7.1. Since $k \in \mathbb{Z}$, the closure of V^+ by the weighting functions is $V = \mathbb{Z}$.

(2) For a more accurate representation of programs, we could choose $||\langle \mathtt{IP}, \sigma\rangle|| = (|\mathtt{stk}_1|, \ldots, |\mathtt{stk}_s|)_{\mathtt{stk}_i \in \mathcal{S}}$, the vector where each component is the length of a stack (given an enumeration of the stacks). This would leads to $V^+_{\mathtt{lbl}} = V^+ = \mathbb{N}^s$ where s is the number of stacks for each label $\mathtt{lbl}$, $\prec$ being the component-wise partial order. In this case, $\omega(i) = \lambda x.x + z$ where $z \in \mathbb{Z}^s$. This leads to $V = \mathbb{Z}^s$. That is, in this case, RCG are VASS.

(3) However, even this representation can be improved. Typically, using these VASS it is impossible to detect anything happening to registers. If we have a suitable size function $||\bullet|| : \Sigma \to \mathbb{N}$ for registers, we can choose $||\langle \mathtt{IP}, \sigma\rangle|| = (||\mathbf{r}_1||, \ldots, ||\mathbf{r}_r||)_{\mathbf{r}_i \in \mathcal{R}}$. In this case, depending on the operators, weight could be either vectors addition or matrices multiplication (to allows copy of a register).

As stated before, we will write $v \circledast \omega(i)$ instead of $\omega(i)(v)$ and $\omega(i) \fatsemi \omega(j)$ instead of $\omega(j) \circ \omega(i)$.

LEMMA 8.3. *Let p be a program, G be its RCG and $p \vdash \theta_0 \to \ldots \to \theta_n$ be an execution with trace t. There exists an admissible walk $(s_0, ||\theta_0||) \to \ldots \to (s_n, ||\theta_n||)$ with the same trace t.*

COROLLARY 8.4. *Let p be a program and G be its RCG. If G is uniformly terminating, then p is also uniformly terminating.*

Remark 8.5. Taking exactly the image of $||\bullet||$ as the set of admissible valuations V^+ might be a bit too harsh. Indeed, this set might have any shape and is probably not really easy to handle. So, it is sometimes more convenient to consider a superset of it in order to easily decide if a valuation is admissible or not. The convex hull (in V) of the image of $||\bullet||$ is typically such a superset. Notice that it is very similar to the idea of trying to find an admissible set of sequences of states which will be more manageable than the set of executions. Here, we try to find an admissible set of valuations which is more manageable than the actual set of sizes. For more details on how to build and manage such a convex superset, see the work of Avery [2006].

Remark 8.6. The size function is not specified and may depend on the property one want to study. We do not address here the problem of finding a suitable size function for a given program. As hinted, it might be a simple vector of functions over stacks and register but it can also be a more complicated function such as a linear combination or so. Hence, with a proper size function, one is able not only to check that a given register (seen as an integer) is always positive but also that a given register is always bigger that another one. This is similar to Avery's functional inequalities [2006].

Using $||\langle \mathtt{IP}, \sigma\rangle|| = (|\mathtt{stk}_1|, \ldots, |\mathtt{stk}_s|)_{\mathtt{stk}_i \in \mathcal{S}}$ as a size measure, that is VASS as RCG, we can already achieve a good termination analysis. Indeed, as previously seen, uniform termination of VASS is decidable and uniform termination of the RCG induce uniform termination of the program.

This kind of RCG has weight $(0, \ldots, 0, -1, 0, \ldots, 0)$ (resp. $(0, \ldots, 0, +1, 0, \ldots, 0)$) for the `pop` (resp. `push`) instruction, where the non-0 component correspond to the stack

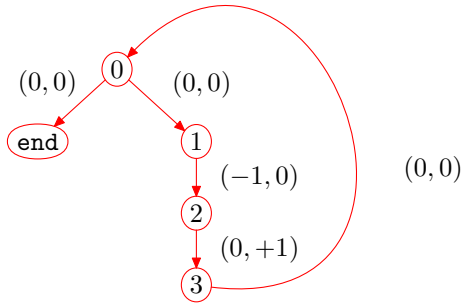

Fig. 9. A RCG for the reverse program

poped (resp. pushed). Other instructions have weight 0.

Example 8.7.

(1) If we apply this to the reverse program, that leads to $||\langle \texttt{IP}, \sigma\rangle|| = (|\sigma(\mathbf{l})|, |\sigma(\mathbf{l}')|)$. The resulting RCG (which is a VASS), is displayed on Figure 9. Since this VASS is uniformly terminating (there is no cycle of positive weight), the reverse program is also uniformly terminating.

(2) Let us consider the following program, working on integers (that is the alphabet is the set of 32 bits positive integers):

0 : `if` $\mathbf{i} = \mathbf{n}$ `then goto end`
1 : $\mathbf{i} := \mathbf{i} + 1$
2 : some instructions modifying neither $\mathbf{i}$ nor $\mathbf{n}$
3 : `goto` 0
`end` : `end`

This is simply a loop `for`$(;i<n;i{+}{+})$ (in a C-like syntax). If we consider a size function that simply takes the vector of the registers, that is $||\langle \texttt{IP}, \sigma\rangle|| = (\mathbf{i}, \mathbf{n})$, then the loop will have weight $(+1, 0)$ and thus lead to a cycle of positive weight. However, a clever analysis of the program could detect that inside the loop we must necessarily have $n - i > 0$ and thus suggest the size $||\langle \texttt{IP}, \sigma\rangle|| = \mathbf{n} - \mathbf{i}$. Using this, the loop has weight -1 and we can prove uniform termination of the program.
As stated, we do not address here the problem of finding a correct size function for a given program. This problem is undecidable in general. But invariants can often be automatically generated, usually by looking at the pre- and post-conditions of the loops.
Notice also that since this inequality must hold only in the loop (**i** and **n** could be reused outside), it can be useful to have a different size function out of the loop, hence different sets of valuations for each vertex.

This first termination analysis is close to the Size Change Termination [Lee et al. 2001] in the sense that the size of data is monitored and a well ordering on it ensure that it cannot decrease forever. It is sufficient to prove uniform termination of most common lists programs such as reversing a list or insertion sort. It is also, in some way, slightly more efficient than the original SCT because it can take into account not only decrease of the size but also increases, so that a program that would loop on something like `pop pop push` (2 `pop`s and 1 `push`) is not caught by SCT but is proved uniformly terminating with this

analysis. In this sense, it is closer to the SCT with difference constraints (δSCT) [Ben-Amram 2006].

This method is in PTIME, as we've shown, uniform termination of VASS is in PTIME. The original SCT, as well as δSCT, is PSPACE-complete. However, this simple method do not allow for data duplication or copy. Lee, Jones and Ben-Amram already claimed in the original SCT that there exists a poly-time algorithm for SCT dealing with "programs whose size-change graphs have in- and out-degrees bounded by 1". It is easy to check that VASS can only modelise such kind of programs accurately[4], hence the poly-time bound is not a big surprise.

Moreover, this method has a fixed definition of size and hence won't detect termination of programs whose termination argument does not depend on the decrease of the length of a list. Among other, any program working solely on integer (represented as letters of the alphabet) will not be analysed correctly.

8.2 Grounded resource Control Graph

Definition 8.8 *(RCG).* Let p be a program and G be its control flow graph. Let $\widehat{V}$ be a set of ground (and admissible) valuations (and $\prec$ be a well partial order on it). Let $||\bullet|| : \Theta \to \widehat{V}$ be a size function from states to valuations and $\widehat{V}_{\mathtt{lbl}}$ be the image by $||\bullet||$ of all states $\langle \mathtt{lbl}, \sigma\rangle$ for all stores σ. Let $V^+_{\mathtt{lbl}}$ be the set obtained by adding to $\widehat{V}_{\mathtt{lbl}}$ a least upper bound for each infinite subset and V^+ be the union of them.

For each i edge of G, let $\omega(i)$ be a function such that for all stores σ verifying $p \vdash \langle \mathtt{IP}, \sigma\rangle = \theta \xrightarrow{i} \theta'$, $||\theta'|| \preceq \omega(i)(||\theta||)$. Let V be the closure of V^+ by all the weighting functions $\omega(i)$.

The *Grounded Resource Control Graph* (Grounded-RCG) of p is the RSS build on G with weights $\omega(i)$ for each edge i, valuations V, admissible valuations V^+ (ordered by $\prec$) and ground valuations $\widehat{V}$. $V^+_{\mathtt{lbl}}$ being the admissible valuations for vertex $\mathtt{lbl}$.

As mentioned with RCG, sometimes the size function is such that no proper weight can be found for some instruction that will correctly carry on the size. This is solved here by the use of a non-ground valuation. Indeed, if there are no function $\omega(i) = f$ that verifies exactly the equality ($||\theta'|| = f(||\theta||)$) for all states with instruction pointer $\mathtt{lbl}$, then we can return the least upper bound of all the sizes. This is a way to mean that there is at this point no more information in the grounded-RCG on the actual memory and the valuation is not that much related to it. However, when looking for non uniform termination, we must have an oracle which is able to guess properly what the valuation (size of the state) should be.

LEMMA 8.9. *Let p be a program, G be its Grounded-RCG and $p \vdash \theta_0 \to \ldots \to \theta_n$ be an execution with trace t. Let $\widehat{v}_i = ||\theta_i||$ be an sequence of admissible ground valuations. There exists a ground-admissible walk starting at $(s_0, \widehat{v}_0)$, with trace t and $(\widehat{v})_{\mathbb{N}}$ as oracle.*

COROLLARY 8.10. *Let p be a program and G be its Grounded-RCG. If G is uniformly terminating, then p is also uniformly terminating.*

Example 8.11. The following program computes the exponential of an integer using

[4] And cannot even modelise all those programs due to the restriction on copying variables previously mentioned

an operator for multiplication by 2.

0 : $\mathbf{a} := \mathbf{b}$	3 : $\mathbf{b} := \texttt{twice}(\mathbf{b})$
1 : `if` $\mathbf{a} = 0$ `then goto end`	4 : `goto` 1
2 : $\mathbf{a} := \mathbf{a} - 1$	`end` : `end`

For some reason, one may want to choose the vector of the registers as a size function ($||\theta|| = (\mathbf{a}, \mathbf{b})$) and a VASS as RCG, either because this is used in a program where this size function is useful, or because things are decidable with VASS. However, both the copy and multiplication by two are not representable with a VASS. With a grounded VASS, however, this is doable. Indeed, both these instruction can return "unknown" (*i.e.* $+\infty$) for the corresponding register and let the oracle find out what the correct value should be. Using this, we have the grounded VASS of Figure 6 as an RCG for the program (modulo a couple of edges with weight 0). Since this grounded RCG is uniformly terminating, so is the exponentiation program.

9. δ-SIZE CHANGE TERMINATION

We explain here how to build RCG in order to perform the same kind of analysis as the Size-Change Termination with difference constraints (δSCT). We assume here that the reader is familiar with the original SCT work of Lee et al. [2001].

In this whole section, a given size function on states is assumed. We do not consider here the problem of finding a proper size function for each program (or family of programs).

9.1 Size-Change Graphs and matrices

Definition 9.1 *(Size Change graphs).* Let p be a program, and for each label $\texttt{lbl}_a$ in it consider a fixed integer k_a. Let $V_a = \mathbb{Z}^{k_a}$ and $V_a^+ = \mathbb{N}^{k_a}$ be sets of (admissible) valuations associated with each label and consider given a size function $||\bullet||$ such that for each label $\texttt{lbl}_a$ and for each store σ, $||\langle \texttt{lbl}_a, \sigma\rangle|| \in V_a^+$.

Let G be the CFG of p and $\texttt{lbl}_a \xrightarrow{i} \texttt{lbl}_b$ two nodes and an edge in it. The *Size Change Graph* (SCT graph) for i is a labelled directed bipartite graph with k_a input nodes $\{A_1, \cdots, A_{k_a}\}$ and k_b output nodes $\{B_1, \cdots, B_{k_b}\}$ and labelled arcs $A_j \xrightarrow{\delta} B_l$ such that for all stores σ_a, σ_b such that $p \vdash \theta_a = \langle \texttt{lbl}_a, \sigma_a\rangle \xrightarrow{i} \langle \texttt{lbl}_b, \sigma_b\rangle = \theta_b$ we have $||\theta_b||_l \leq ||\theta_a||_j + \delta$

The *Size Change Matrix* (SCT matrix) of i is the $k_a \times k_b$ matrix M such that $M_{j,l} = \delta$ if there is an arc $A_j \xrightarrow{\delta} B_l$ in the SCT graph of i and $+\infty$ otherwise.

Notice that this definition do not constrain too much the size function whereas the one used in the previous section was much more restricted. A very simple use would be, if the alphabet is the one of 32 bits positive integers to choose $||\langle \texttt{lbl}, \sigma\rangle|| = (\sigma(\mathbf{r}_1), \ldots, \sigma(\mathbf{r}_n))$ and thus find termination proofs for programs over integers. However, some more complicated relation between registers could also be provided in the size function. For example, if a loop is controlled by (with a `C`-like syntax): `for(i=0;i<j;i++)`, one of the component of the vector (for the corresponding labels) could be $\texttt{j} - \texttt{i}$. This allows for a more clever analysis and corresponds exactly to the functional inequalities of Avery [2006].

Notice also that there is not necessarily the same number of components for each label. This is indeed very useful, typically to avoid carrying over functional inequalities out of their scope. That is, if we consider again the previous loop, the inequality $\texttt{j} - \texttt{i} \geq 0$

only makes sense inside the loop and it would be a mistake to keep it out of the loop (especially if `i` and `j` are reused). Another typical example would be to consider local variables (and not take them into account out of their scope) or functional programs with different numbers of arguments for each function (as done in the original SCT).

This gives us an example on when separated sets of valuations for each label can be useful.

The uses of matrices rather than Size Change Graphs follows the works of Abel and Altenkirch [2002] where similar SCT matrices are used (but over a 3-valued set, thus mimicking the initial SCT and not the work with difference constraints).

Definition 9.2 *(Annotated Control Graphs).* Let p be a program. Its *Annotated Control Graph* (ACG) is the control flow graph $\mathcal{G}$ where each edge i has been labelled with the SCT matrix of i.

We now consider matrix multiplication over the $(\overline{\mathbb{Z}}, \min, +)$ semi-ring ($\overline{\mathbb{Z}} = \mathbb{Z} \bigcup \{+\infty\}$). We denote $\oplus = \min$ and $\otimes = +$ the operations over integers as well as the corresponding operations over matrices using usual matrices addition and multiplication.

Notice that in this case, the relation over the states becomes $||\theta_b|| \leq ||\theta_a|| \otimes M$ (where comparison is done component-wisely).

Definition 9.3. Let $\mathtt{lbl}_a \xrightarrow{i} \mathtt{lbl}_b \xrightarrow{j} \mathtt{lbl}_c$ be a path in an ACG $\mathcal{G}$ and M, N be the SCT matrices of i and j. The SCT matrix of path ij is the product $M \otimes N$

LEMMA 9.4. *The SCT matrix of a path verifies the same condition as the SCT matrix of an edge. That is, if $p \vdash \theta \xrightarrow{t} \theta'$ with trace t and M is the SCT matrix of t then $||\theta'|| \leq ||\theta|| \otimes M$.*

To each matrix $M \in \mathcal{M}_{m,n}(\overline{\mathbb{Z}})$, we associate a boolean matrix $\underline{M} \in \mathcal{M}_{m,n}(\mathcal{B})$ such that $\underline{M}_{i,j} = 0$ if $M_{i,j} = +\infty$ and $\underline{M}_{i,j} = 1$ otherwise. Notice that $\underline{\bullet}$ is a morphism, that is $\underline{M \otimes N} = \underline{M} \times \underline{N}$.

Definition 9.5 *(δSCT).* An annotated Control Graph $\mathcal{G}$ *does not* satisfies the δSCT condition if there exists a cycle in it whose SCT matrix M is such that:

(1) $\underline{M}$ is idempotent: $\underline{M} \times \underline{M} = \underline{M}$;

(2) and M is not decreasing: there is no strictly negative number on the diagonal of M.

THEOREM 9.6 (BEN-AMRAM [2006]). *Let p be a program and $\mathcal{G}$ be its ACG. If $\mathcal{G}$ satisfies the δSCT condition then p is uniformly terminating.*

Notice that this condition is undecidable in general. However, if the SCT graphs are *fan-in free*, that is in each row of each SCT matrix, there is at most one non-$+\infty$ coefficient, then the problem is PSPACE-complete. See [Ben-Amram 2006] for details. Notice that in this paper, Ben-Amram uses mostly SCT graphs and not SCT matrices. The translation from one to the other is, however, quite obvious. Similarly we present here directly a condition on the cycles of ACG without introducing the multipaths. This is close to the "graph algorithm" introduced in [Lee et al. 2001].

The simple Size Change Principle of Lee et al. [2001] can be seen as an approximation of the δSCT principle where only labels in $\{-1, 0, +\infty\}$ are used. Since this only gives way to finitely many different SCT matrices, this is decidable in general (PSPACE-complete).

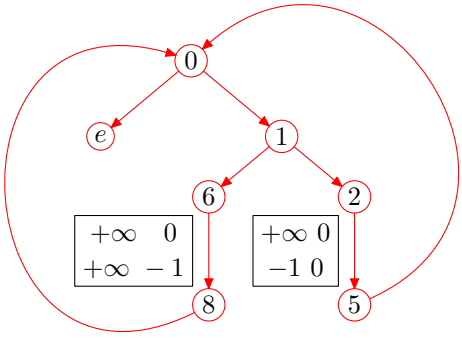

Fig. 10. Annotated Control Graph.

Example 9.7. Consider the following program (adapted from [Lee et al. 2001] fifth example):

$$
\begin{array}{l|l}
0 : \texttt{if } \mathbf{y} = 0 \texttt{ then goto end} & 5 : \texttt{goto} 0 \\
1 : \texttt{if } \mathbf{x} = 0 \texttt{ then goto } 6 & 6 : \mathbf{x} := \mathbf{y} \\
2 : \mathbf{a} := \mathbf{x} & 7 : \mathbf{y} := \mathbf{y} - 1 \\
3 : \mathbf{x} := \mathbf{y} & 8 : \texttt{goto} 0 \\
4 : \mathbf{y} := \mathbf{a} - 1 & \texttt{end} : \texttt{end}
\end{array}
$$

It can be proved terminating by choosing the size function $||\theta|| = (\mathbf{x}, \mathbf{y}, \mathbf{a})$. With this size, its Annotated Control Graph is displayed on Figure 10. For convenience reason, instructions $2-4$, as well as $6-7$ have been represented as a single edge (with a single matrix). This allow to completely forget register **a** and so use $(\mathbf{x}, \mathbf{y})$ as size. Similarly, the other SCT matrices are not depicted since they are the identity matrix. Since there is no cycle whose SCT matrix is both idempotent and not decreasing, the ACG satisfies the δSCT criterion and hence the program is uniformly terminating.

9.2 SCT as a Grounded-RCG

Now, let's take a closer look at the Annotated Control Graphs. These look quite close to RCG. Indeed, there is an underlying graph which is the control flow graph of a program and weight for each edge that are composed along paths, that is we would have $W = \mathcal{M}(\overline{\mathbb{Z}})$ and $\fatsemi = \otimes$. However, in order for them to be truly a RCG we need to define valuations (and admissible valuations).

Even more, the relation between sizes of states and the SCT matrices, as in Lemma 9.4 is exactly the one fulfilled by Grounded-RCG. Hence, we can simply choose $\mathbb{N}^k$ as admissible (and ground) valuations and this will leads to $\overline{\mathbb{N}}^k$ as admissible valuations (where $\overline{\mathbb{N}} = \mathbb{N} \bigcup \{+\infty\}$). the complete set of valuations will then be $\overline{\mathbb{Z}}^k$ where $\overline{\mathbb{Z}} = \mathbb{Z} \bigcup \{+\infty\}$. notice that we do not need here to add $-\infty$ because only upper bounds are considered.

Definition 9.8 *(Size Change RCG)*. Let p be a program and $||\bullet||$ be a size function such that for all stores σ, $||\langle \texttt{lbl}_a, \sigma \rangle|| \in \mathbb{N}^{k_a}$. Let G be the CFG of p and for each edge i in it let M_i be the corresponding SCT matrix.

The *Size Change RCG* (SCT-RCG) of p is the Grounded-RCG for p build with ground valuations $\mathbb{N}^{k_a}$, admissible valuations $\overline{\mathbb{N}}^{k_a}$ and valuations $\overline{\mathbb{Z}}^{k_a}$ for vertex $\texttt{lbl}_a$. The weight

for edge i is the SCT matrix M_i and we have $\circledast = \wr = \otimes$.

LEMMA 9.9. *Let p be a program, G be its SCT-RCG and $p \vdash \theta_0 \to \ldots \to \theta_n$ be an execution with trace t. There exists an admissible walk $(s_0, v_0) \to \ldots \to (s_n, v_n)$ with the same trace t such that for all i, $||\theta_i|| \leq v_i$.*

This is a direct consequence of Lemma 9.4.

COROLLARY 9.10. *Let p be a program, G be its SCT-RCG and $p \vdash \theta_0 \to \ldots \to \theta_n$ be an execution with trace t. There exists an admissible grounded-walk $\theta_0 \rightsquigarrow \widehat{\theta_0} \to \ldots \to \theta_n \rightsquigarrow \widehat{\theta_n}$ with the same trace t such that for all i, $||\theta_i|| = \widehat{v_i}$.*

So, if the SCT-RCG has no infinite ground-admissible walk, then the program uniformly terminates. As usual, the converse is not true. But we can do even better: indeed, uniform termination of the SCT-RCG is equivalent to verifying the δSCT criterion as we'll show now.

LEMMA 9.11. *Let p be a program and G be its SCT-RCG. Let c be a finite path in it, c is the underlying path of a ground-admissible walk.*

This is the equivalent of Lemma 5.3 for VASS and the proof is also similar.

PROOF. Consider a walk $\theta_0 \to \ldots \to \theta_n$ following c. Let $x_{i,j}$ be the ith component of the jth valuation. Each sub-path of c induces relations of the shape: $x_{a,b} = \min\{x_{c,d} + \delta_{a,b,c,d}\}$ where δs might be either an integer (positive or not) or $+\infty$.

So, if $x_{c,d} \geq -\delta_{a,b,c,d}$, then the corresponding term will be positive. If all are, then $x_{a,b}$ is also positive.

So, if $y_{i,j}$ is the ith component of the jth ground valuation in the sequence used as an oracle, putting $y_{c,d} = \max_{a,b}\{-\delta_{a,b,c,d}\}$ leads to the wanted ground-admissible walk. □

THEOREM 9.12. *Let p be a program, G be its SCT-RCG and $\mathcal{G}$ be its annotated control graph. G is uniformly terminating if and only if $\mathcal{G}$ verifies the δSCT condition.*

This theorem states that RCG encompass the Size Change Principle, even with some refinements such as the difference constraints of Ben-Amram [2006] or the functional inequalities of Avery [2006] and even allows to combine them easily.

Of course, the property is still undecidable in general but results on the δSCT tell that this is decidable if the matrices are fan-in free, that is at most one non-$+\infty$ per row.

PROOF. If $\mathcal{G}$ does not verify the δSCT condition, then there is a cycle c whose corresponding SCT matrix M is idempotent and not decreasing. This matrix is also the weight of c in G. By the previous lemma, this cycle is the underlying path of a ground-admissible walk. Let $(x_1, \cdots, x_n)$ and $(y_1, \cdots, y_n)$ be the first and last valuations in this walk. Since $\underline{M}$ is idempotent, y_i either depends on x_i or on one of the ground valuations introduced in the oracle. If y_i depend on x_i, since M is not decreasing we must have $y_i \geq x_i$.

If y_i depend on a valuation from the oracle, then we have again two cases. Either y_i depends on a component that was introduced to replace a $+\infty$ in a valuation or it depends on a component that was introduce but replace a non-∞ component of the valuation. Indeed, if the non ground valuation $(10, +\infty)$ is reached, then the oracle could well replace it with $(7, 24)$. Since the replacement valuation must be smaller than the replaced one, this is not harmful (we cannot arbitrarily increase some components on the valuation whose value is already known, only guess for the values that are unknown).

In the second case, we can always use a maximal oracle, in the sense that if a component of a non-ground valuation is not $+\infty$, then the oracle do not changes it (takes the maximum possible value for this component). Then, y_i is still dependant directly on x_i and the previous argument apply. In the first case, we can choose any value to replace $+\infty$ in the oracle valuation, so we can always find one that ensure that $y_i \geq x_i$.

So, if $\mathcal{G}$ does not verify the δSCT condition, then there is a ground-admissible cycle (c) such that $(s, \widehat{v}) \xrightarrow{c} (s, \widehat{v}')$ and $\widehat{v} \preceq \widehat{v}'$. Hence, by Theorem 6.15, G is not uniformly terminating (it is easy to see that G is indeed positive and monotonic).

Conversely, suppose that G is not uniformly terminating and consider a ground-admissible infinite walk $(s_0, v_0) \xrightarrow{i_1} \ldots \xrightarrow{i_n} (s_n, v_n) \ldots$ Let M_i be the SCT matrix associated with edge i.

Define a *2-set* to be a 2-elements set $\{t, t'\}$ of positives integers. Without loss of generality, $t < t'$.

Let the sign matrix $\overline{M}$ of M be a matrix over $\{-1, 0, +1, +\infty\}$ such that $\overline{M}_{i,j} = +\infty$ (resp. $+1$, 0, -1) if $M_{i,j} = +\infty$ (resp. is strictly positive, 0, strictly negative). Notice that $\underline{M} = \underline{\overline{M}}$.

Now, for each sign matrix $\overline{M}$, define the class $P_{\overline{M}}$ of 2-sets yielding it by:

$$P_{\overline{M}} = \{(t, t') | \overline{M} = \overline{M_{i_t} \otimes M_{i_{t+1}} \otimes \ldots \otimes M_{i_{t'-1}}}\}$$

The sets $P_{\overline{M}}$ forms a partition of the 2-sets. Indeed, they are mutually disjoint and every 2-set belongs to exactly one of them. Since the set of sign matrices (of bounded dimensions) is finite, it is also finite. Hence, by Ramsey's theorem, there is an infinite set of positive integers, T, such that all 2-sets $\{t, t'\}$ with $t, t' \in T$ are in the them class. Let P_{M° be this class.

Now, consider $t < t' < t'' \in T$. We have:

$$\begin{aligned}
\underline{M^\circ} &= \underline{\overline{M_{i_t} \otimes \ldots \otimes M_{i_{t''-1}}}} \\
&= \underline{\overline{M_{i_t} \otimes \ldots \otimes M_{i_{t''-1}}}} \\
&= \underline{\overline{M_{i_t} \otimes \ldots \otimes M_{i_{t'-1}}} \times \overline{M_{i_{t'}} \otimes \ldots \otimes M_{i_{t''-1}}}} \\
&= \underline{M^\circ} \times \underline{M^\circ}
\end{aligned}$$

Hence, $\underline{M^\circ}$ is idempotent.

Now, let $I_k = i_t$ such that t is the kth element of T (in increasing order). Let $N_j = M_{I_j} \otimes \ldots \otimes M_{I_{j+1}-1}$ and suppose that $\overline{N_j}_{k,k} = -1$, that is each N_j has a negative number on the diagonal. Then, if x_l is the kth component of v_l in the walk, for each $l < m \in T$ we must have $x_m < x_l$ (because $v_m = v_l \otimes N_j$ for some j. Since there are infinitely many integers in t, that would lead to an infinite decrease on one component of the valuation and the walk would no be admissible (or ground-admissible).

Hence, N_j does not have any negative number on the diagonal and $\underline{N_j}$ is idempotent. So $\mathcal{G}$ do not verify the δSCT condition. □

10. MORE ON MATRICES

If we use vectors as valuations and (usual) matrices multiplication as weights, we can define Matrices Multiplication Systems with States (MMSS) in a way similar to VASS. Admissible valuations will still be the ones in $\mathbb{N}^k$ but k is not fixed for the RSS and may depend on the current vertex.

Definition 10.1 *(Matrices Multiplication System with States).* A *Matrices Multiplication System with States* (MMSS) is a RSS $G = (G, V, V^+, W, \omega)$ where:

—$V_i = \mathbb{Z}^{k_i}$, $V_i^+ = \mathbb{N}^{k_i}$ for some constant k_i (depending on the vertex s_i).

—Weights are matrices with integer coefficients.

—$\overset{\circ}{9} = \circledast = \times$.

Using this, it is quite easy to model copy instructions of counters machines ($x := y$) simply by using the correct permutation matrix as a weight. To represent increment or decrement of a counter, an operation which was quite natural with VASS, we now need a small trick. Simply represent the n counters as a $n + 1$ components vector whose first component is always 1. Then, increment of decrement of a variable just becomes a linear combination of components of the vector which can perfectly done with matrices multiplication.

But there is even more. VASS are able to forbid a $x \neq 0$ branch of a test being taken in an admissible walk if x is 0 simply by decrementing x and then incrementing immediately after. The net effect is null but if x is 0, the intermediate valuation is not admissible. This is still doable with MMSS. VASS, like Petri nets, are however not able to test if a component is empty, that is forbid the $x = 0$ branch of a test to be taken if x is not 0.

With MMSS, we can perform this test to 0. It is indeed sufficient to multiply the correct component of the valuation by -1. If it was different from 0, then the resulting valuation will not be admissible.

So, using these tricks it is possible to perfectly model a counters machine by a MMSS: each execution of the machine will correspond to exactly one admissible walk in the MMSS and each admissible walk in the MMSS will correspond to exactly one execution of the machine.

This leads to the following theorem:

THEOREM 10.2. *Uniform termination of MMSS is not decidable.*

However, the study can go further. Indeed, using matrices of matrices (that is, tensors) we can represent the adjacency graph of a MMSS (a matrix where component (i, j) is the coefficient of the edge between vertices i and j). That is, a first order program can be represented as such kind of tensors. However, it would then be possible to uses these tensors (and tensors multiplication) in order to study second-order programs. In turn, the second order programs would probably be representable by a tensor (with more dimensions) and so one.

This would lead to a tensor algebra representing high order programs.

10.1 Polynomial time

Another interesting approach of program analysis using matrices is the one done by Niggl and Wunderlich [2006]. The programs they study are similar to our stacks machines except that the (conditional) jump is replaced by a fixed iteration structure (`loop`) where the number of iterations is bounded by the length of a given stack. It is quite easy to see that both models are very similar and can simulate one another without major trouble.

Then, they assign to each basic instruction a matrix, called a *certificate* which contains information on how to polynomially bound the size of the registers (or stacks) after the instruction by their size before executing the instruction. It appears that when sequencing

instructions, the certificate for the sequence turns out to be the product of the certificates for each instruction in turn. Certificates for loops are some kind of iterate of the certificate for the body and certificate for `if` statements are the maximum of the two branches.

Building the certificate of a program thus leads to a polynomial bound on the result depending on the inputs which can then be turned into a polynomial bound on the running time (depending on the shape of the loops).

So, these certificates can very well be expressed in a MMSS where the valuation would give information on the size of registers (depending on the size of the inputs of the program) and the weight of each instruction will be these certificates. This will exactly be a Resources Control Graph for the program. If the program is certified, then this RCG will be polynomially resource aware.

11. CONCLUSION

We have introduced a new generic framework for studying programs. This framework is highly adaptable via the size function and can thus study several properties of programs with the same global tool. Analysis apparently quite different such as the study of Non Size Increasing programs or the Size Change Termination can quite naturally be expressed in terms of Resource Control Graph, thus showing the adaptability of the tool.

Moreover, other analysis look like they can also be expressed in this way, thus giving hopes for a truly generic tool to express and study programs properties such as termination or complexity. It is even likely that high order could be studied that way, thus giving insights for a better comprehension of high order complexity.

Theory of algorithms is not well established. This work is really on the study of programs and not of functions. Further works in this direction will shed some light on the very nature of algorithms and hopefully give one day rise to a theoretical framework as solid as our knowledge of functions. Here, the study of MMSS and the tensors multiplication hints that a tensors algebra might be used as a mathematical background for a theory of algorithms and must then be pursued.

REFERENCES

ABEL, A. AND ALTENKIRCH, T. 2002. A Predicative Analysis of Structural Recursion. *Journal of Functional Programming 12*, 1 (Jan.), 1–41.

AMADIO, R., COUPET-GRIMAL, S., ZILIO, S. D., AND JAKUBIEC, L. 2004. A functional scenario for bytecode verification of resource bounds. In *Computer Science Logic, 12th International Workshop, CSL'04*. Springer.

ASPINALL, D. AND COMPAGNONI, A. 2003. Heap Bounded Assembly Language. *Journal of Automated Reasoning (Special Issue on Proof-Carrying Code) 31*, 261–302.

AVERY, J. 2006. Size-change termination and bound analysis. In *Functional and Logic Programming: 8th International Symposium, FLOPS 2006*. Lecture Notes in Computer Science. Springer. to appear.

BELLANTONI, S. AND COOK, S. 1992. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity 2*, 97–110.

BEN-AMRAM, A. 2006. Size-Change Termination with Difference Constraints. Tech. rep.

BONFANTE, G., MARION, J.-Y., AND MOYEN, J.-Y. 2004. Quasi-interpretation: a way to control resources. *Theoretical Computer Science*. Under revision.

COBHAM, A. 1962. The intrinsic computational difficulty of functions. In *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, Y. Bar-Hillel, Ed. North-Holland, Amsterdam, 24–30.

COLSON, L. 1998. Functions versus Algorithms. *EATCS Bulletin 65*. The logic in computer science column.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to Algorithms*. MIT Press.

GIRARD, J.-Y. 1987. Linear logic. *Theoretical Computer Science 50*, 1–102.

HOFMANN, M. 1999. Linear types and Non-Size Increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*. 464–473.

HOFMANN, M. 2000. A type system for bounded space and functional in-place update. *Nordic Journal of Computing 7*, 4, 258–289.

JONES, N. 2000. The expressive power of higher order types or, life without cons. *Journal of Functional Programming 11*, 1, 55–94.

LEE, C. S., JONES, N. D., AND BEN-AMRAM, A. M. 2001. The Size-Change Principle for Program Termination. In *Symposium on Principles of Programming Languages*. Vol. 28. ACM press, 81–92.

LEIVANT, D. AND MARION, J.-Y. 1993. Lambda Calculus Characterizations of Poly-Time. *Fundamenta Informaticae 19*, 1,2 (Sept.), 167–184.

MARION, J.-Y. AND MOYEN, J.-Y. 2006. Heap-size analysis for assembly programs. Tech. rep., LIPN.

MATIYASEVICH, Y. V. 1993. *Hilbert's 10th Problem*. Foundations of Computing Series. The MIT Press. MAT y 93:1 1.Ex.

MOYEN, J.-Y. 2003. Analyse de la complexité et transformation de programmes. Ph.D. thesis, University of Nancy 2.

NIGGL, K.-H. AND WUNDERLICH, H. 2006. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM Journal on Computing 35*, 5 (Mar.), 1122–1147. published electronically.

REUTENAUER, C. 1989. *Aspects mathématiques des réseaux de Petri*. Masson.

RICE, H. G. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc. 74*, 358–366.

ROGERS, H. 1967. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill. Reprint, MIT press 1987.

SHEPHERDSON, J. AND STURGIS, H. 1963. Computability of recursive functions. *Journal of the ACM 10*, 2, 217–255.

Received September 2006; revised September 2006; accepted hopefully someday